Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Framework für einen zusammengesetzten Datenspeicher

Roman Bitz

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Dr. rer. nat. Pascal Hirmer, Dipl.-Inf. Mathias Mormul
Beginn am:	3. Dezember 2018
Beendet am:	3. Juni 2018

Kurzfassung

Die Digitalisierung der Industrie 4.0 führt zu einer enormen Erzeugung an Daten. Um mit der Datenmenge zurechtzukommen, werden diese oft in die Cloud ausgelagert. Auch Zulieferer müssen Stammdaten ihrer gelieferten Produkte an Konzerne übertragen. Die Übertragung solcher Daten wird meist durch eine individuelle Softwarelösung übernommen, welche im ersten Schritt nur eine reine Datenübertragung von Datenspeicher zu Datenspeicher tätigt. Diese Ausarbeitung beschäftigt sich mit der Erstellung und Umsetzung eines Frameworks, welches eine automatisierte Datenübertragung zwischen Datenspeichern beliebiger Art ermöglicht. Bestehend aus einem Datenmodell zur Definition von verteilten Datenspeichern, einem erweiterbaren Adapterkonzept und die Nutzung von Datenfluss-Policies, können sich Datenübertragungen konfigurierbar steuern lassen. Die Bedienung und Konfiguration wurden durch die Nutzung einer Web UI, sowie den Datenstandards JSON und CSV ermöglicht. Das Framework wurde auf dem Open-Stack-System des IPVS der Universität Stuttgart bereitgestellt und evaluiert.

Abstract

The digitalization of industry 4.0 leads to an enormous generation of data. In order to cope with the volume of data, it is often outsourced to the cloud. Suppliers also have to transfer data of their delivered products to corporations. The transfer of such data is usually carried out by an individual software solution, which in the first step only transfers data from data storage to data storage. This paper deals with the creation and implementation of a framework that enables automated data transfer between data storage devices of any kind. Consisting of a data model for the definition of distributed data storages, an expandable adapter concept and the use of data flow policies, data transfers can be controlled in a configurable way. The operation and configuration was made possible by the use of a Web UI, as well as the data standards JSON and CSV. The framework was provided and evaluated on the Open Stack environment of the IPVS of the University of Stuttgart.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung und Motivation

Die Digitalisierung der Industrie 4.0 führt nicht nur zur einem höheren Grad der Automatisierung und Konnektivität, sondern erzeugt zudem eine enorme Menge an Daten [BWHT12]. Diese Datenmengen stellen nicht nur eine Belastung an die Hardware dar [CSGK16], sondern verlangen auch nach einem Management und dem richtigen Umgang mit den Daten. Da lokale Kapazitäten der Edge Cloud [PM17] nicht immer ausreichen, werden Daten oft in ein Cloud-Rechenzentrum [BDPP16] ausgelagert. Auch hier gilt, ohne ein durchdachtes Management der Daten, entsteht mehr Datenmüll als brauchbare Information. Weiterhin muss durch neue Datenschutzgesetze [Ros17] mehr auf den Umgang und die Verarbeitung der Daten Acht gegeben werden. So muss beispielsweise bekannt sein, wo sich die Daten befinden und wie lange sie dort bleiben.

Dieser Herausforderungen kann man durch einzelne eigenständige Lösungen bewältigen, jedoch führt das in der Summe zu einem nicht unerheblichem Aufwand und heterogenen Lösungsansätzen. An dieser Stelle ergibt sich ein Bedarf für eine einheitliche Lösung, welche die genannten Herausforderungen bewältigt.

1.1 Aufgabenstellung

Ziel dieser Arbeit ist das Design und die Entwicklung eines Frameworks, welches für die Verwaltung von Daten in verteilten Datenspeichern eingesetzt werden kann. Der Umgang mit verschiedenen Konstellationen von verteilten Datenspeichern soll das Kernprinzip des Frameworks sein. Als primäres Beispiel sind Systeme gemeint, die ihre Daten verteilt in der Edge und in der Cloud speichern. Das Framework muss vorwiegend mit diesen beiden Konzepten umgehen können. Dabei kann sich das Framework sowohl als eigene Entität unabhängig von den Konzepten positionieren, als auch Teil eines der beiden Konzepte sein. Durch Datenzugriffe auf die Edge-Cloud und eine Cloud-Rechenzentrum soll das Framework Daten verwalten und diese zwischen den Konzepten bewegen können. Um das Konzept des Frameworks zu evaluieren, soll dieses nach der Fertigstellung implementiert werden. Anschließend kann diese lauffähige Umsetzung in der Open-Stack-Umgebung des IPVS der Universität Stuttgart bereitgestellt und genutzt werden.

1.2 Gliederung

Die Ausarbeitung dieser Masterarbeit ist in acht Kapitel eingeteilt.

Kapitel 1 enthält eine Einführung in das Thema, sowie eine Aufgabenstellung der Ausarbeitung.

In Kapitel 2 wird auf Verwandte Arbeiten in den Bereichen Datenframeworks und Datenflusspolicies eingegangen.

In Kapitel 3 werden Grundlagenkonzepte wie Edge und Cloud Computing erläutert, sowie auf Alleinstellungsmerkmale von Framework erklärt.

Kapitel 4 listet alle Anforderungen an das zu entwickelnde Framework auf.

In Kapitel 5 werden allen Komponenten des Frameworks erklärt und die Funktionsweise anhand eines Beispiels verdeutlicht.

Kapitel 6 erläutert die Implementierung des Frameworks, in dem die Umsetzung der einzelnen Komponenten aufgezeigt wird.

Kapitel 7 erläutert die Evaluation des Frameworks anhand der Bereitstellung auf einer Open-Stack-Umgebung.

Kapitel 8 fasst die Arbeit zusammen und gibt einen Ausblick.

2 Verwandte Arbeiten und Projekte

Dieses Kapitel stellt einige wissenschaftliche Veröffentlichungen und Projekte vor, welche Überschneidungen in der Problemstellung oder deren Lösungsansätze mit dem Framework dieser Arbeit besitzen. Dafür werden zwei Themenbereiche klassifiziert. Unter Abschnitt 2.1 finden sich Frameworks, die Datenbewegung zwischen verschiedenen Systemen steuern. Abschnitt 2.2 beinhaltet Policy-Metamodelle, mit welchen sich Regeln für die Verwaltung von Daten festlegen lassen.

2.1 Frameworks für Datenbewegung

Dieser Abschnitt stellt einige Frameworks vor, die Datenbewegung zwischen den Bereichen IoT und Cloud Computing steuern.

2.1.1 An IoT-Oriented Data Storage Framework in Cloud Computing Platform

In diesem Paper von Jiang L. u. a. [JDC+14] wird ein Prototyp für eine effiziente Speicherung von IoT-Daten in der Cloud vorgestellt. Die behandelte Problemstellung ist das Volumen der IoT-Daten. Die Idee des Frameworks ist es mehreren Mandanten eine Möglichkeit zu bieten ihre IoT-Daten geordnet und automatisiert in der Cloud zu speichern. Das Framework ist in der Lage ankommende IoT Daten zu klassifizieren und in dafür passende Datenspeicher abzulegen. Strukturierte Daten werden in SQL- und NoSQL-Datenbanken abgelegt. Unstrukturierte Daten in einem *Hadoop Distributed File System* (HDFS)[1]. Als Beispiel wird ein Szenario genommen, in welchem Logistiksysteme mit angebundenen IoT-Sensoren das Framework nutzen. Der Aufbau des Frameworks besteht aus 4 Hauptmodulen, die sich wie folgt zusammensetzen.

- Das *Service management module* setzt sich aus einem ''Service Generator'', ''Services API'' und den ''REST services URIs'' zusammen. Ziel dieses Moduls ist es, Schnittstellen des Frameworks bereitzustellen und durch Web Application Description Language (WADL) konfigurierte Services zu generieren. An diesem Modul können dann weitere IoT-Applikationen angebunden oder Daten empfangen werden.
- Das *Resource configuration module* besteht aus mehreren Komponenten, die vor allem die Konfiguration des Frameworks abbilden. In diesem Modul wird die Mandantenfähigkeit durch Konfiguration im Framework ermöglicht. Dazu gehört eine Konfiguration für die Verteilung der Resourcen pro Mandant. In diesem Modul können auch die WADL-Konfigurationen für die Services abgelegt werden.

[1] https://hadoop.apache.org/

- Das dritte Modul *Data repository* abstrahiert die APIs unterschiedlicher SQL- und NoSQL-Datenbanken und integriert diese mit dem Framework. Dieses Modul enthält auch eine ''Object-Entity Mapping''-Komponente, welche die erhaltenen IoT-Werte auf Datenbankobjekte übersetzt.
- Das letzte Modul *File repository* sorgt für eine Anbindung von *Hadoop* 1, um unstrukturierte Daten abzuspeichern. Hierbei wird auch eine Versionskontrolle-Komponente genutzt, um bessere Übersicht über die Daten zu erhalten.

Das im Paper vorgestellte Framework erfüllt einen ähnlichen Zweck, wie das in dieser Ausarbeitung. Das Framework schafft es automatisiert Daten aus unterschiedlichen Quellen mit uneinheitlichem Syntax in verschiedene Datenspeicher abzulegen. Die Anbindung der Cloud-Datenspeicher geschieht durch Adapter, welche pro Datenbanksprache implementiert werden müssen. Diese Adapter übersetzen die internen API-Aufrufe des Frameworks in die entsprechende Datenbanksprache wie z.B. SQL. Die IoT-Sensoren werden nicht vom Framework direkt angebunden, sondern müssen durch separate IoT-Systeme ihre Daten an das Framework übersenden. Dafür müssen WADL-Konfigurationen von den IoT-System im Framework abgelegt werden, damit die passende Schnittstelle am Framework durch Service-Generierung erstellt werden kann. Dadurch müssen keine Clients oder Wrapper im IoT-System integriert werden. Jedoch ist mit den generierten HTTP-Services nur eine Art der Datenübertragung möglich. In der IoT genutzte Protokolle wie MQTT können dadurch nicht angebunden werden. Weiterhin ist das Framework darauf angewiesen Daten zu erhalten und bietet keine Möglichkeit selbst Daten anzufragen. Das Framework dieser Ausarbeitung ermöglicht die Anbindung an unterschiedliche Protokolle, sowie die freie Wahl der Übertragungsart.

2.1.2 Building a Framework for Internet of Things and Cloud Computing

Anon F. u. a. [ANHL14] stellen ein Framework-Modell vor, auf Basis dessen sich IoT-Applikationen bauen lassen. Hierbei wird eine Schichtarchitektur für IoT-Applikationen vorgeschlagen. Die oberste Schicht repräsentiert den *Cloud-Layer*, in welchem Datenbanken und die grundlegenden Berechnung der Applikation laufen sollen. Dieser *Cloud-Layer* ist an einen *Central-Hub-Layer* angebunden. Seine Aufgabe sind die Kommunikation und Übersetzung zwischen dem *Cloud-Layer* und dem *Device-Layer*. Der *Device-Layer* steht mit einer direkten Anbindung mit den Sensoren in Kontakt. Als Beispiele für den *Device-Layer* werden Geräte wie Raspberry Pi oder Xbee genannt, welche über Funkmodule oder Kabel Sensordaten mit standardisierten Protokollen übermitteln können.

In einem früheren Entwurf existierte der *Central-Hub-Layer* nicht im Framework. Er wurde nachträglich eingeführt, um die Anbindung des *Device-Layers* zu vereinfachen, die Schwächen des kabellosen Übertragungsverfahren auszugleichen und die Last von den Geräten zu nehmen. Der Sinn solcher Komponenten ist die Entkapselung spezifischer Umsetzung von der Applikationslogik. Die Applikation im *Cloud-Layer* kann unabhängig von dem Endgerät laufen. Die Komplexität einer solchen Komponente kann zunehmend wachsen, wenn eine stark heterogene Anzahl an Systemen unterstützt werden soll. In dieser Ausarbeitung wurde dieses Konzept angepasst, indem solche Komponenten gekapselt pro angebundenes System vorliegen und nicht als eine zentrale Schicht im Framework.

2.1.3 Rule based inference and action selection based on monitoring data in IoT

In der Masterarbeit von Fasihi A. [Fas16] wird sich mit regelbasiertem Datenfluss im IoT-Bereich beschäftigt. Hierzu wurde ein Framework umgesetzt, welches IoT-Daten durch Nutzung von maschinellem Lernen steuern kann. Der Schwerpunkt liegt vor allem darin, aus Echtzeitdaten bewusste Aktuatoren richtig zu bedienen. Dabei soll das maschinelle Lernen helfen vor allem unvorhergesehene Ereignisse den richtigen Aktuatoren korrekt zuzuordnen. Das Framework nutzt Interfaces, um unterschiedliche Teile des Frameworks austauschbar zu halten. So lässt sich auch der Algorithmus für das maschinelle Lernen austauschen.

Das Framework der Masterarbeit hat starke Parallelen zu dem Framework dieser Ausarbeitung. Der Einsatz von Interfaces ermöglicht beiden Frameworks die Anbindung beliebiger Komponenten und macht die Frameworks sehr flexibel. Ein Algorithmus für das maschinelle Lernen ist jedoch fester Bestandteil des Frameworks, weswegen immer Testdatensätze für die Nutzung des Frameworks verlangt sind. Damit ist die Steuerung der Daten stark vom richtigen Testdatensatz abhängig. Das Framework dieser Ausarbeitung steuert die Daten auf klar definierten Regeln, dies ermöglicht ein eindeutiges Management der Daten und benötigt keine Abhängigkeit eines Testdatensatzes.

2.2 Regelwerke für Datenflüsse

In diesem Abschnitt werden Regelwerke zur Steuerung von Daten vorgestellt.

2.2.1 PolicyDSL: Towards generic access control management based on a policy metamodel

Das Paper von Trinini'c B. u. a. [TSM+13] präsentiert *PolicyDSL*, ein Metamodell zur Definition von Access-Control-Modellen. Dazu wird die *Meta Object Facility* (MOF)-Metadaten Architektur herangezogen. *PolicyDSL* positioniert sich dabei in der M2-Ebene, in dem Bereich der Metamodelle. Damit ist PolicyDSL mit Metamodellen wie UML[2] zu vergleichen. *PolicyDSL* besteht aus vier Komponenten.

- **AC Entity** - Bildet Entitäten eines Access-Control-Modells ab. Beispielhafte Instanzen wären *User*, *Role* oder *Permission*.
- **AC Relation** - Repräsentiert Beziehungen zwischen *AC Entities*. Eine Beziehung wäre zum Beispiel die Zugehörigkeit eines Benutzers zu einer Rolle.
- **AC Concept** - Ist das Grundobjekt, von welchem *AC Entity* und *AC Relation* erben. Jedes AC-Concept-Objekt enthält *Attributes*.
- **Attribute** - Bilden Werte eines *AC-Concept*-Objekts ab. Hierbei handelt es sich um primitive Datentypen wie Zahlen oder boolesche Werte.

[2] https://www.uml.org/

Durch das Paper wird klar, dass es sich bei der Erstellung von Modellen unter Umständen lohnt eine Metamodell-Sprache zu nutzen, um die eigenen Modelle zu beschreiben bzw. zu erstellen. Dadurch kann man die Modelle nachträglich einfacher erweitern und weitere Modelle in einem einheitlichen Syntax definieren. Dies erleichtert die Einarbeitung Dritter in die Modelle, da nur das Metamodell verstanden werden muss. Jedoch ist *PolicyDSL* darauf ausgelegt reine Access-Policies-Modelle zu definieren. Weiterhin besteht keine Möglichkeit eine Fallunterscheidung auf Basis von kontextsensitiven Informationen zu erstellen. Im Framework dieser Ausarbeitung sind die Erstellung von Access-Policy-Modellen sowie von Datenfluss Policies möglich. Die Ergebnisse dieser Policies können durch kontextsensitive Informationen beeinflusst werden.

2.2.2 A Review and Comparison of Rule Languages and Rule-based Inference Engines for the Semantic Web

Rattanasawad T. u. a. [RSBS13] vergleichen mehrere Rule-Engine-Sprachen, sowie Rule-Engines miteinander, um anderen Forschern und Entwicklern die Entscheidung für eine Rule-Sprache bzw. Engine zu erleichtern. Der Vergleich fokussiert sich vor allem auf die Features der unterschiedlichen Sprachen und Engines. Dabei ist zu sehen, dass pro Feature mehrere Alternativen existieren. Bei dem Format der Sprachen gibt es sowohl XML-basierte, als auch Sprachen mit eigener Syntax. Die Engines liegen dagegen am häufigsten in Java vor, werden aber auch als Web Service angeboten.

Der Vergleich von unterschiedlichen Engines zeigt wie komplex eine Rule-Engine-Sprache sein kann. Um verschiedene Funktionen abzubilden wird auch meist ein dementsprechendes mächtiges Format für die Sprache genommen. Dies erhöht jedoch die Komplexität und die Einarbeitungszeit in solche Sprachen. Das Policy-Modell dieser Ausarbeitung ermöglicht es Regeln auf Basis von Programmierlogik zu erstellen. Dabei lassen sich die Regeln in einem simplen CSV-Format notieren.

3 Grundlagen

In diesem Kapitel werden grundlegende Begriffe und Konzepte erläutert, die im Kontext dieser Arbeit stehen.

3.1 Edge Computing

Unter Edge Computing versteht man die dezentrale Verarbeitung von Daten am Rande des Netzwerks, bevor diese in die Cloud übertragen werden [Sat17]. Dies ermöglicht die Analyse und Erfassung von Daten direkt an der Quelle, wo sie generiert wurden [SD16]. Dadurch können Daten schneller verarbeitet, die Latenzzeiten verringert und die Auslastung der Bandbreite niedrig gehalten werden. Einer der Hauptgründe für die Entstehung des Edge Computing ist der Anstieg von Datenerzeugern. Immer mehr IoT-Geräte werden an Netzwerke angeschlossen und erzeugen Daten. Diese Unmengen an Daten können aufgrund der limitierten Bandbreite nicht in ein Cloud-Rechenzentrum synchronisiert werden. Eine Verarbeitung vor Ort wird somit notwendig. Als Edge Computing wird oft jede beliebige Rechen- und Netzwerkressource bezeichnet die zwischen Datenerzeuger und einem Cloud-Rechenzentrum liegt [SCZ+16]. Dabei muss Edge nicht immer ein ''Mini-Datencenter'' sein, sondern kann auch ein Smartphone repräsentieren. Als Beispiel kann hierfür ein Fitness-Tracker herangezogen werden. Der Fitness-Tracker sammelt Rohdaten über die Aktivität des Nutzers. Durch die Anbindung an das Smartphone können die Rohdaten durch Berechnungen in aussagekräftige Ergebnisse, wie ''Anzahl der Schritte'' verwandelt werden. Als nächstes können die Ergebnisse in die Cloud mit dem Account des Nutzers synchronisiert werden.

3.2 Cloud Computing

Cloud Computing beschreibt das Bereitstellen von IT-Infrastrukturen über ein Netz, ohne diese auf dem lokalen Rechner installieren zu müssen. Gemäß der NIST-Definition [MG+11] wird Cloud Computing durch fünf Eigenschaften charakterisiert.

- On-demand Self Service - Provisionierung von Ressourcen ohne Interaktion mit dem Service Provider.
- Broad Network Access - Services sind nicht an einen bestimmten Client gebunden, sondern über das ganze Netz verfügbar.
- Resource Poolings - Anwender können eine Vielzahl von Ressourcen nutzen, ohne wissen zu müssen, wo diese vorliegen.

- Rapid Elasticity - Die Ressourcen stehen aus Anwendersicht unendlich zur Verfügung, dadurch dass die Services schnell und elastisch bereitgestellt werden können.
- Measured Services - Die Ressourcennutzung kann gemessen und überwacht werden und entsprechend den Cloud-Anwendern zur Verfügung gestellt werden.

Cloud Computing bietet drei verschiedene Kategorien von Angebotsarten an [FLR+14]. Eine der Angebotsarten ist Infrastructure as a Service (IaaS) [BJJ10]. Bei IaaS ist es dem Kunden möglich, IT-Ressourcen zu kaufen und darauf seine eigenen Services aufzubauen. Hierbei behält der Kunde die volle Kontrolle über das IT-System vom Betriebssystem aufwärts. In der Kategorie Platform as a Service (PaaS) [Law08] stellen PaaS-Provider den Kunden fertige Infrastrukturen zur Verfügung. Die Infrastrukturen können dann von den Kunden mithilfe von standardisierten Schnittstellen genutzt werden. Das hat zur Folge, dass der Kunde nur die Kontrolle über seine Anwendung hat, jedoch nicht über die Plattform auf der die Anwendung läuft. Die Kategorie Software as a Service (SaaS) [Cho07] beinhaltet alle Anwendungen, die die Kriterien des Cloud Computing erfüllen. Somit übergibt der Kunde die gesamte Kontrolle an den Cloud-Service-Provider.

3.3 Dependency Injection

Das Konzept von *Dependency Injection* wurde namentlich von Martin Fowler im Jahre 2004 auf seiner Webseite erläutert [DEP04]. Mit *Dependency Injection* wird eine Abhängigkeit, die eine Entität besitzt, erst zur Laufzeit berechnet und eingeführt. Im Code muss nicht mehr manuell die Abhängigkeit gesetzt werden, sondern wird durch die *Dependency Injection* durchgeführt [Pra09]. Die Information der Zusammenhänge geht dabei nicht verloren, sondern wird verlagert. Dazu kann z.B. eine Konfigurationsdatei genutzt werden, die die Abhängigkeiten aufführt und definiert wohin diese eingesetzt werden sollen. Dies führt jedoch zu einem gleichwertigen Aufwand wie das manuelle Setzen der Abhängigkeiten. Eine weitere Möglichkeit ist es die Information aus dem Code selbst herauszulesen. Sind Argumente im Konstruktor angegeben, so sind diese automatisch für die Initialisierung erforderlich und bilden eine Abhängigkeit für das konkrete Objekt. Nun kann dies durch einen Algorithmus ausgelesen und die passenden Abhängigkeiten eingefügt werden. Grundsätzlich gibt es drei Arten von *Dependency Injection*: *Constructor Injection*, *Setter Injection* und *Interface Injection* [DEP04].

Bei *Constructor Injection* werden die Konstruktor-Argumente als Abhängigkeiten angesehen. Um ein Objekt der Klasse zu bilden, werden die Abhängigkeiten initialisiert, sodass diese in den Konstruktor gegeben werden können. Verlangen die Abhängigkeiten für ihre Initialisierung Argumente, werden diese auch als Abhängigkeiten betrachtet und zuerst aufgelöst.

Mit *Setter Injection* werden die Attribute einer Klasse als Abhängigkeiten angesehen. Um diese Abhängigkeiten zu initialisieren werden die jeweiligen Setter-Funktionen aufgerufen. Besitzen die Abhängigkeiten eigene Attribute, werden diese zuerst injiziert.

Bei *Interface Injection* werden Schnittstellen mit spezifischen Funktionen definiert, welche für die Injektion genutzt werden. Klassen können diese Schnittstellen implementieren und somit die definierten Funktionen umsetzen. Anschließend wird jede Implementierung jeder Injektions-Schnittstelle als eine Abhängigkeit aufgelöst.

3.4 Inversion of Control

Frameworks und Libraries dienen beide der Erweiterung des Funktionsumfangs eines Programms. Libraries tun dies, indem sie eine Sammlung von Funktionen und Klassen bieten, auf welche der Entwickler direkt zugreifen kann. Folglich behält der Entwickler die Kontrolle über die Aufrufe [FRLB18]. Ein Framework basiert auf dem Paradigma des *Inversion of Control* [INV05]. Dabei geht es um die Abgabe der Kontrolle seiner eigenen Applikation an ein Framework. Das heißt der Datenfluss wird nicht mehr direkt von dem eigenen Code gesteuert sondern von einem Framework, welches meist einem gewissen Muster folgt. Mit *Dependency Injection* wird *Inversion of Control* ermöglicht, da hier eigene Klassen in ein Framework injiziert werden.

3.5 MOF - Meta-Object Facility

Meta-Object Facility (MOF) beschreibt eine spezielle Metadaten-Architektur, die von der Object Management Group (OMG) spezifiziert wurde [Obj18]. OMG ist ein Konsortium, welches unter anderem die Standards Unified Modeling Language (UML) und Business Process Model and Notation (BPMN) definiert hat. Eine MOF besteht aus mindestens 2 bis beliebig vielen Ebenen [Obj16]. Die Anzahl der Ebenen ist davon abhängig, welches System aus Objekt-Modellen man mit MOF beschreiben will [Poe06]. Jeder Ebene ist ein Modell zugeordnet, welches durch das Modell in der jeweils darüberliegenden Ebene beschrieben wird. Die letzte und höchste Ebene enthält ein MOF-Objekt, welches sich selber beschreiben kann [Obj18]. Anhand eines Beispiels mit UML lässt sich MOF verdeutlichen.

In Abbildung 3.1 sieht man UML in einer MOF dargestellt. Die unterste Ebene M0 beschreibt die Objekte der Realität, wie den Nutzer Alice. In der Ebene M1 wird der Nutzer durch eine Modellkomponente ''Nutzer'' eines Klassenmodells beschrieben. M2 ist die Ebene, in der sich UML einordnet. Diese Ebene beschreibt das Klassendiagramm. Die letzte Ebene M3 enthält nur noch ein MOF-Objekt, welches eine Art Wurzel darstellt. Diese Art von MOF-Objekt ist dafür vorgesehen, um der Ebenenstruktur eine letzte Ebene zu verleihen und eine endlose Abstraktionen zu verhindern.

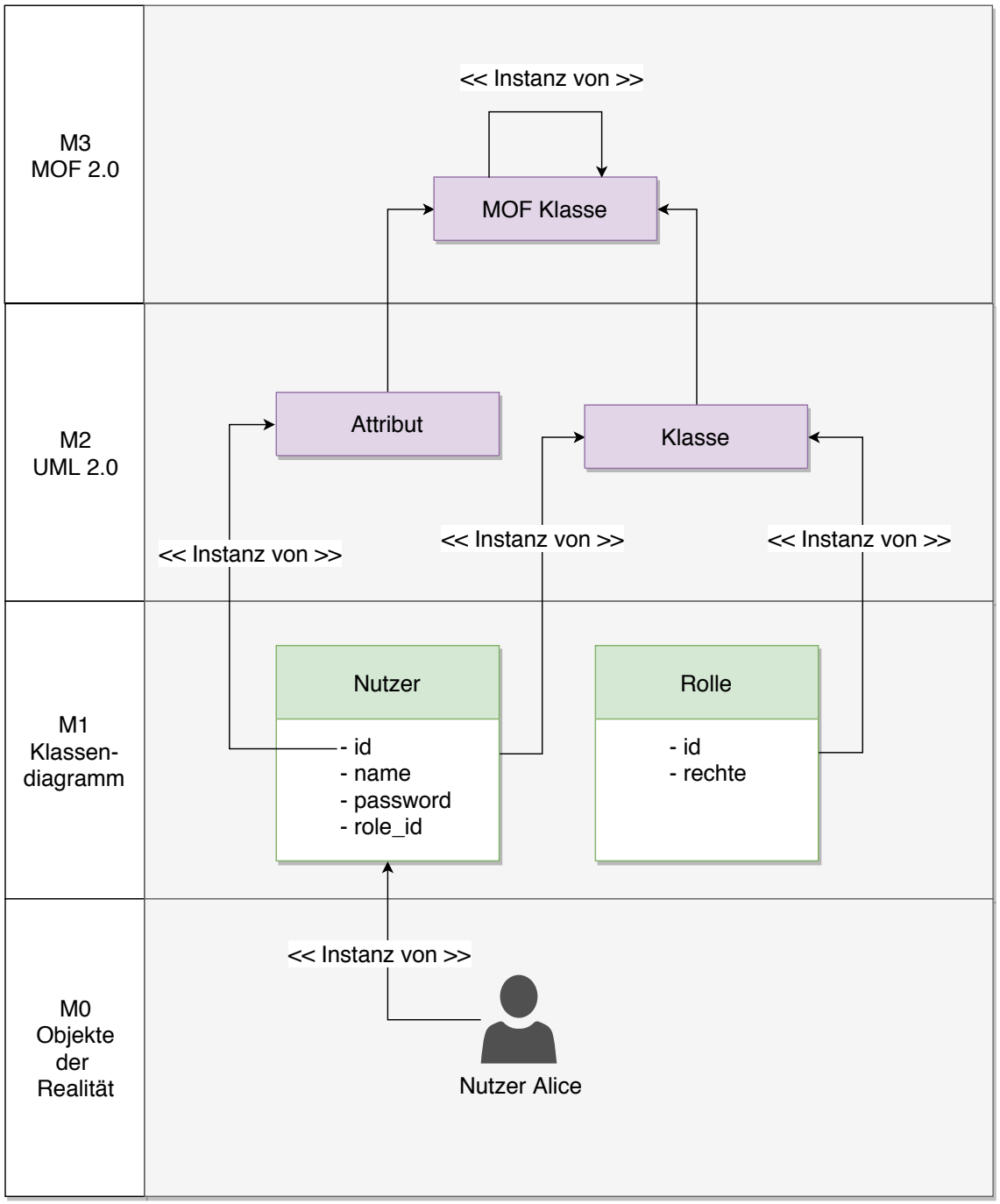


Abbildung 3.1: Darstellung von UML in einer MOF.

4 Anforderungen

Das Ziel dieser Ausarbeitung ist das Design und die Entwicklung eines Frameworks, welches für die Verwaltung von Daten zwischen verteilten Datenspeicher eingesetzt werden kann. Dazu wird in diesem Abschnitt vorab geklärt, welche Anforderungen das Framework zu erfüllen hat.

Abbildung 4.1 zeigt das Framework im Kontext von Edge und Cloud Computing. Mittig im Bild ist eine bereits vorhandene Verbindung zwischen der dargestellten Edge und Cloud abgebildet. Auf diese Verbindung wird vom Framework kein Einfluss genommen, sondern das Framework bildet eine weitere Entität, die an Edge und Cloud angebunden ist. Durch diese Anbindung soll ein Datenzugriff erfolgen, welcher dem Framework erlaubt Aktionen auf Daten auszuführen. Speziell kann das z.B. Sensordaten betreffen oder reguläre Datenbankdaten, welche in die Cloud verschoben werden müssen.

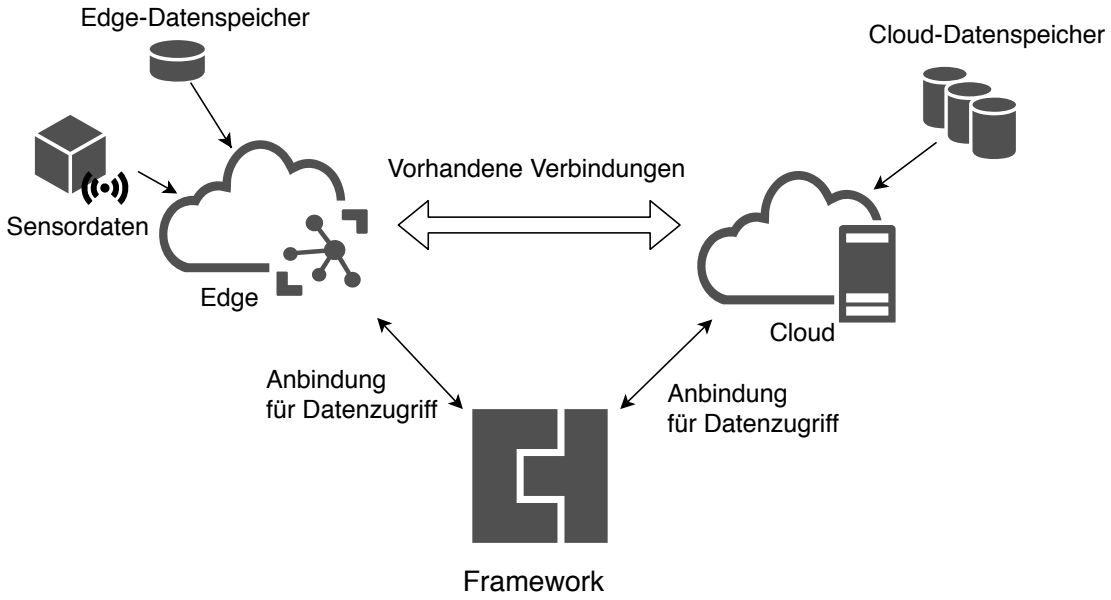


Abbildung 4.1: Das zu entwickelnde Framework im Kontext Edge Computing und Cloud Computing.

Verwaltung von verteilten Daten Eine grundlegende Anforderung an das Framework ist der Umgang und die Verwaltung von verteilten Datenspeichern und deren Daten. Die Verwaltung beinhaltet grundlegende Ausführungen von CRUD-Operationen auf die jeweiligen Daten. Daten zwischen Datenspeichern, die den gleichen Typ besitzen, sollten

verschiebbar sein. Da jeder Datenspeicher seine eigenen Funktionalitäten mit sich bringt, wie z.B. ''Backup'' oder ''Trigger'', sollte das Framework eine Möglichkeit bieten solche spezifischen Funktionalitäten nutzen zu können.

Persistente und dynamische Daten Das Framework soll sowohl persistente als auch dynamische Daten verwalten können. Bei persistenten Daten handelt es sich um Daten, die in einem persistenten Speicher abgelegt sind. Diese Art von Daten können zu jedem Zeitpunkt abgefragt werden, solange der Speicher verfügbar oder die Daten nicht gelöscht sind. Mit dynamischen Daten sollen alle Daten abgedeckt werden, welche nur für einen bestimmten Zeitraum verfügbar sind. Dynamische Daten können Echtzeitdaten sein wie z.B. Sensordaten, welche nicht zwischengespeichert, sondern zu einem Zeitpunkt gesendet werden und anschließend verloren gehen. Das Framework muss also Daten beziehen und empfangen können.

Konfiguration durch Policies Die Konfiguration von Regeln und die Verwaltung der Daten soll über Policies gehandhabt werden. Policies sollen definieren, unter welchen Bedingungen, welche Aktion mit den Daten ausgeführt werden soll und wohin diese abgelegt werden. Weiterhin soll durch Policies bestimmt werden ob gewisse Aktionen auf den Daten erlaubt oder verboten sind. Die Policies selbst sollten in einem definierten textuellem Dateistandard vorliegen, so dass diese auch von Menschen zu verstehen und zu bearbeiten sind. Die Policy-Struktur kann entweder selber definiert oder aus bereits passenden vorhandenen Projekten bzw. Forschungsergebnissen genutzt werden.

Erweiterbarkeit Um Zukunftssicherheit zu gewährleisten muss das Framework erweiterbar sein. Das Konzept des Frameworks sollte nachträgliche Erweiterungen durch z.B. neue Komponenten ins Design einbeziehen. Das Framework sollte nicht nur auf bisher vorhandenen Technologien und Konzepten anwendbar sein, sondern auch für zukünftige Datenspeicher-Konzepte erweiterbar bleiben. Jegliche Erweiterung muss mit den gegebenen Komponenten des Frameworks durchführbar sein und nicht eine grundlegende Änderung des Konzeptes verlangen.

5 Framework Modell

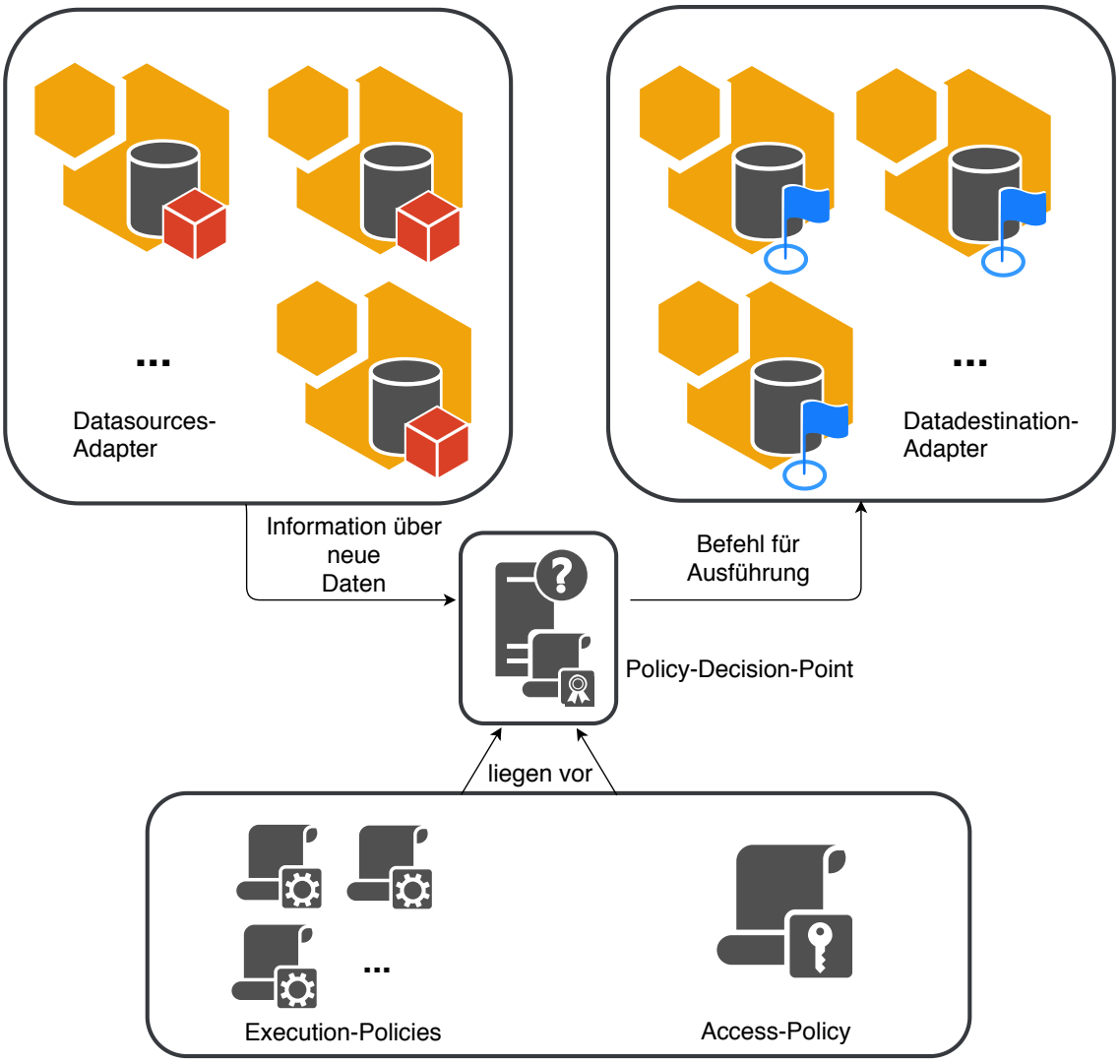


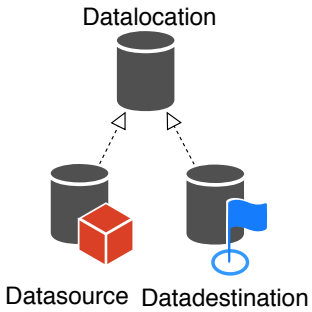
Abbildung 5.1: Übersicht aller Framework Komponenten

Dieses Kapitel stellt das Model des Frameworks vor. Hierbei werden abstrakte Komponenten und Vorgehensweisen auf der Modellebene definiert. Das Model bildet einen grundlegenden Lösungsvorschlag für die Verwaltung von zusammengesetzten Dateispeichern. Das Framework arbeitet mit einem Datenmodell, welches generisch Datenspeicher abbildet. In Abbildung 5.1 sind die zwei wichtigen Datenmodelle *Datasources Adapter* und *Datadestination-Adapter* jeweils oben links und oben recht dargestellt. Die *Datasources*

Adapter sind für die Ankunft von Daten zuständig und stellen die Dateneingabe für das Framework dar. *Datadestination-Adapter* bilden das Gegenstück in dem diese alle Datenziele für das Framework darstellen. Im unteren Teil der Abbildung ist das *Policy-Modell* des Frameworks zu sehen. Hierbei sind zwei Arten von Policies dargestellt. Mit *Execution Policies* werden Verbindungen zwischen *Datasources* und *Datadestinations* geknüpft, sowie Ausführungsbefehle. Die *Access-Policy* beschreibt dahingegen die Zugangsberechtigung für *Datasource-Adaptern* zu *Datadestination-Adaptern*. Das Datenmodell und das Policy-Modell kommen an der zentralen Modellkomponente *Policy-Decision-Point* zusammen. Hierbei werden alle ankommenden Daten der *Datasources Adapter* mithilfe des Policy-Modells evaluiert und an den richtigen *Datadestination-Adapter* weitergereicht.

5.1 Datenmodell

Mit dem Datenmodell wird definiert mit welchen Entitäten das Framework arbeiten soll. Eine Übersicht der einzelnen Datenmodelle und darin enthaltenen Informationen ist in der Tabelle der Abbildung 5.2 aufgelistet. Weiterhin ist hier eine Vererbungshierarchie zu sehen, in der *Datasource* und *Datadestination* von *Datalocation* erben.



Information	Datalocation	Datasource	Datadestination
Aufenthaltsort	✓	✓	✓
Zugang	✓	✓	✓
Lokale Navigation	✓	✓	✓
Kriterium		✓	
Funktionalitäten			✓

Abbildung 5.2: Vererbung des Datenmodells

5.1.1 Datalocation

Mit *Datalocation* wird ein Standort von Daten definiert. Eine *Datalocation* besitzt alle nötigen Informationen, um auf den Bereich der gewünschten Daten zugreifen zu können. Dafür müssen der **Aufenthaltsort** der Daten beschrieben und das ''Wo?'' der Daten eindeutig identifiziert sein. Der **Zugang** der Daten enthält alle nötigen Authentifizierungsinformationen, um auf die gewünschten Daten zugreifen zu können. Weiterhin ist eine **lokale Navigation** vorhanden. Die **lokale Navigation** stellt eine detaillierte Beschreibung zum exakten Bereich der Daten innerhalb des Aufenthaltsortes bereit. Somit setzt sich eine *Datalocation* aus den Informationen des Aufenthalts, Zugangs und einer lokalen Navigation zusammen. In Abbildung 5.2 ist *Datalocation* oben im Bild als ein grauer Zylinder dargestellt. Für ein besseres Verständnis ist im folgenden Paragraphen ein Beispiel gegeben.

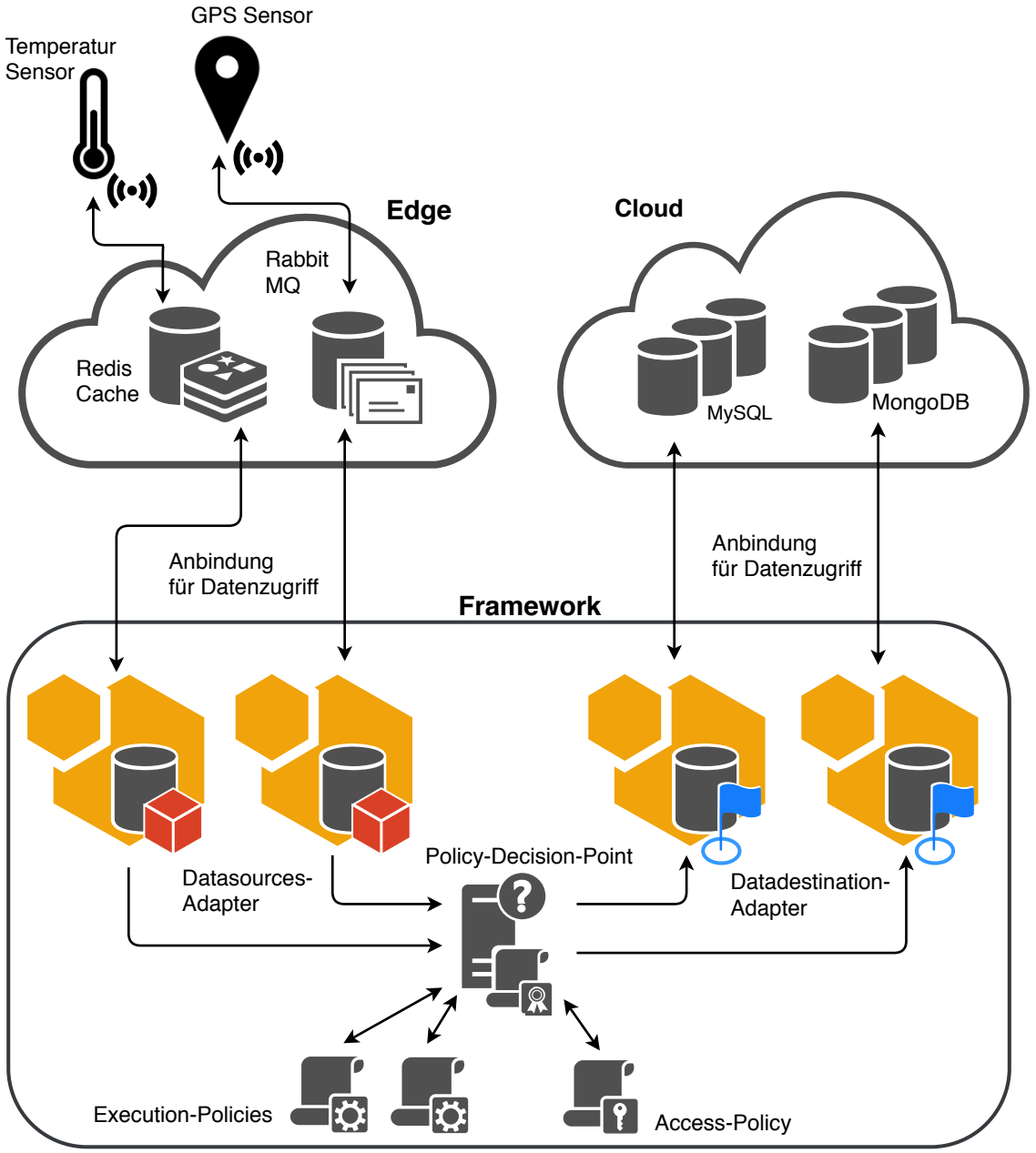


Abbildung 5.3: Beispielhafte Darstellung des Frameworks in einer Umgebung zwischen Edge und Cloud

Beispiel Das Beispiel in Abbildung 5.3 zeigt links zwei *Datasources*, welche die Informationen einer *Datalocation* beinhalten. Diese sind jeweils an einen Redis Cache[1] und RabbitMQ[2] angebunden. Die Information des Aufenthaltsortes sind die IP Adresse und der Port. Die Information über den Zugang sind der Benutzername und das Passwort. Die lokale Navigation ist der Topic Name im MQTT Broker bzw. eine Hash Name im Redis Cache. Durch diese Informationen kann das Framework auf die nötigen Daten zugreifen bzw. empfangen. Das gleiche Beispiel lässt sich analog auf eine relationale Datenbank anwenden. In diesem Fall wäre die lokale Navigation der Datenbankname und Tabellenname der jeweiligen Datensätze.

5.1.2 Datasource

Datasource definiert einen Dateneingang zum Framework. Eine *Datasource* enthält alle nötigen Informationen, um Daten für das Framework anzuliefern. Voraussetzungen für die Anlieferung von Daten sind der Aufenthaltsort und die Zugriffsmöglichkeit der Daten. Dementsprechend muss *Datasource* alle Informationen enthalten, die *Datalocation* besitzt. An dieser Stelle ist eine Vererbung der Informationen aus *Datalocation* zu *Datasource* gegeben. Um die gewünschten Datensätze zu konsumieren, braucht es ein **Kriterium** des Datensatzes. Das **Kriterium** ist eine Information anhand dessen ein Datensatz identifiziert werden kann. Hierbei handelt es sich weniger um die ID eines Datensatzes, sondern um ein Attribut oder Typ der Daten. Damit lassen sich dann z.B. Daten eines gewissen Typs verarbeiten. In Abbildung 5.2 ist *Datasource* links unten im Bild als ein grauer Zylinder mit einem roten Würfel dargestellt. Im Anschluss wird durch ein Beispiel *Datasource* verdeutlicht.

Beispiel In Abbildung 5.3 sieht man zwei *Datasources* im linken Abschnitt dargestellt. Diese sind an zwei Systeme, Redis Cache und RabbitMQ, in der Edge angebunden. Durch die vererbte Information aus *Datalocation* können IP-Adresse und Port, der Benutzername und Passwort, sowie der passende Topic bzw. Hash Namen genutzt werden, um eine Verbindung aufzubauen. Jedoch können diese Systeme viele heterogene Datensätze enthalten, die nicht den gleichen Syntax besitzen. Das Kriterium enthält einen Attributnamen, der für die Suche und Filterung der entsprechenden Datensätzen angewandt werden kann. Man könnte das Kriterium auch flexibler wählen, in dem man z.B. einen regulären Ausdruck wählt, welcher Teile der Datensätzen prüft.

5.1.3 Datadestination

Bei *Datadestination* handelt es sich um das Datenziel. Mit dem Datenziel wird festgelegt durch welche Funktionalitäten Daten ihren gewünschten Zielzustand erreichen. Ebenfalls wie bei *Datalocation* besitzt auch *Datadestination* Informationen über einen Datenstandort. Auch hier ist eine Vererbung der Informationen aus *Datalocation* zu *Datadestination*

[1] https://redis.io/
[2] https://www.rabbitmq.com/

gegeben. Das Alleinstellungsmerkmal von *Datadestination* ist ein Satz von Funktionalitäten. Dabei bestimmt eine Funktionalität was mit den Daten geschehen soll, damit diese ihr Ziel erreichen. Man kann bei *Datadestination* auch von einem Zielzustand der Daten sprechen, da bei einer Löschung oder Verarbeitung der Daten am gleichen Standort sich nur der Zustand der Daten ändert. In Abbildung 5.2 ist *Datadestination* rechts unten im Bild als ein grauer Zylinder mit einer blauen Flagge dargestellt. Im folgenden Paragraphen wird ein Beispiel zur Veranschaulichung vorgestellt.

Beispiel Das Beispiel in Abbildung 5.3 zeigt zwei *Datadestinations* im rechten Teil der Abbildung. Eine Datadestination ist an ein Cluster von MySQL-Datenbanken[3] angebunden, während die andere *Datadestination* an ein Cluster von MongoDB-Datenbanken[4] angebunden ist. Die vererbten Informationen über Aufenthaltsort, Zugang und lokaler Navigation sind analog aus den Beispielen für *Datalocation* und *Datasource* zu entnehmen. Der Satz an Funktionalitäten beinhaltet zwei Werte, ''Abspeicherung'' und ''Löschung''. Nun können ankommende Daten an der *Datadestination* mit der Funktionalität ''Abspeicherung'' in einer gewünschten Tabelle bzw. Collection gespeichert werden. Die Funktionalität ''Löschung'' kann dazu genutzt werden, vorhandene Daten zu entfernen. Eine weniger simple Funktionalität wäre "Transformieren", bei welcher der Syntax eines ankommenden Datensatzes abgeändert wird.

5.2 Adapter

Ein *Adapter* sorgt für die richtige Kommunikation zwischen Datenmodell und Datenspeicher. Das Datenmodell (Abschnitt 5.1) enthält alle nötigen Informationen, um mit Daten arbeiten zu können. Damit ist aber nicht geregelt, dass alle Informationen standardisiert und immer gleich anzuwenden sind. Abhängig vom Datenspeicher können z.B. Zugangsinformationen unterschiedliche Struktur besitzen. Der *Adapter* schließt diese Lücke, indem er Zuordnung und die richtige Anwendung der Informationen enthält. Somit existiert für jede *Datasource* und *Datadestination* jeweils ein Adapter, welcher die konkrete Kommunikation mit dem eigentlichen Datenspeicher regelt. Der *Adapter* ist in Abbildung 5.3 als gelbes hexagonales Muster hinter *Datadestinations* und *Datasources* dargestellt.

Beispiel Abbildung 5.3 zeigt vier *Adapter*, zwei *Datasource-Adapter* und zwei *Datadestination-Adapter*. Ein *Datasource-Adapter* ist mit einem Redis-Cache verbunden. Die *Datasource* enthält alle nötigen Informationen, um sich mit dem Broker zu verbinden. Der *Adapter* führt hierbei das entsprechende Protokoll für die Kommunikation mit dem Broker aus. So enthält der Adapter den offiziellen Client von Redis um die Kommunikation zu ermöglichen. Weiterhin ist ein *Datadestination-Adapter* mit einer MongoDB verbunden. Der *Adapter* kann hier nicht nur wie bei der *Datasource* die Kommunikation steuern,

[3] https://www.mysql.com/de/
[4] https://www.mongodb.com/

sondern auch die *Funktionalitäten* der *Datadestination* umsetzen. Wird die Funktionalität ''Löschen'' ausgelöst, so kann der *Adapter* entsprechend den gewünschten Datensatz löschen.

5.3 Policy-Modell

Im Framework sind Policies als Regelwerke zu betrachten, welche über die Nutzung und Verbindungen zwischen *Datasources* und *Datadestinations* bestimmen. Das Konzept ist grundlegend an klassische Security Policies [SS94] angelehnt. Jedoch wird hier das Konzept etwas adaptiert und für Datenfluss-Zwecke genutzt. Beide Policyarten sind auf dem *PERM Metamodel* aufgebaut und weisen dementsprechend den gleichen Aufbau auf. Durch eine unterschiedliche semantische Verwendung dieser Struktur ergeben sich unterschiedliche Funktionsweisen, wie z.B. Zugangskontrolle (*Access-Policy*) oder Verhaltenskontrolle (*Execution-Policy*).

5.3.1 Execution-Policy

Eine *Execution-Policy* enthält Regeln, die bestimmen mit welchem Datensatz welche Aktion ausgeführt werden soll. Dafür muss zunächst eine *Datasource* vermerkt sein. Durch die Nutzung der *Datasource* kann bestimmt werden, um welche eingehenden Datensätze es sich handelt. Weiterhin braucht es den Vermerk einer *Datadestination*, um das Ziel der Daten festzulegen. Anschließend kann durch die Notation einer *Funktionalität* der *Datadestination* festgelegt werden, was mit den Daten geschehen soll. Optional kann die *Execution-Policy* weitere Parameter festlegen, wie z.B. Attributwerte eines Datensatzes, welche als Voraussetzung der Policy gelten. Grundsätzlich stellt die *Execution-Policy* eine Verknüpfung zwischen *Datasource*, *Datadestination* und einer *Funktionalität* der *Datadestination* dar. In Abbildung 5.1 ist die grafische Darstellung der *Execution-Policy* links unten im Bild zu sehen.

Beispiel Die *Execution-Policy* liegt in Form einer Textdatei vor, die ihre Regelwerke als einzeilige Einträge enthält. In den Einträgen sind in einem Tupel notiert, mit welchen *Datasource*-Datensätzen, welche Funktionalitäten von *Datadestination* ausgeführt werden müssen. Weiterhin sind im Tupel noch weitere Informationen notiert, wie Attributwerte. Die Metainformationen eines ankommenden Datensatzes können dann mit dem Tupel abgeglichen werden. Enthält der Datensatz z.B. ein Attribut ''color'' mit dem Wert "red", wie es im Tupel vermerkt ist, kann diese Regel ausgeführt werden.

5.3.2 Access-Policy

Bei der *Access-Policy* handelt es sich um Regeln, welche die *Execution-Policy* durch Zugriffskontrolle beschränken. In der *Access-Policy* wird festgelegt mit welchen Datensätzen welche Funktionalitäten erlaubt sind. Im Kontext des Datenmodells bedeutet das, welche *Datasource* zu welcher *Datadestination* durch die *Execution-Policy* verbunden werden darf.

Dies kann durch Freigabe- als auch Verbotsregelungen definiert werden. Sind mehrere Einträge vorhanden, ist der *Effect* (Kapitel 5.4) verantwortlich für die entscheidende Ausgabe. Im Gegensatz zu den *Execution Policies* existiert nur eine einzige *Access-Policy*. Würden zwei *Access Policies* auf eine Anfrage unterschiedliche Ergebnisse liefern, bedürfte es einer zusätzlichen Konfiguration oder Hierarchie von *Access Policies*, um ein konkretes Ergebnis für die Anfrage auszugeben. In der Abbildung 5.1 ist die grafische Darstellung der *Access-Policy* rechts unten im Bild zu sehen.

Beispiel Eine *Access-Policy* kann z.B. in einer Textdatei notiert werden. Einzeilige Tupel stellen jeweils eine Freigabe- oder Verbotsregel dar. Ein Tupel enthält eine *Datasource*-Information, *Datadestination*-Information und einen Wert zur Bestimmung, ob es sich um eine Freigabe- oder Verbotsregel handelt. Anschließend müssen bei jeder ankommenden Anfrage die Einträge einmalig geprüft werden.

5.3.3 Policy-Decision-Point

Der *Policy-Decision-Point* ist eine Instanz, welche alle Policies besitzt und diese lesen, evaluieren und ausführen kann. Bei der Ankunft eines Datensatzes, welcher durch eine *Datasource* beschrieben ist, evaluiert der *Policy-Decision-Point* zunächst alle vorhandenen *Execution Policies*. Alle zutreffenden Einträge werden als nächstes an der *Access-Policy* autorisiert. Anschließend können die autorisierten *Execution Policies* ausgeführt werden. In Abbildung 5.1 ist der *Policy-Decision-Point* im Zusammenhang des Datenmodells dargestellt. Hier sieht man die Informationen von *Datasource-Adaptern* an den *Policy-Decision-Point* fließen. Anschließend kann der *Policy-Decision-Point* anhand der vorliegenden *Policies* eine Ausführung an die *Datadestination-Adapter* zuordnen.

5.4 PERM Metamodel

Das *PERM Metamodel* ist ein Policy-Metamodell entwickelt von Yang Luo bei seiner Arbeit an der Autorisierungs-Bibliothek ''Casbin'' [GIT18]. PERM steht für *Policy*, *Effect*, *Request*, *Matchers* und nennt damit die vier Grundkomponenten des Metamodells [PERM18]. Mit dem *PERM Metamodel* werden sowohl die *Execution Policies* als auch die *Access-Policy* des Frameworks erstellt. Die Entscheidung für die Nutzung des *PERM Metamodel* ist die Simplizität des Metamodells. PERM bedarf aufgrund der einfachen Struktur und der Nutzung von einzeiligen Definitionsschritten wenig Einarbeitungszeit sowohl für das Modell selbst als auch für die Policies dies es definiert.

```
1 [request_definition]
2 r = sourceID, gender
3
4 [policy_definition]
5 p = datasetID, actionID, targetID, attribute
6
7 [policy_effect]
8 e = some(where (p.eft == allow))
```

```
9
10 [matchers]
11 m = r.sourceID == p.datasetID && r.gender == p.attribute
```

Listing 5.1: PERM Model Beispiel

Listing 5.1 zeigt ein komplettes Beispiel eines PERM-Metamodells. Das Listing beginnt zunächst mit einer *Request*-Definition, in der Anfragen an Policies definiert werden. Anschließend wird die *Policy*-Komponente definiert, welche den Syntax einer Policy aufzeigt. Als dritter Abschnitt ist der *Effect* beschrieben, welcher das Aussageergebnis des Models festlegt. Im letzten Abschnitt ist die *Matchers*-Komponente zu sehen, mit der die Art und Weise des Vergleichs zwischen *Request* und *Policy* bestimmt wird. Hierbei ist auch zusehen, dass jede Komponente nur einmalig in einem Model vorkommt.

Policy Die Komponente *Policy* definiert, welche Struktur die einzelnen Policy-Einträge besitzen sollen. Das heißt hier wird die Syntax eines einzelnen Eintrags in der Policy festgelegt. Dies wird durch ein Tupel von Argumenten dargestellt. Das Tupel kann beliebig viele Argumente besitzen, jedoch mindestens eines.

```
1 # Modeldefinition PERM Policy
2 p = datasetID, actionID, targetID, attribute
3
4 # Passender Eintrag zur Definition
5 GPS_DATA_RABBITMQ, DELETE_DATA, MYSQL_DATABASE, android
```

Listing 5.2: PERM Policy Beispiel

In Listing 5.2 sieht man eine beispielhafte Definition der Komponente *Policy* und einen passenden validen Eintrag. In diesem Beispiel enthält die Policy vier Argumente. Der Eintrag listet die Argumente in der richtigen Reihenfolge auf. Zunächst wird eine Datensatzidentifikation verlangt, als nächstes eine Aktion abgefragt. An dritter Stelle wird ein Ziel verlangt und an vierter Stelle ein Attribut abgefragt. Eine mögliche Verarbeitung dieser Informationen kann z.B. die Löschung von Haustierdaten in einer MySQL-Datenbank sein, wenn die Haustiere das männliche Geschlecht besitzen.

Request Das Gegenstück zu der *Policy* bildet die *Request*-Komponente. Diese wird genau wie die *Policy*-Komponente durch ein Tupel definiert. Auch hier kann das Tupel beliebig viele Argumente besitzen, mindestens jedoch eins. Die *Request*-Komponente definiert den Syntax einer Anfrage an eine Policy. Diese muss nicht dem Aufbau der *Policy*-Komponente gleichen, sondern kann je nach benötigten Informationsgehalt passend definiert werden.

```
1 # Modeldefinition PERM Request
2 r = sourceID, phone
3
4 # Passender Eintrag zur Definition
5 GPS_DATA_RABBITMQ, iphone
```

Listing 5.3: PERM-Request Beispiel

In Listing 5.3 wird die Definition einer *Request*-Komponente aufgezeigt. Hierbei wird ein Tupel mit zwei Argumenten definiert. Der darunterliegende Eintrag hält sich an die Syntax der Definition und beschreibt zunächst eine Datensatzidentifikation und ein Attribut. Diese beiden Informationen müssen in einer Anfrage an eine Policy enthalten sein.

Effect Die Komponente *Effect* stellt die Auswirkung des Gesamtergebnisses dar [SYN18]. Sind mehrere Einträge zur gleichen Anfrage in der Policy vorhanden, so kann es vorkommen das zwei gegensätzliche Ausgaben zu einer Anfrage entstehen. Um trotzdem eine Gesamtaussage treffen zu können, liegt der *Effect* in folgenden Konfigurationen vor. Ein *Effect* vom Typ *allow-override* erlaubt nur eine Freigabe, wenn auch eine passende Policy mit einer Freigabe existiert. Existiert keine Freigabe, so ist das Gesamtergebnis immer ein *deny*. Ein *Effect* vom Typ *deny-override* evaluiert das mindestens keine Verbotsregelung existiert. In diesem Fall muss nicht einmal eine Freigaberegel existieren damit das Gesamtergebnis ein *allow* wird. Ist der *Effect* ein *Allow and no deny*, so darf keine Verbotsregel existieren und es muss mindestens eine Freigaberegel vorhanden sein.

```
1 # Modeldefinition PERM Effect
2 # Deny-override:
3 e = !some(where (p.eft == deny))
4
5 # Allow-override:
6 e = some(where (p.eft == allow))
7
8 # Allow and no deny:
9 e = some(where (p.eft == allow)) && !some(where (p.eft == deny))
```

Listing 5.4: PERM Effect Beispiel

Listing 5.4 zeigt alle Typen der *Effect*-Komponente auf. Die Schreibweise der einzelnen Statements ist dabei fest vorgegeben und beinhaltet keine zusätzliche Logik. Das heißt die kleinste Abänderung des Statements führt zu einer nicht validen *Effect*-Komponente.

Matcher Die *Matcher*-Komponente ist ein Ausdruck der zu einem booleschem Wert berechnet wird. Hier können *Policy*- und *Request*-Tupels miteinander verglichen werden, indem ein vorangegangenes ''r'' für *Request*-Tupel und ein ''p'' für *Policy*-Tupel geschrieben wird. Der Vergleich zwischen den Tupelwerten basiert auf klassischer Programmierlogik wie der bedingten Anweisung[5].

```
1 # Modeldefinition PERM Matcher
2 m = r.sourceID == p.datasetID && r.gender == p.attribute
```

Listing 5.5: PERM Matcher Beispiel

[5] http://www.inf.fh-flensburg.de/lang/prog/if.htm

In Listing 5.5 ist die Definition einer *Matcher*-Komponente dargestellt. In dem Beispiel werden zwei Argumente aus einer *Policy*-Komponente und einer *Request*-Komponente verglichen. Zunächst soll die ''sourceID'' aus der Request mit der ''datasetID'' aus der Policy übereinstimmen. Anschließend werden die Attribute ''gender'' mit dem Attribut ''attribute'' verglichen. Wenn beide Vergleiche wahr sind, bedeutet das, dass der *Request* valide ist und mit der *Policy* übereinstimmt.

6 Umsetzung des Frameworks

In diesem Kapitel wird auf die Umsetzung des Frameworks eingegangen. Dabei werden verschiedene Grundsteine erklärt, auf welchen die Implementierung des Frameworks basiert. Dazu wird die Architektur der Umsetzung erläutert, die Realisierung der Framework Modelle, sowie die Wahl der Deployment Lösung.

6.1 Architektur

Die Architektur der Umsetzung ist durch die Abbildungen 6.1 und 6.2 dargestellt. Die Umsetzung besteht aus den zwei Projekten ''Application'' und ''Domain''. Domain ist der Kern der Umsetzung, welches in Abbildung 6.1 zu sehen ist. Das Projekt Application besitzt Abhängigkeiten zu Domain, weswegen in Abbildung 6.2 die Domain in verkleinerter Form als Teil von Application abgebildet ist. Als Kern der Umsetzung implementiert Domain alle nötigen Komponenten aus der Modellierung des Frameworks (Kapitel 5). Application enthält alle Implementierungen der Schnittstellen aus Domain, sowie die Implementierung der jeweiligen Datenspeicher Adapter. Die Legenden der beiden Abbildungen sind am unteren Ende zu sehen und zeigen alle Elemente der Abbildungen. Pakete sind als dunkelgraue Kästen dargestellt und gruppieren thematisch gleiche Komponenten. Lilafarbige Komponenten stellen Schnittstellen dar, welche von anderen Komponenten implementiert werden müssen. Blaue Kästen bilden abstrakte Komponenten ab, welche ihre Funktionalitäten an andere Komponenten vererben müssen. Türkis farbige Kästen stellen implementierte Komponenten dar. Weiterhin sind noch zwei Arten von Verbindungen zwischen den einzelnen Komponenten möglich. Ein durchgezogener Pfeil signalisiert, dass eine Komponente ein Attribut der jeweils anderen Komponente darstellt. Ein gestrichelter Pfeil mit einer leeren Spitze zeigt eine Vererbung zwischen zwei Komponenten an.

Unter Domain findet sich oben in der Abbildung 6.1 das Paket ''Adapter''. Die enthaltenen Komponenten legen fest, wie ein Datenspeicher Adapter auszusehen hat. Durch ''IDataSourceAdapter'' und ''IDataDestinationAdapter'' werden zu implementierende Schnittstellen vorgegeben, während ''AbstractDataSourceAdapter'' und ''AbstractDataDestinationAdapter'' bereits implementierte Funktionen bereitstellen, die in einer erbenden Klasse vorhanden sein müssen. So enthält der *AbstractDataSourceAdapter* den ''PolicyDecisionPoint'' als Attribut, welcher bei Datenankunft benachrichtigt werden kann. Eine erbende Klasse enthält somit immer eine Anbindung an den *PolicyDecisionPoint*. Die ankommenden Daten können vom *PolicyDecisionPoint* durch die vorliegenden Klassen ''Enforcer'' und ''ExecutionEnforcer'' validiert und an den richtigen *AbstractDataDestinationAdapter* weitergereicht werden. Das Ganze wird dadurch ermöglicht, indem dem *PolicyDecisionPoint* eine Liste aller *AbstractDataDestinationAdapter* vorliegt. Im Paket ''Model'' liegen alle

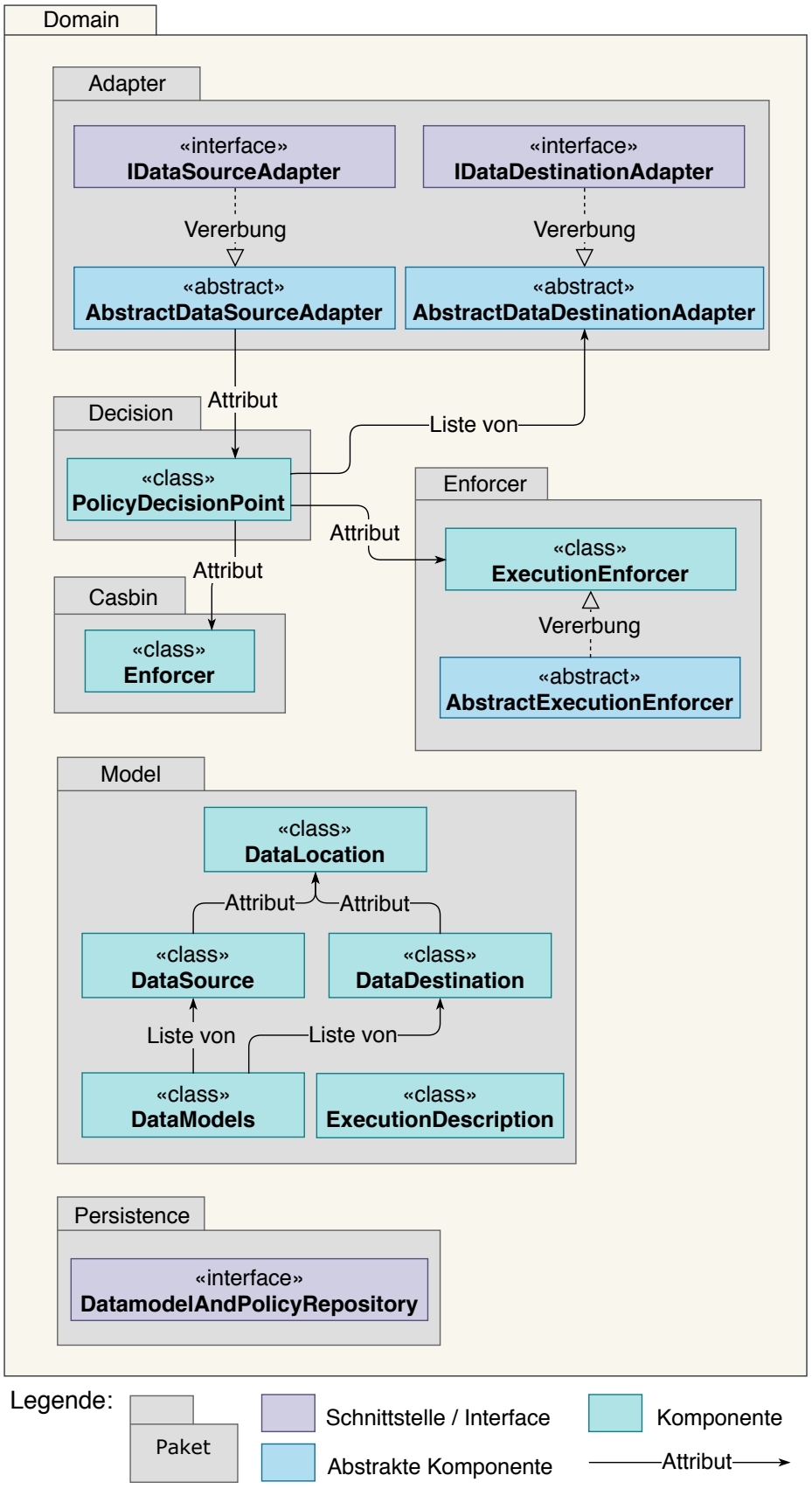


Abbildung 6.1: Darstellung aller wesentlichen Komponenten des Frameworks.

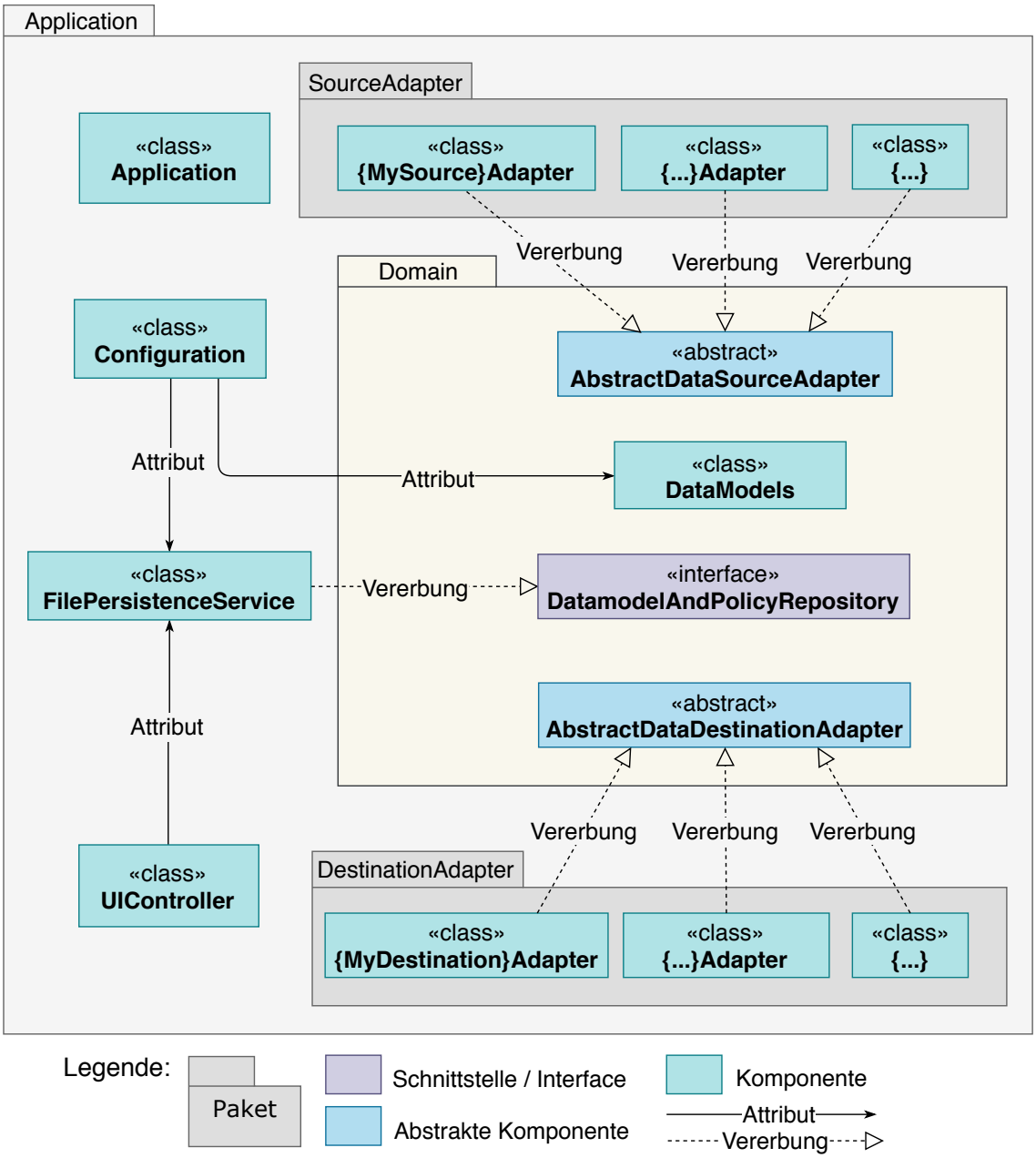


Abbildung 6.2: Darstellung aller wesentlichen Komponenten des Frameworks.

Datenklassen der Umsetzung. So sind hier die Datenmodelle des Frameworks ''DataLocation'', ''DataSource'' und ''DataDestination'' zu finden. Im Paket ''Persistence'' findet sich eine Schnittstelle zur Persistierung der Datenmodelle und Policies.

In der Abbildung 6.2 ist das Projekt Application mit dem Domain Kern dargestellt. Ganz oben in der Abbildung ist das Paket ''SourceAdapter'' zu sehen. Hier sind symbolisch mehrere Adapter Klassen dargestellt, die alle aus der Domain vom ''AbstractDataSourceAdapter'' erben. Dadurch müssen die Adapter alle benötigten Schnittstellen implementieren, um als ein *DataSourceAdapter* des Frameworks zu fungieren. Diese Vererbung führt auch zu einer Verlinkung (Abbildung 6.1) des *PolicyDecisionPoint* an den jeweiligen Adapter. Ein ähnliches Gefüge ergibt sich auch mit dem Paket ''DestinationAdapter''. Hierbei erben die Adapter vom ''AbstractDataDestinationAdapter'' und liegen dem *PolicyDecisionPoint* in einer Liste vor. Der Zustand der Applikation besteht aus der Zusammensetzung des Datenmodells und der Policies. Die Verwaltung dafür wird vom ''DatamodelAndPolicyRepository'' Interface vorgegeben und vom ''FilePersistenceService'' umgesetzt. Dadurch lässt sich der Zustand der Applikation auch in eine Datenbank auslagern, sofern ein passender Service die Implementierung des ''DatamodelAndPolicyRepository'' Interfaces umsetzt. Der ''FilePersistenceService'' ist weiterhin auch als ein Attribut beim ''UIController'' und ''Configuration'' hinterlegt, da an diesen Stellen auf die Policies und das Datenmodell zugegriffen wird. Letzteres sorgt beim Start der Umsetzung für die richtige Initialisierungsreihenfolge aller Komponenten und benötigt dementsprechend einen Lesezugriff. Der ''UIController'' steuert ein Frontend in welchem eine Bearbeitung des Datenmodells und der Policies möglich ist. Zuletzt ist ''Application'' oben links abgebildet, welches den Eintrittspunkt der ganzen Umsetzung beinhaltet.

6.1.1 Ports und Adapter Pattern (Hexagonale Architektur)

Die Architektur des Frameworks wurde nach dem *Ports und Adapter Pattern* entworfen. Das *Ports and Adapter Pattern* oder auch *Hexagonale Architektur* genannt, wurde von Dr. Alistair Cockburn in einem Artikel auf seiner Webseite im Jahre 2005 vorgestellt [HEX05]. Dabei handelt es sich um ein Strukturmuster für Software Architekturen. Das Ziel ist es die Logik der Applikation unabhängig und isolierbar zu halten. Das wird erreicht, indem die Logik der Applikation von ihrer spezifischen Umgebung entkoppelt wird. Konkret heißt das, dass die Logik an keine spezifische Datenbank, grafische Oberfläche, spezifischen Broker oder jegliche andere Softwareprodukte direkt gebunden ist.

Der Kern einer Applikation ist ihre Logik. Diese Logik arbeitet meist mit eigenen internen Objekten die für den Vorgang der Logik notwendig sind. Produziert die Logik eine Ausgabe, so wird diese an den jeweiligen *Adapter* weitergegeben. Bei einer Eingabe wird ein *Port* genutzt. Beide haben die gleiche Funktionalität: Die Vermittlung und Übersetzung zwischen den Objektmodellen der Logik und den Objektmodellen der jeweiligen Umgebung. Dabei ist zu beachten, dass die Logik keine direkte Anbindung an Ports oder Adapter besitzt und diese auch nicht kennt. Jedoch kennen die *Adapter* und *Ports* die Logik. Somit ist eine Abhängigkeit der *Adapter* und *Ports* von der Logik gegeben. Dadurch kann die Logik Schnittstellen vorgeben, welche der Adapter bzw. Port umzusetzen hat, damit eine Kommunikation mit der Logik gewährleistet ist.

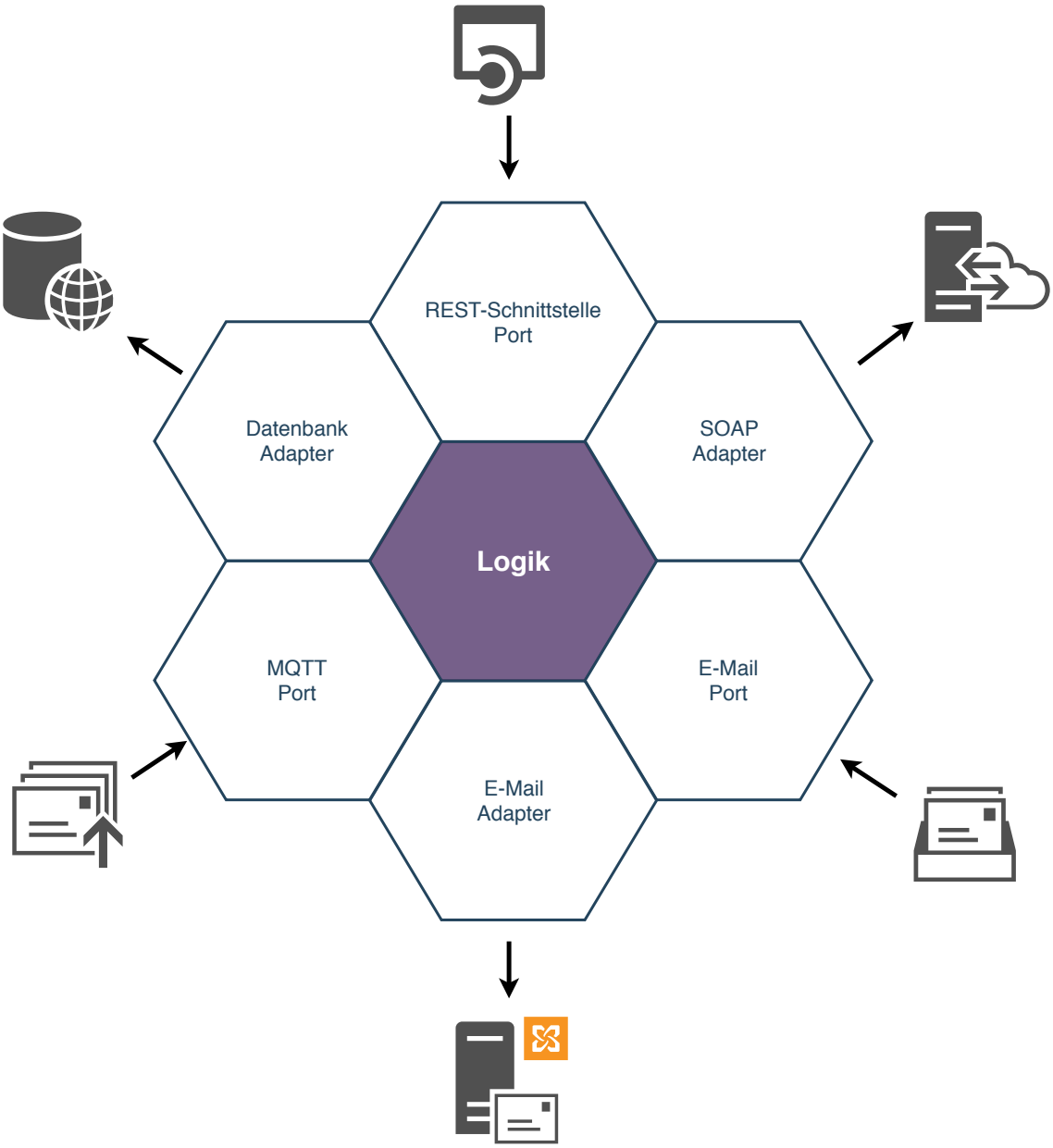


Abbildung 6.3: Darstellung einer hexagonalen Architektur.

Sollen ankommende Daten von der Logik der Applikation verarbeitet werden, so müssen diese durch einen *Port* der jeweiligen Schnittstelle durchgeführt werden. Hat die Applikation eine REST-Schnittstelle, existiert in der Applikation ein *Port* der diese REST-Schnittstelle nach außen bereitstellt und entgegenkommende Daten soweit vorbereitet, dass diese von der Logik der Applikation verarbeitet werden können. Entsprechend sind Datenbanken durch einen *Adapter* angebunden, der die Ausgaben der Logik in der jeweiligen Datenbank speichert. Ändert sich was an der Kommunikation mit der Datenbank, so muss nur der *Adapter* angepasst werden.

In Abbildung 6.3 ist eine *Hexagonale Architektur* visualisiert. Hier sieht man die Logik als farbiges Sechseck in der Mitte der Abbildung dargestellt. Umgeben ist die Logik von verschiedenen *Adaptern* und *Ports* die jeweils auch als Sechsecke abgebildet sind. Diese regeln den spezifischen Verkehr mit den jeweiligen Entitäten wie Datenbank, MQTT Brokern oder E-Mail Versand.

Das Pattern ist in der Umsetzung des Frameworks wie folgt umgesetzt. Das Projekt ''Domain'' reflektiert den beschriebenen Logikkern des Pattern. Alle Komponenten die vom ''AbstractDataSourceAdapter'' im Projekt ''Application'' erben stellen die *Ports* des Pattern dar. Alle Komponenten die vom ''AbstractDataDestinationAdapter'' erben stellen die *Adapter* des Pattern dar. Daraus ergibt sich wie im Pattern nur eine Abhängigkeit des ''Application'' Projekt vom ''Domain'' Projekt. Damit wäre das ''Application'' Projekt austauschbar ohne Änderung des ''Domain'' Projektes.

6.1.2 Nutzung von Dependency Injection

Einer der Grundsteine der Umsetzung ist die Nutzung von *Dependency Injection*(Kapitel 3.3). Dieses Verfahren wird genutzt um implementierte Instanzen von *AbstractDataSource-Adapter* im *Policy-Decision-Point* zu hinterlegen.

In Abbildung 6.4 ist eine *Dependency Injection* vom Framework dargestellt. Die grauen Blöcke repräsentieren zwei Klassen des Frameworks. Dabei ist die Abstrakte Klasse *AbstractDataSourceAdapter* als ein Attribut in der Klasse *Policy-Decision-Point* hinterlegt. Die rote Klasse *MySQL-Adapter* gehört nicht zum Framework und wurde nachträglich erstellt, um eine MySQL Datenbank anzubinden. Die Klasse erbt von der Framework Klasse *AbstractDataSourceAdapter*. Das *Spring Framework* nutzt jetzt das Verfahren der *Dependency Injection* um eine instanziierbare Klasse von *AbstractDataSourceAdapter* zu injizieren. Dadurch wird *MySQL-Adapter* in *Policy Descicion Point* als ein Attribut injiziert. *Policy-Decision-Point* erhält somit die Kontrolle über *MySQL-Adapter*, obwohl es zum Entwurfszeitpunkt die Klasse nicht kannte.

6.2 Umsetzung der Modellkomponenten

In diesem Abschnitt wird beschrieben, wie das Datenmodell, Policy-Modell, sowie die Adapter umgesetzt worden sind.

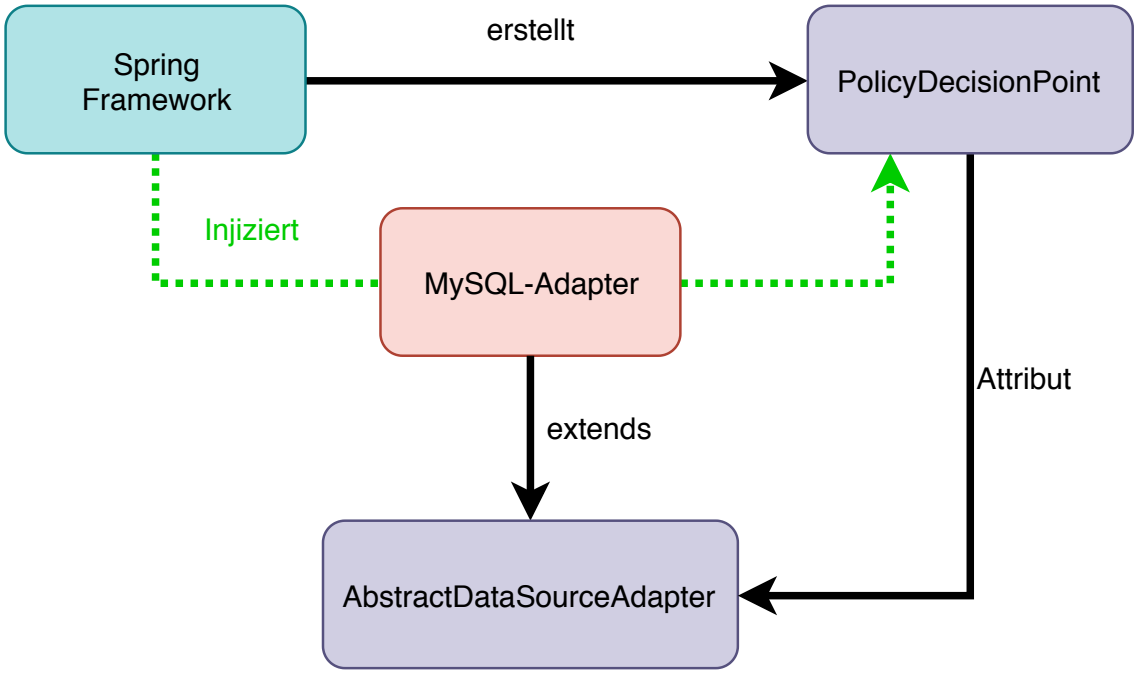


Abbildung 6.4: Darstellung von Dependency Injection

6.2.1 Umsetzung des Datenmodells

Das in Kapitel 5.1 vom Framework definierte Datenmodell wurde als JSON umgesetzt. Die Umsetzung des Datenmodells besteht aus zwei JSON Dateien. Die Datei ''datasource.json'' beinhaltet alle *Datasources* Informationen. Analog enthält ''datadestination.json'' alle Informationen von *Datadestination*. Für das Modell *Datalocation* existiert keine separate Datei. Die Informationen aus *Datalocation* sind in den Dateien ''datasource.json'' und ''datadestination.json'' inkludiert. Dieser Schritt verringert die Komplexität des Datenmodells, da es eine Referenzierung zwischen unterschiedlichen JSON Dateien obsolet macht. Gleichzeitig sind alle nötigen Informationen zu einer *Datasource* bzw. *Datadestination* in einem JSON Objekt dargestellt. Im folgenden werden die Dateien ''datadestination.json'' und ''datasource.json'' beschrieben, um die Umsetzung des Datenmodells zu verdeutlichen.

```
1  [
2    {
3      "_id": "GPS_DATA_RABBITMQ",
4      "location": {
5        "host": "localhost",
6        "port": "5672",
7        "user": "rabbitmq",
8        "pass": "rabbitmqpwd",
9        "entrypoint": "/",
10       "path": "gps",
11       "attr": {
12         "erlangcookie": "SWQOKODSQALRPCLNMEQG",
13         "vhost": "/",
```

```
14          "hostname": "rabbit1"
15        }
16      },
17      "criteria": "something=1234",
18      "attr": {
19        "key1": "optional value here",
20        "key2": "optional value here"
21      }
22    },
23    {
24      "_id": "TEMP_DATA_REDIS",
25      "location": {
26        "host": "localhost",
27        "port": "3306",
28        "user": "root",
29        "pass": "mysqlrootpwd",
30        "entrypoint": "mydatabase",
31        "path": "tablename",
32        "attr": {}
33      },
34      "criteria": "",
35      "attr": {
36        "key1": "optional value here",
37        "key2": "optional value here"
38      }
39    }
40  ]
```

Listing 6.1: Datasource.json mit zwei Datasource Einträgen

Listing 6.1 zeigt eine ''datasources.json'' mit zwei Einträgen. Die komplette Datei besteht aus einem JSON Array, welches einzelne JSON Objekte enthält, die jeweils eine *Datasource* repräsentieren. Ein solches JSON Objekt enthält das Attribut ''_id'', welches das Objekt von allen anderen *Datasource* Objekten identifiziert. Das Attribut ''location'' enthält ein neues verschachteltes JSON Objekt, welches eine *Datalocation* repräsentiert. Die Attribute ''host'' und ''port'' bilden die Information des *Aufenthaltsorts*, ''user'' und ''pass'' bilden die Information des *Zugangs* und ''entrypoint'' und ''path'' bildet die Information der *lokalen Navigation* ab. Das Attribut ''attr'' enthält ein JSON Objekt, in welchem optionale Attribute abgelegt werden können. Dies bietet eine Möglichkeit zusätzliche Informationen zu der jeweiligen *Datalocation* festzulegen, falls die bisherigen Attribute nicht ausreichend für diese sind. Das Alleinstellungsmerkmal einer *Datasource* ist die Information über das *Kriterium*. Dieses wird durch das Attribut ''criteria'' repräsentiert. Weiterhin enthält das *Datasource* JSON Objekt ein Attribut ''attr'', in welchem optionale Key-Value Attribute abgelegt werden können.

```
1   [
2     {
3       "_id": "MYSQL",
```

```
4       "location": {
5         "host": "localhost",
6         "port": "3306",
7         "user": "root",
8         "pass": "mysqlrootpwd",
9         "entrypoint": "mydatabase",
10        "attr": {},
11        "path": "temperatur"
12      },
13      "functionalities": [
14        "INSERT_DATA",
15        "DELETE_DATA"
16      ],
17      "attr": {}
18    },
19    {
20      "_id": "MONGODB",
21      "location": {
22        "host": "localhost",
23        "port": "27017",
24        "user": "root",
25        "pass": "pass",
26        "entrypoint": "admin",
27        "attr": {},
28        "path": "mydatabase.gps"
29      },
30      "functionalities": [
31        "SYNC_DATA",
32        "DELETE_DATA",
33        "BACKUP_TO_ARCHIVE"
34      ],
35      "attr": {}
36    }
37  ]
```

Listing 6.2: Datadestinations.json mit zwei Datadestination Einträgen

Listing 6.2 stellt eine ''datadestination.json'' mit zwei Einträgen dar. Analog zum *Datasource* Beispiel besteht die Datei aus einem JSON Array, welches einzelne JSON Objekte enthält, die jeweils eine *Datadestination* repräsentieren. Ein einzelnes *Datadestination* JSON Objekt besitzt jeweils ein Attribut ''_id'', mit dessen Hilfe die einzelnen Objekte voneinander unterschieden werden können. Das Attribut ''location'' repräsentiert eine *Datalocation*. Die Beschreibung der einzelnen Attribute des *Datalocation* JSON Objekts ist analog aus der Beschreibung des Listing 6.1 zu entnehmen. Das Attribut ''functionalities'' beschreibt die *Funktionalitäten* der *Datadestination*. Dieses ist als JSON Array umgesetzt, das die einzelnen Funktionalitäten als String beinhaltet. Das Attribut ''attr'' kann optionale Key-Value Attribute enthalten, um Zusatzinformationen abzuspeichern.

6.2.2 Umsetzung der Policies

Für die Umsetzung der Policies wurde das *PERM Metamodel* (Kapitel 5.4) benutzt. Wie im Framework beschrieben, wurden in der Umsetzung eine *Access-Policy* und mehrere *Execution Policies* erstellt. Alle erstellten Policies sind nachträglich editierbar und werden beim Start der Umsetzung einmalig eingelesen. Aufgrund des *PERM Metamodels* existieren pro Policy zwei Dateien. Eine Konfigurationsdatei mit der Dateiendung ''.conf'', welcher Policysyntax, Funktionalität und Anfragesyntax der Policy beschreibt. Durch eine CSV Datei liegen die einzelnen Regeln einer Policy vor. Eine Regel wird pro Zeile in der CSV definiert.

```
1  [request_definition]
2  r = datasourceid, functionality, datadestinationid
3
4  [policy_definition]
5  p = datasourceid, functionality, datadestinationid, eft
6
7  [policy_effect]
8  e = !some(where (p.eft == deny))
9
10 [matchers]
11 m = r.datasourceid == p.datasourceid && keyMatch(r.functionality, p.functionality) &&
   keyMatch(r.datadestinationid, p.datadestinationid)
```

Listing 6.3: Konfigurationsdatei der Access-Policy

In Listing 6.3 ist die Konfigurationsdatei der Access-Policy zu sehen. Anfragen an die Policy sind in diesem Fall als dreistelliger Tupel definiert. Hier wird verlangt, dass der Syntax einer Anfrage an erster Stelle eine *Datasource* ID, an zweiter Stelle eine Funktionalität der *Datadestination* und an dritter Stelle eine *Datadestination* ID enthalten muss. Der Syntax der Policy ist ähnlich der Anfrage definiert mit der Ausnahme, dass an der vierten Stelle der *Policy Effect* vermerkt sein muss. Der *Policy Effect* ist auf Deny-Override gesetzt, das heißt Anfragen sind grundsätzlich erlaubt, solange keine Verbotsregelung existiert. Der unterste Eintrag definiert den *Matcher* der Access-Policy. Hierbei sieht man wie die Anfrage mit der Policy verglichen werden soll. Dabei ist zu erkennen, dass eine Funktion ''keyMatch()'' genutzt wird. Diese Funktion wird von der Ausführungsumgebung *Casbin* (Kapitel 6.3) bereitgestellt. ''keyMatch()'' vergleicht zwei Werte auf Äquivalenz. Zusätzlich kann in den Werten das Zeichen ''*'' enthalten sein, welches in der Funktion eine Wildcard repräsentiert.

```
1  ## ACCESS POLICY ##
2
3  p, GPS_DATA_RABBITMQ, INSERT_DATA, MYSQL, allow
4  p, TEMP_DATA_REDIS, *, *, deny
```

Listing 6.4: CSV der Access-Policy

In Listing 6.4 ist die CSV der Access-Policy dargestellt. Die CSV enthält zwei Einträge. Diese sind wie in Listing 6.3 beschrieben, als vierstelliger Tupel definiert. Der erste Eintrag erlaubt die Verarbeitung aller Daten aus der *Datasource* ''GPS_DATA_RABBITMQ'' mit der *Funktionalität* ''INSERT_DATA'' in die *Datadestination* ''MYSQL''. Der zweiten Eintrag ist eine Verbotsregel, welche man am letzten Eintrag ''deny'' des Tupels erkennt. Für die *Funktionalität* und die *Datadestination* wurde hier eine Wildcard verwendet. Daraus resultiert, dass keine Verarbeitung mit den Daten aus der *Datasource* ''TEMP_DATA_REDIS'' erlaubt ist.

```
[request_definition]
r = dataobj

[policy_definition]
p = datasourceid, functionality, datadestinationid, attribute

[policy_effect]
e = some(where (p.eft == allow))

[matchers]
m = r.dataobj.id == p.datasourceid && r.dataobj.attr.attribute == p.attribute
```

Listing 6.5: Konfigurationsdatei einer Execution-Policy

In Listing 6.3 ist die Konfigurationsdatei einer Execution-Policy zu sehen. Der Syntax der Anfrage ist durch einen einstelligen Tupel beschrieben, welches ein Datenobjekt symbolisiert. Der Syntax der Policy wird als ein vierstelliges Tupel definiert, dass aus einer *Datasource* ID, *Funktionalität*, *Datadestination* ID und einem Attribut besteht. Der *Policy Effect* spielt in der Execution-Policy keine Rolle, muss jedoch aufgrund der Struktur des *PERM Metamodel* eingetragen sein. Die letzte Zeile beinhaltet die Beschreibung des *Matcher*, in welcher die Anfrage mit der Policy verglichen wird. Hierbei ist zu erkennen, dass es sich bei ''dataobj'' um ein komplexes Objekt mit eigenen Attributen handelt. Dieses repräsentiert einen *Datasource* JSON Objekt, dessen Aufbau in Kapitel 6.2.1 beschrieben ist.

```
## EXECUTION POLICY ##

p, GPS_DATA_RABBITMQ, INSERT_DATA, MYSQL, android
p, TEMP_DATA_REDIS, DELETE_DATA, MONGODB, 30
```

Listing 6.6: CSV einer Execution-Policy

Listing 6.6 zeigt die CSV einer Execution-Policy. Die CSV enthält zwei Einträge, die jeweils eine Regel der Policy darstellen. Diese sind wie in Listing 6.3 beschrieben, als vierstelliges Tupel definiert. Der erste Eintrag erlaubt, dass Daten aus der *Datasource* ''GPS_DATA_RABBITMQ'' an die *Datadestination* ''MYSQL'' weitergeleitet werden und die *Funktionalität* ''INSERT_DATA'' ausgeführt wird, wenn das Attribut ''android'' im Datensatz vorhanden ist. Grundsätzlich werden hier GPS Daten von Android Smartphones abgespeichert. Der zweite Eintrag ist analog zu betrachten. Hierbei handelt es sich um Temperatur Werte. Ist der Wert 30, so werden Daten in der MongoDB gelöscht.

6.2.3 Umsetzung der Adapter

Ein Adapter ist in der Umsetzung durch eine Klasse repräsentiert. Diese Klasse muss von der entsprechenden abstrakten Klasse erben und jeweils zwei Methoden implementieren. Im Folgenden werden zwei Codebeispiele beschrieben, um die Funktionsweise zu verdeutlichen.

```
1  @Component("GPS_DATA_RABBITMQ")
2  public class RabbitMQAdapter extends AbstractDataSourceAdapter implements DeliverCallback {
3
4    DataSource metaDataSource;
5    Channel channel;
6    String queuename;
7
8    @Override
9    public void initializeConnection(DataSource dataSource) {
10
11     DataLocation dataLocation = dataSource.getLocation();
12     this.metaDataSource = dataSource;
13     this.queuename = dataLocation.getPath();
14
15     ConnectionFactory factory = new ConnectionFactory();
16     factory.setHost(dataLocation.getHost());
17     factory.setPort(Integer.parseInt(dataLocation.getPort()));
18     factory.setUsername(dataLocation.getUser());
19     factory.setPassword(dataLocation.getPass());
20     factory.setVirtualHost(dataLocation.getEntrypoint());
21
22     this.channel = factory.newConnection().createChannel();
23     this.channel.queueDeclare(this.queuename, false, false, false, null);
24
25   }
26
27   @Override
28   public void startReceivingData(DataSource dataSource) throws IOException {
29     this.channel.basicConsume(this.queuename, true, this,
30         consumerTag -> {
31         }
32     );
33   }
34
35   @Override
36   public void handle(String consumerTag, Delivery delivery) throws IOException {
37
38     ...
39
40     this.applyPolicies(metaDataSourceCopy, payloadObject);
41
42   }
43 }
```

Listing 6.7: Implementierung eines Datasource-Adapters für RabbitMQ

Listing 6.7 zeigt die Klasse ''RabbitMQAdapter'', welche eine Anbindung an den RabbitMQ Broker darstellt. In Zeile 1 sieht man eine Spring Annotation, welche mit der *Datasource* ID gefüllt ist. Dadurch kann das Framework den Adapter zur richtigen *Datasource* zuordnen. In Zeile 2 wird die Klasse definiert und erweitert die abstrakte Klasse ''AbstractDataSourceAdapter''. Damit muss ''RabbitMQAdapter'' zwei Methoden implementieren. Die Methode ''initializeConnection'' (Zeile 9) übergibt der Klasse die passende *Datasource*. Der Adapter muss in dieser Methode die Informationen der Datasource nutzen, um eine Verbindung zum jeweiligen Datenspeicher aufbauen zu können. Die zweite zu implementierende Methode ''startReceivingData'' (Zeile 28) dient als Startzeichen für den Adapter. Die Methode wird vom Framework aufgerufen, wenn dieses bereit zur Verarbeitung der Daten ist. Dementsprechend bindet sich hier der Adapter an die entsprechende Queue. Zeile 36 zeigt eine Methode ''handle'' die vom Interface ''DeliverCallback'' festgesetzt wird. Das Interface ist eine Klasse aus dem RabbitMQ Client. Die Methode ''handle'' wird aufgerufen, sobald eine Message aus der Queue ankommt. Der Adapter hat die Möglichkeit die angekommenen Daten jetzt vorzubereiten und an das Framework abzugeben. Die Abgabe geschieht mit der vererbten Methode ''applyPolicies''.

```
1  @Component("MYSQLDB")
2  public class MySQLAdapter extends AbstractDataDestinationAdapter {
3
4    Connection conn;
5    String tablenametemp;
6    String tablenamegps;
7    String databasename;
8
9    @Override
10   public void initializeConnection(DataDestination dataDestination) throws Exception {
11
12     DataLocation dataLocation = dataDestination.getLocation();
13     String host = dataLocation.getHost();
14     String port = dataLocation.getPort();
15     String user = dataLocation.getUser();
16     String pass = dataLocation.getPass();
17     databasename = dataLocation.getEntrypoint();
18     tablenametemp = dataLocation.getPath();
19     tablenamegps = dataLocation.getAttr().get("tablegps");
20
21     String connectString = "jdbc:mysql://" + host + ":" + port + "/" + databasename
22         + "?" + "user=" + user + "&password=" + pass + "&autoReconnect=true";
23
24     conn = DriverManager.getConnection(connectString);
25
26   }
27
28   @Override
29   public Map<String, Function<ExecutionDescription, Boolean>>
30           initializeAndReturnFunctionalityMapping (DataDestination dataDestination) {
31
32     Map<String, Function<ExecutionDescription, Boolean>> mapping = new HashMap<>();
33     mapping.put("INSERT_GPS_DATA", this::insertDataGPS);
34     mapping.put("DELETE_GPS_DATA", this::deleteDataGPS);
```

```
35
36      return mapping;
37    }
38
39    private boolean insertDataGPS(ExecutionDescription executionDescription) { ... }
40
41    private boolean deleteDataGPS(ExecutionDescription executionDescription) { ... }
42
43  }
```

Listing 6.8: Implementierung eines Datadestination-Adapters für MySQL

Listing 6.8 zeigt die Klasse ''MySQLAdapter'', welche eine Anbindung an die MySQL-Datenbank darstellt. In Zeile 1 sieht man eine Spring Annotation, welche mit der *Datadestination* ID gefüllt ist. Dadurch kann das Framework den Adapter zur richtigen *Datadestination* zuordnen. In Zeile 2 wird die Klasse definiert und erweitert die abstrakte Klasse ''AbstractDataDestinationAdapter''. Dies zwingt den Adapter zwei Methoden zu implementieren. Zeile 10 zeigt die Methode ''initializeConnection''. Hierbei wird ein *Datadestination* Objekt übergeben, welchen der Adapter nutzt, um eine Verbindung zum Datenspeicher aufzubauen. Die Methode ''initializeAndReturnFunctionalityMapping'' (Zeile 29) wird vom Framework aufgerufen, um vom Adapter jeweils ein Mapping zwischen den angegebenen *Funktionalitäten* der *Datadestination* und den Methoden des Adapters zu erhalten. Damit kann das Framework die eingetragenen *Funktionalitäten* in der *Execution-Policy* zu dem richtigen Methodenaufruf zuordnen.

6.3 Casbin

Casbin ist eine Autorisierungsbibliothek für Zugangskontrolle [CAS18]. Diese wurde Open Source implementiert und liegt in vielen verschiedenen Programmiersprachen vor [GIT18]. *Casbin* ist eine Ausführungsumgebung für das *PERM Metamodel* (Kapitel 5.4). Im Bereich der Zugangskontrolle werden von *Casbin* drei grundlegende Mechanismen der Zugangskontrolle umgesetzt. *Access Control List* (ACL) [Shi07] , *Role-Based Access Control* (RBAC) [San98] und Attribute-Based Access Control (ABAC) [HKFV15]. Die Konfiguration von *Casbin* besteht allein aus der Nutzung zweier Dateien. Eine Modellbeschreibung (CONF), welches das Modell einer Policy abbildet und eine CSV Datei, welche die Regeln einer Policy enthält. Aus diesen beiden Dateien lässt sich die Kernkomponente *Enforcer* initialisieren. Anschließend kann der *Enforcer* genutzt werden, um Antworten auf Zugangsanfragen zu erteilen. *Casbin* wurde für die Umsetzung des Frameworks ausgesucht, da es Policies in einem einfach lesbaren Format nutzt. Das CSV Dateiformat ist ein altes [Sha05] und bereits etabliertes Dateiformat, welches von vielen Systemen eingelesen und verarbeitet werden kann. Weiterhin ist *Casbin* durch die Nutzung des *PERM Metamodels* sehr flexibel. Das Verhalten von Policies lässt sich durch das jeweilige Modell konfigurieren und bedarf somit keiner Code Anpassung von *Casbin* selbst.

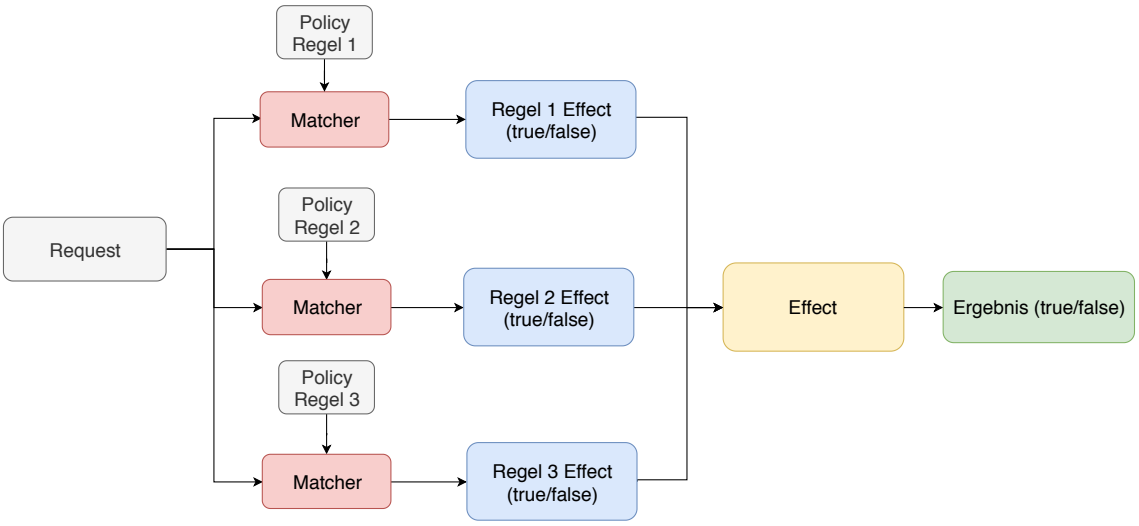


Abbildung 6.5: Ablaufmodel des Enforcers

6.3.1 Enforcer

Der *Enforcer* ist eine zentrale Komponente in *Casbin*. Der Enforcer stellt die ausführbare Umgebung für das *PERM Metamodel* dar. Das heißt, nach dem der *Enforcer* durch das Modell und die Policy initialisiert wurde, kann dieser auf eine Zugangsanfrage einen booleschen Wert *true* oder *false* zurückgeben. Dementsprechend kann der *Enforcer* relativ simpel in bedingte Anweisungen eingebaut werden.

Für die Auswertung nutzt der *Enforcer* den *Matcher*-Eintrag (Kapitel 5.4) aus dem Modell der *Access-Policy*. Anschließend werden die Daten der Anfrage mit den Einträgen in der Policy durch den logischen Ausdruck aus dem Matcher abgeglichen. Der logische Ausdruck des Matchers liefert einen booleschen Wert. Dieser sagt aus, ob die Anfrage mit dem jeweiligen Eintrag der *Access-Policy* übereinstimmt. Finden keine oder mehrere Übereinstimmungen statt, so wird der Policy Effect (Kapitel 5.4) betrachtet, welcher dann einen booleschen Wert festlegt.

In Abbildung 6.5 ist das Ablaufmodel des Enforcers dargestellt. In dieser Abbildung enthält die Policy drei Regeln bzw. Einträge. Links ist die *Request* zu sehen, welche bei Ankunft mit jeder Regel der Policy durch den *Matcher* verglichen wird. Jeder Vergleich führt zu einem Ergebnis *true* oder *false*. Die Menge dieser Vergleiche wird am *Effect* zu einem eindeutigen Ergebnis gewandelt.

6.3.2 ExecutionEnforcer

Beim *ExecutionEnforcer* handelt es sich nicht um eine Komponente von *Casbin*, sondern um eine spezifische Komponente der Umsetzung vom Framework. Die Funktionsweise vom *ExecutionEnforcer* ist fast identisch mit dem des *Enforcers*. Der *ExecutionEnforcer* ist speziell für die *Execution Policies* zuständig. Das Problem am *Enforcer* ist, dass dieser zu Anfragen nur ein *true* oder *false* zurückgeben kann. Da in einer *Execution-Policy*

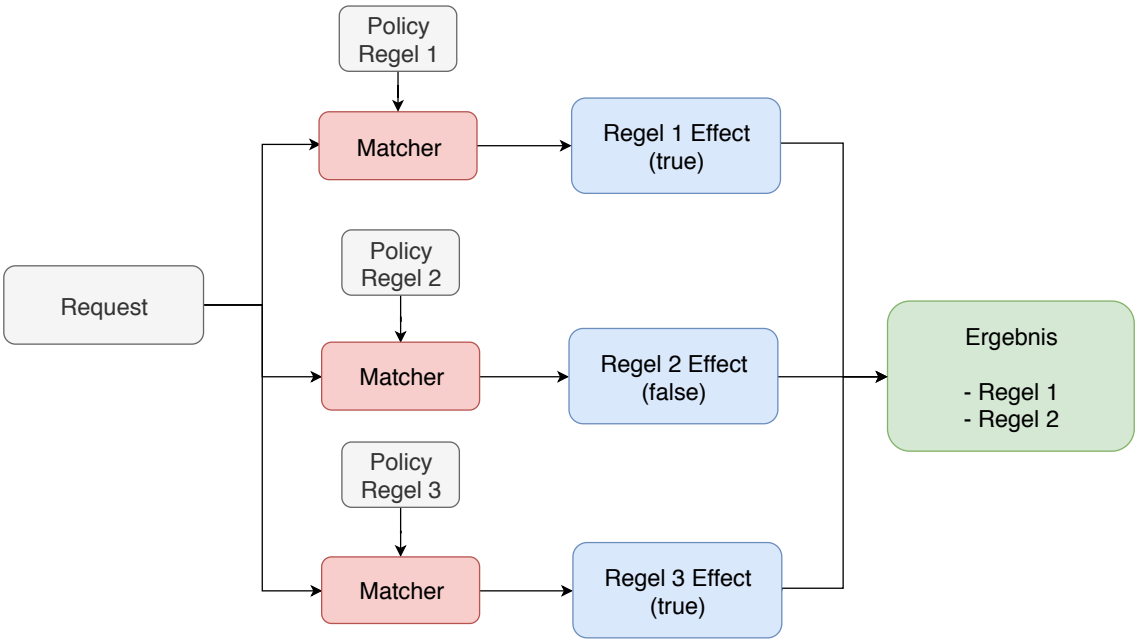


Abbildung 6.6: Ablaufmodel des ExecutionEnforcers

mehrere Einträge und somit mehrere Ausführungen für einen Datensatz vorliegen können, reicht ein boolescher Ausgabewert nicht mehr aus. Vielmehr braucht es eine Liste von Ausführungen als Ausgabe. Diese Information lässt sich am besten durch eine Ausgabe der übereinstimmenden Einträge aus der Policy erhalten. Dafür wurde zunächst die Evaluation über den Policy Effect entfernt. Alle mit dem Matcher übereinstimmenden Einträge werden in einer Liste gesammelt und am Ende ausgegeben.

Die Abbildung 6.6 zeigt das Ablaufmodel des ExecutionEnforcers. In dieser Abbildung enthält die Policy drei Regeln bzw. Einträge. Links ist die *Request* zu sehen, welche bei Ankunft an jeder Regel der Policy mithilfe des *Matchers* verglichen wird. Jeder Vergleich führt zu einem Ergebnis *true* oder *false*. In der Abbildung sind bereits Ergebnisse eingetragen. Alle Vergleiche die das Ergebnis *true* liefern, werden in das Ergebnis aufgenommen. Das Ergebnis besteht aus der Rückgabe der Regeln selbst.

6.4 Datenflüsse

Abbildung 6.7 zeigt einen möglichen Datenfluss in der Umsetzung bei einer Anbindung zwischen einem RabbitMQ Broker und einer MongoDB Datenbank. Blaue Komponenten der Abbildung stellen Drittsysteme dar, an die das Framework angebunden ist. Grüne Komponenten sind Bestandteile des Frameworks. Graue Komponenten zeigen die Stationen an, die Daten durchlaufen müssen. Oben links im Bild ist der RabbitMQ Broker als blaue Komponente dargestellt. Diesem liegen Daten vor, welche der Broker z.B durch einen Sensor erhält. Die Daten werden von einem passenden *Datasource-Adapter* empfangen. Anschließend werden die Daten vorbereitet. Hierbei kann z.B. eine Transformation stattfinden oder zusätzliche Metadaten für die Policy Evaluation injiziert werden. Anschließend werden

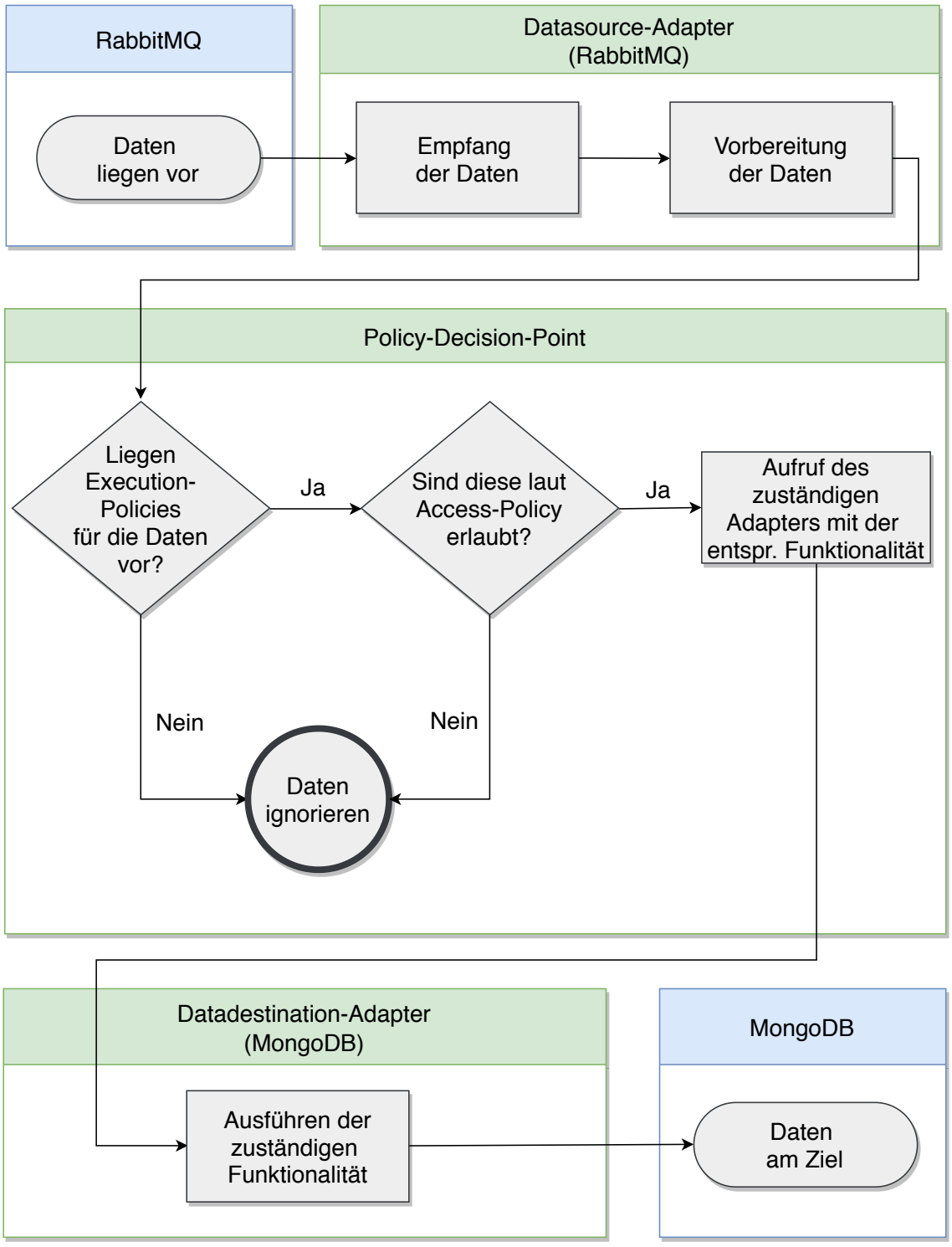


Abbildung 6.7: Datenfluss bei einer Anbindung zwischen RabbitMQ und MongoDB

die Daten an den *Policy-Decision-Point* weitergereicht. Dieser prüft zunächst mit dem *Execution Enforcer*, ob Execution Policies für diesen Datensatz existieren. Gibt es keine zutreffende Regelung, wird der Datensatz ignoriert und verworfen. Sind jedoch Regelungen vorhanden, werden diese an der *Access-Policy* evaluiert. Ist die Ausführung untersagt, wird der Datensatz verworfen. Bei einer Freigabe werden die Execution Policies genutzt, um die entsprechenden *Datadestination-Adapter* mit den vorgesehenen Funktionalitäten aufzurufen. Damit werden die Daten an den jeweiligen *Datadestination-Adapter* übergeben. In der Ausführung der Funktionalität kann z.B. eine entsprechende Transformation der Daten stattfinden und die Daten für die MongoDB vorbereitet werden. Nach der Übergabe der Daten an die MongoDB endet der Datenfluss für das Framework.

6.5 Deployment Lösung - Docker

Die Initiale Umsetzung des Frameworks speichert ihren Zustand auf der Festplatte des Host-Systems ab. Da der Zustand sich nicht in einer Datenbank befindet, muss sichergestellt werden, dass beim Deployment der Umsetzung der Zustand korrekt übertragen wird. In diesem Fall eignet sich die Nutzung eines Docker Containers.

Docker ist eine Containervirtualisierung, die es dem Nutzer erlaubt eigenständige Container zu beschreiben, welche dann isoliert voneinander auf einem Host System ausgeführt werden können. Die Beschreibung der Container wird einer ''Dockerfile'' definiert. Dies ermöglicht den Zustand der Umsetzung während der Bauphase im Container abzulegen. Weiterhin lässt sich im Container die Ausführungsumgebung der Umsetzung installieren, sodass diese auf dem Host System nicht mehr vorhanden sein muss. Ein fertiger Container kann auf jedem Host System deployed werden, welches Docker installiert hat.

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 COPY /build/resources/main/datamodel/ /tmp/datamodel
4 COPY /build/resources/main/policies/ /tmp/policies
5 COPY /build/libs/compositedatastore-1.0.0.RELEASE.jar /app.jar
6 EXPOSE 8080
7 ENTRYPOINT ["java","-Dspring.profiles.active=docker","-jar","/app.jar"]
```

Listing 6.9: Dockerfile der Applikation

Listing 6.9 zeigt die verwendete Dockerfile für die Umsetzung des Frameworks. In der ersten Zeile wird mit dem Befehl **FROM** das Basis Image ausgewählt. In diesem Fall handelt es sich um ein Linux Image mit vorinstallierter Java-Ausführungsumgebung. Anschließend wird mit **VOLUME** eine Partition für das Datenmodell und die Policies erstellt. Alle folgenden **COPY** Befehle kopieren das Datenmodell, die Policies und die fertig gebaute Applikation in das Image. Durch **EXPOSE** wird der Port geöffnet, auf welchem später die Applikation laufen wird. In der letzten Zeile werden mit **ENTRYPOINT** Konsolenbefehle beschrieben, welche zum Start der Applikation benötigt werden.

6.5.1 Docker Volume

Wird ein Docker Container gelöscht, gehen alle Daten im Container verloren. Durch Anbindung von Docker Volumes können Daten aus Containern auch nach der Löschung des Containers erhalten bleiben. Dies geschieht durch eine explizite Notation einer Partition in der Dockerfile, mithilfe des Schlüsselwortes **VOLUME**. Alle Daten, die der Container in die angegeben Partition schreibt, bleiben unabhängig vom Zustand des Containers erhalten. Docker Volumes lassen sich exportieren und von andern Containern importieren. Dadurch können Daten an neuere Versionen eines Containers weitergegeben werden. In der Umsetzung wurde Docker Volume genutzt, um alle Datenmodelle und Policies separat ablegen zu können.

6.5.2 Docker Container Export

Üblicherweise werden Docker Images zwischen verschiedenen Host Systemen über eine Docker Registry geteilt. Für das Deployment der Umsetzung des Frameworks wurde keine Docker Registry genutzt, da das Aufsetzen einer privaten Registry nur für das Projekt einen zu hohen Aufwand im Vergleich zum Nutzen gewesen wäre. Stattdessen wurden die fertigen Container als tar Datei exportiert, um diese zwischen den Host Systemen zu teilen. Die Export-Funktion ist in Docker nativ vorhanden und kann durch den Konsolenbefehl ''docker save '' ausgeführt werden. Durch den Befehl ''docker load'' kann die entsprechende tar Datei als Container importiert werden.

7 Evaluation

Um die Idee des Frameworks zu validieren, wurde die Umsetzung auf der Open-Stack-Umgebung vom IPVS der Universität Stuttgart bereitgestellt. Hierbei läuft das Framework in einer Umgebung, wo es mit unterschiedlichen Datenspeichern verschiedenen Typs kommunizieren muss.

7.1 Web UI

Für die Modifikation des Daten- und Policymodells während der Laufzeit wurde eine Web UI auf Basis von HTML, CSS und JavaScript erstellt. Die Web UI erlaubt grundlegende CRUD-Möglichkeiten um die Modelle zu modifizieren. Dafür wird eine REST-Schnittstelle am Framework genutzt, welche die Modelle direkt auf dem Docker-Volume bearbeiten kann. Jedoch können die Änderung nicht zur Laufzeit übernommen werden. Das heißt eine geänderte Policy wird zwar gespeichert, jedoch nicht sofort in den Speicher geladen und somit nicht für aktuelle Datenflüsse übernommen. Nach einem Neustart des Frameworks können die Änderung von dem Docker-Volume in den Speicher eingelesen werden.

7.2 Aufbau der Umgebung

Abbildung 7.1 zeigt den Aufbau der Umgebung. Für die Umgebung wurden zwei Virtuelle Maschinen (VM) des Typs *m1.medium* bereitgestellt. Die VMs besitzen jeweils 2 Virtuelle CPU Kerne, 4GB RAM und 40GB Festplattenspeicher. Beide VMs nutzen das Betriebssystem Ubuntu in der Version 16.04 LTS. Weiterhin wurde Docker als Container-Umgebung gewählt, um möglichst unterschiedliche Systeme einfach einzubinden. Auf der ersten VM mit dem Instanznamen *MA_CDS_MAIN* sind zwei Docker-Container bereitgestellt. In dem ersten befindet sich das Framework selbst. In dem zweiten ein Datengenerator welcher GPS und Temperatursensoren emuliert. Die zweite VM mit dem Instanznamen *MA_CDS_ENV* enthält vier Systeme die jeweils in einem Docker-Container ausgeführt werden. Die folgenden Systeme wurden genutzt: Die Key-Value Datenbank *Redis*, der Messaging Broker *RabbitMQ*, die relationale Datenbank *MySQL* und die Dokumentdatenbank *MongoDB*.

7.3 Ablauf der Umgebung

Der Ablauf imitiert grundlegend das gewählte Beispiel aus Kapitel 5 in der Abbildung 5.3. Das Beispiel wurde jedoch so angepasst, dass es ressourcensparend in einer Dauerschleife laufen kann.

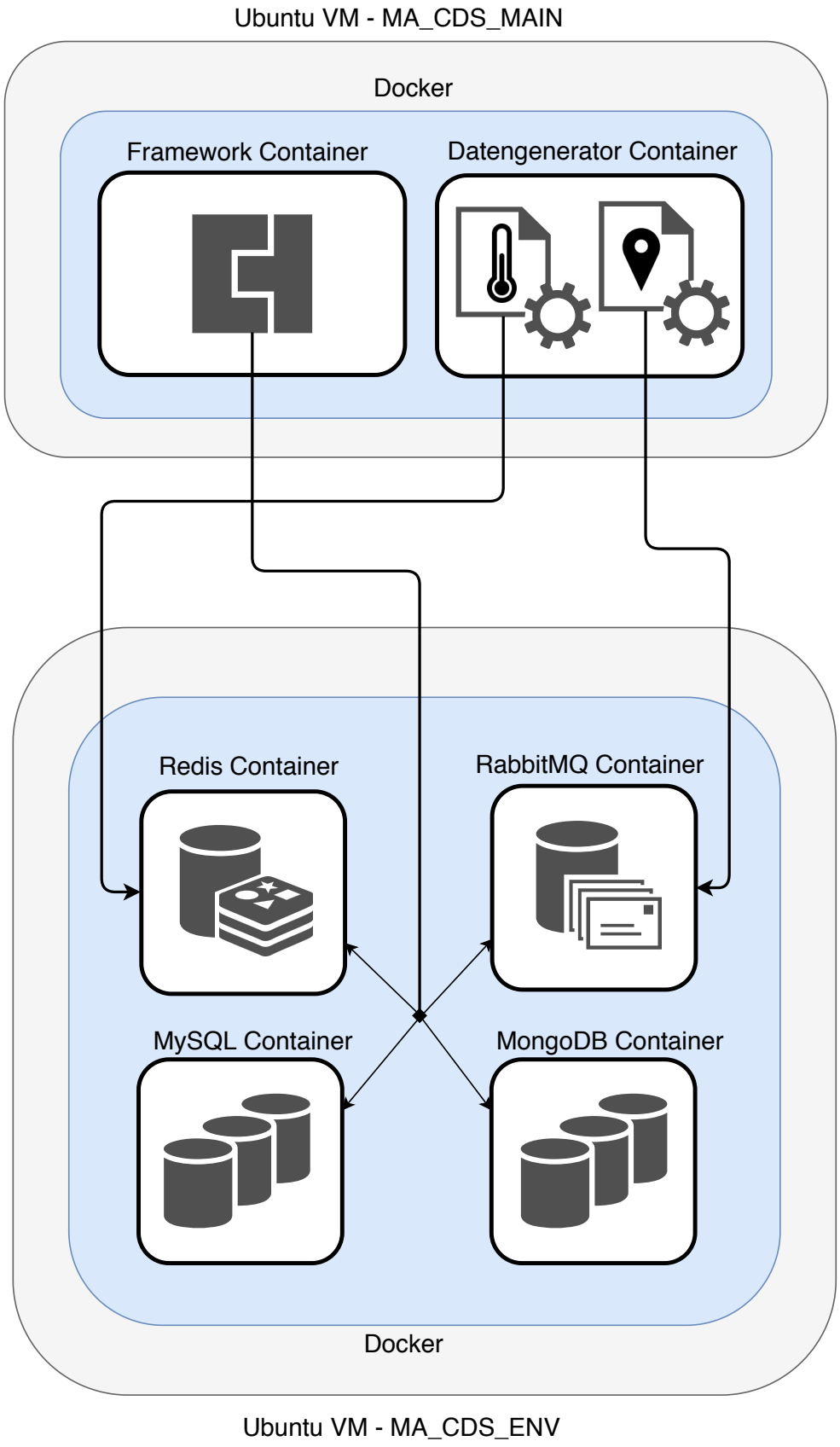


Abbildung 7.1: Aufbau der Umgebung in Open-Stack

Startpunkt des Datenflusses ist der Datengenerator, welcher auf der VM *MA_CDS_MAIN* ausgeführt wird. Dieser generiert randomisierte Temperaturdaten, als auch Geodaten eines Smartphones. Die Temperaturdaten werden in *Redis* abgelegt, während die Geodaten in eine Queue von *RabbitMQ* übertragen werden. Das Framework ist durch zwei *Datasource-Adapter* an *Redis* und *RabbitMQ* angebunden. Die *Execution Policies* sind so konfiguriert, dass die Temperaturdaten aus *Redis* bei einem Wert zwischen 40 bis 65 in die MySQL-Datenbank abgelegt werden. Liegt der Wert zwischen 66 und 90, so werden die Daten in die MongoDB abgespeichert. Die Geodaten werden in die *MongoDB* synchronisiert, wenn diese von einem Android Smartphone stammen. Sind diese von einem iPhone, so kommen diese in die *MySQL*-Datenbank. Damit die Datenspeicher nicht an ihre Kapazitätsgrenzen kommen wurde noch ein Löschmechanismus eingebaut. Dazu wurden zwei weitere Datasource-Adapter angelegt, die sich mit der *MySQL*-Datenbank und *MongoDB* verbinden. Diese prüfen die Mengen der Dateneinträge in den beiden Datenbanken und senden diese an das Framework. In einer weiteren *Execution-Policy* wird hinterlegt, wenn die Anzahl der Einträge über 100 ist, sollen die Datenbanken geleert werden. Dadurch können die Speicher nie volllaufen und der Datenfluss kann in einer Endlosschleife laufen.

7.4 Verifikation der Anforderungen

In diesem Abschnitt wird beschrieben, wie das Framework die Anforderungen aus Kapitel 4 angeht und erfüllt:

- **Verwaltung von verteilten Daten** - Das Framework erlaubt es mit seinem Datenmodell verschiedene Datenspeicher zu definieren. Dadurch lassen sich Datenspeicher an beliebigen Orten im Framework registrieren. In der Open-Stack-Umgebung konnten die Daten an vier unterschiedlicher Systeme angebunden und untereinander verwaltet werden. Aktionen auf Daten konnte sich mit dem Datenmodell *Datadestination* umsetzen. Durch die *Funktionalitäten* in *Datadestination* kann jede beliebige Aktion für die Daten angegeben werden. Jede einzelne CRUD-Operation kann als eine *Funktionalität* abgebildet werden. Konfigurationen und Verbindungen zwischen den Datenspeichern und deren Daten konnten unabhängig durch das *Policy-Modell* definiert werden.
- **Persistente und dynamische Daten** - Die Komponente *Adapter* im Framework sorgt für die richtige Kommunikation zwischen Datenmodell und Datenspeicher. In der Open-Stack-Umgebung wurden durch MySQL und MongoDB auf persistente Daten zugegriffen. Durch die Anbindung an RabbitMQ konnten asynchron Daten empfangen werden. Weiterhin mussten in Redis Daten rechtzeitig abgeholt werden, bevor diese überschrieben wurden. Die *Adapter*-Komponente kann persistente Daten aus Datenspeichern auslesen, als auf dynamische Daten reagieren und zum richtigen Zeitpunkt empfangen.
- **Konfiguration durch Policies** - Das Framework bietet ein *Policy-Modell*, welches zwei Arten von Policies definiert. Mit *Execution Policies* lassen sich Verwaltungen der Daten abbilden und mit der *Access-Policy* die Zugangsberechtigung zwischen den

Datenspeichern regeln. Durch das *PERM-Metamodell* wurde eine Modellierungsmöglichkeit für die Policies definiert, welche in textbasierten Dateiformaten umsetzbar ist. In der Open-Stack-Umgebung konnten mehrere Policies definiert werden, die für das Verhalten der Daten zuständig waren.

- **Erweiterbarkeit** - Das Datenmodellkonzept im Framework ist auf kein spezifisches Datenspeicherkonzept festgelegt. Damit lassen sich zukünftige Speichertechnologien im Framework umsetzen, indem man einen neuen *Datalocation* hinzufügt. Weiterhin wird durch den *Adapter* eine Möglichkeit geboten, neue Logik oder Protokolle umzusetzen als auch nachträglich abzuändern. Policies und ihre Funktionsweise können durch das *PERM-Metamodell* modifiziert und angepasst werden. Für das Szenario in der Open-Stack-Umgebung mussten verschiedene Policy-Modelle erstellt werden, um das Szenario zu realisieren.

7.5 Verbesserungsmöglichkeiten

Bei der Evaluation des Frameworks sind einige Verbesserungsmöglichkeiten aufgefallen, wie das Framework in Zukunft erweitern werden kann. Hierbei gliedern sich die Verbesserungsmöglichkeiten in zwei Kategorien. Es gibt Verbesserungsmöglichkeiten am grundlegendem Konzept des Frameworks, sowie an der Umsetzung des Frameworks.

7.5.1 Verbesserungsmöglichkeiten am Konzept des Frameworks

Das Framework ist so konzipiert, dass alle Daten die vom Framework verwaltet werden sollen, durch eine lauffähige Instanz des *Policy-Decision-Points* durchlaufen müssen. Dies kann im Zusammenhang der vielen verteilten Datenspeicher ein Bottleneck verursachen. Die gleiche Problematik ergibt sich auch, wenn riesige Datenmengen verwaltet werden. Weiterhin könnte es ein Problem für den Datenschutz darstellen, falls das Framework in Form eines Software as a Service (SaaS) angeboten werden soll. Unter Umständen will man nicht, dass die Daten das Framework durchlaufen. Eine Herangehensweise an dieses Problem wäre ein erweitertes Adapterkonzept, welches auf dem jeweiligen Datenspeicher läuft. Damit würde auf jedem Datenspeicher lokal ein Teil des Frameworks vorhanden sein. Durch Austausch von reinen Metainformationen mit dem *Policy-Decision-Point* können die nötigen Verwaltungsbefehle an die zuständigen Adapter weitergeleitet werden. So kann z.B. eine Verschiebung von Daten direkt zwischen zwei Datenspeichern stattfinden und diese müssen nicht durch das Framework geschoben werden.

Bei der Nutzung des Frameworks fokussiert man sich vor allem auf die *Execution Policies* und die *Adapter*. Besonders bei der Anbindung von neuen Datenspeichern wird am meisten an diesen zwei Komponenten gearbeitet. Dadurch entfällt die *Access-Policy* aus dem Fokus. Experimentelle Datenpfade werden intuitiv für Testzwecke zunächst in der Execution-Policy entfernt und nicht durch die *Access-Policy* verboten. Die *Access-Policy* als eine optionale Komponente zu deklarieren würde den Aufwand beim Aufsetzen von Szenarien vereinfachen und gleichzeitig die Funktionalität nicht aus dem Framework entfernen.

7.5.2 Verbesserungsmöglichkeiten an der Umsetzung

Jede *Execution-Policy* beinhaltet im *Matcher* einen Abgleich der *Datasource* ID mit der ID in der Policy. Damit wird sichergestellt, dass Anfragen mit den dafür bestimmten Regeln der *Execution-Policy* übereinstimmen. Ein Eintrag in der *Execution-Policy* mit einer *Datasource* ID 'X' ist nicht für eine *Datasource* mit der ID 'Y' bestimmt. Dieser Abgleich im *Matcher* ist nicht nur trivial, sondern führt auch zu einer Zugriffszeit von $O(n)$, wobei n die Anzahl aller Regeln ist. Dies lässt sich durch Vorberechnung in eine Map-Datenstruktur optimieren. Dazu kann man nach dem Laden der *Execution-Policy* die Policies in eine Map sortieren, deren Keys *Datasource* IDs entsprechen. Die Werte dieser Einträge wären dann alle dazugehörigen Execution-Policies zu der zugehörigen *Datasource* ID. Damit wäre eine Zugriffszeit von $O(1)$ pro Anfrage erreicht.

Die Umsetzung liest das Datenmodell, sowie die Policies zum Startzeitpunkt der Applikation ein. Ändert man das Datenmodell oder die Policies über die UI, werden diese nicht zur Laufzeit übernommen. Um die Änderungen wirksam zu machen muss die Applikation neu gestartet werden. Dies kann unter Umständen zu Datenverlust führen. Eine Möglichkeiten Änderungen zur Laufzeit zu übernehmen wäre von Vorteil. Ein Ansatz dafür kann z.B. der Spring Cloud Restart Endpoint[1] sein.

Beim Anbinden der Adapter an die jeweiligen Datenspeicher müssen die Zugangsinformationen aus dem Datenmodell mit dem jeweiligen Client des Datenspeichers integriert werden. Eine Erstellung von vorgefertigten Adaptern für populäre Datenspeicher kann diesem Aufwand entgegenwirken.

[1] https://www.baeldung.com/java-restart-spring-boot-app

8 Zusammenfassung und Ausblick

Ziel dieser Ausarbeitung war die Erstellung und Umsetzung eines Frameworks für einen zusammengesetzten Datenspeicher. Nach der Darlegung der Aufgabenstellung folgten verwandte Arbeiten im Bereich Datenframeworks, sowie Regelwerke für Datenflüsse. Hierbei wurde klar, dass einige Datenframeworks im Bereich Edge und Cloud Computing existieren, jedoch keines sich explizit mit der Steuerung und dem Management von Daten zwischen unterschiedlichen Datenspeichern beschäftigt. Dazu angrenzend gab es keine Daten-Policy-Modelle zur Lenkung von Datenflüssen. Anschließend wurden Grundlagen in Anwendungsbereichen des Frameworks gefestigt, sowie Grundkonzepte eines Frameworks erläutert. Nach Festlegung der Anforderungen konnte mit der Entwicklung des Frameworks begonnen werden. Durch die Erstellung eines Datenmodells für unterschiedliche Datenspeicher, die Nutzung eines Adapterkonzepts, sowie die Modellierung von Datenfluss-Policies unter Nutzung eines Metamodells konnte ein Daten-Framework entwickelt werden, welches alle festgelegten Anforderung erfüllt. Für die Evaluation wurde das Framework in Java umgesetzt und in einem Szenario auf der Open-Stack-Umgebung des IPVS der Universität Stuttgart bereitgestellt. Dabei entstanden Verbesserungsvorschläge für die Erweiterung des Framework-Konzeptes, sowie für die Implementierung.

Ausblick

Datenbewegung bzw. Datenaustausch zwischen unterschiedlichen Systemen ist ein Grundkonzept der Informationstechnologie. Die Anbindung zweier Systeme bedarf meist immer einer individuellen Anpassung. Protokoll-, Daten-, sowie Schnittstellenstandards helfen dabei Systeme mit immer weniger Aufwand zu verbinden. Mit dem Framework dieser Masterarbeit wurde ein Schritt zur Vereinfachung der Datenanbindung unterschiedlicher Systeme erfüllt. Ein Weiterentwicklungsaspekt wäre die Entwicklung von vorgefertigten Adaptern, welche bereits etablierte Systemprotokolle, wie MQTT, AMQP oder HTTP unterstützen. Alternativ könnte die Erstellung der Adapter durch maschinelles Lernen umgesetzt werden. Mit der Hilfe von Beispieldaten könnte ein passender Adapter für das jeweilige System generiert werden. Dadurch wäre der Aufwand für Systemanbindungen weiter minimiert. Mit der immer größer werdenden Bedeutung von Big Data, kann das Adapter-System des Frameworks erweitert werden. Ein Client kann direkt auf dem jeweiligen System laufen und eine Datenübertragung zu anderen Clients auf Remote-Anweisung des Frameworks tätigen. Damit würde man den kürzesten Weg für die Übertragung wählen und das Framework als Bottleneck ausschließen.

Literaturverzeichnis

[ANHL14] F. Anon, V. Navarathinarasah, M. Hoang, C.-H. Lung. „Building a framework for internet of things and cloud computing“. In: *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*. IEEE. 2014, S. 132–139 (zitiert auf S. 12).

[BDPP16] A. Botta, W. De Donato, V. Persico, A. Pescapé. „Integration of cloud computing and internet of things: a survey“. In: *Future generation computer systems* 56 (2016), S. 684–700 (zitiert auf S. 9).

[BJJ10] S. Bhardwaj, L. Jain, S. Jain. „Cloud computing: A study of infrastructure as a service (IAAS)“. In: *International Journal of engineering and information Technology* 2.1 (2010), S. 60–63 (zitiert auf S. 16).

[BWHT12] P. Barnaghi, W. Wang, C. Henson, K. Taylor. „Semantics for the Internet of Things: early progress and back to the future“. In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 8.1 (2012), S. 1–21 (zitiert auf S. 9).

[CAS18] Yang Luo. *Casbin - An authorization library*. 2018. URL: https://casbin.org/en/ (zitiert auf S. 44).

[Cho07] V. Choudhary. „Software as a service: Implications for investment in software development“. In: *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE. 2007, 209a–209a (zitiert auf S. 16).

[CSGK16] F. Caglar, S. Shekhar, A. Gokhale, X. Koutsoukos. „Intelligent, performance interference-aware resource management for iot cloud backends“. In: *Internet-of-Things Design and Implementation (IoTDI), 2016 IEEE First International Conference on*. IEEE. 2016, S. 95–105 (zitiert auf S. 9).

[DEP04] Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. URL: https://martinfowler.com/articles/injection.html (zitiert auf S. 16).

[Fas16] A. Fasihi. „Rule based inference and action selection based on monitoring data in IoT“. Magisterarb. 2016 (zitiert auf S. 13).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. DOI: 10.1007/978-3-7091-1568-8 (zitiert auf S. 16).

[FRLB18] t3n. *Library vs. Framework: Das sind die Unterschiede*. 2018. URL: https://t3n.de/news/library-vs-framework-unterschiede-1022753/ (zitiert auf S. 17).

[GIT18] Yang Luo. *GitHub: Casbin - Casbin authorization library and the official middlewares*. 2018. URL: https://github.com/casbin (zitiert auf S. 27, 44).

[HEX05] Dr. Alistair Cockburnn. *Hexagonal Architecture - Ports and Adapter Pattern*. 2005. URL: http://web.archive.org/web/20140329201018/http://alistair.cockburn.us/Hexagonal+architecture (zitiert auf S. 34).

[HKFV15] V. C. Hu, D. R. Kuhn, D. F. Ferraiolo, J. Voas. „Attribute-based access control“. In: *Computer* 48.2 (2015), S. 85–88 (zitiert auf S. 44).

[INV05] Martin Fowler. *Inversion of Control*. 2005. URL: https://www.martinfowler.com/bliki/InversionOfControl.html (zitiert auf S. 17).

[JDC+14] L. Jiang, L. Da Xu, H. Cai, Z. Jiang, F. Bu, B. Xu. „An IoT-oriented data storage framework in cloud computing platform“. In: *IEEE Transactions on Industrial Informatics* 10.2 (2014), S. 1443–1451 (zitiert auf S. 11).

[Law08] G. Lawton. „Developing software online with platform-as-a-service technology“. In: *Computer* 41.6 (2008), S. 13–15 (zitiert auf S. 16).

[MG+11] P. Mell, T. Grance et al. „The NIST definition of cloud computing“. In: (2011) (zitiert auf S. 15).

[Obj16] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification*. 2016. URL: https://www.omg.org/spec/MOF/2.5.1/PDF (zitiert auf S. 17).

[Obj18] Object Management Group. *OMG'S METAOBJECT FACILITY*. 2018. URL: https://www.omg.org/mof/ (zitiert auf S. 17).

[PERM18] Yang Luo. *PERM Meta-Model*. 2018. URL: https://vicarie.in/posts/generalized-authz.html (zitiert auf S. 27).

[PM17] J. Pan, J. McElhannon. „Future edge cloud and edge computing for internet of things applications“. In: *IEEE Internet of Things Journal* 5.1 (2017), S. 439–449 (zitiert auf S. 9).

[Poe06] I. Poernomo. „The meta-object facility typed“. In: *Proceedings of the 2006 ACM symposium on Applied computing*. ACM. 2006, S. 1845–1849 (zitiert auf S. 17).

[Pra09] D. R. Prasanna. *Dependency injection*. Manning, 2009 (zitiert auf S. 16).

[Ros17] M. Rost. „Organisationen grundrechtskonform mit dem Standard-Datenschutzmodell gestalten“. In: *IT-Prüfung, Sicherheitsaudit und Datenschutzmodell*. Springer, 2017, S. 23–56 (zitiert auf S. 9).

[RSBS13] T. Rattanasawad, K. R. Saikaew, M. Buranarach, T. Supnithi. „A review and comparison of rule languages and rule-based inference engines for the Semantic Web“. In: *2013 International Computer Science and Engineering Conference (ICSEC)*. IEEE. 2013, S. 1–6 (zitiert auf S. 14).

[San98] R. S. Sandhu. „Role-based access control“. In: *Advances in computers*. Bd. 46. Elsevier, 1998, S. 237–286 (zitiert auf S. 44).

[Sat17] M. Satyanarayanan. „The emergence of edge computing“. In: *Computer* 50.1 (2017), S. 30–39 (zitiert auf S. 15).

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu. „Edge computing: Vision and challenges“. In: *IEEE Internet of Things Journal* 3.5 (2016), S. 637–646 (zitiert auf S. 15).

[SD16] W. Shi, S. Dustdar. „The promise of edge computing“. In: *Computer* 49.5 (2016), S. 78–81 (zitiert auf S. 15).

[Sha05] Y. Shafranovich. „Common format and MIME type for comma-separated values (CSV) files“. In: (2005) (zitiert auf S. 44).

[Shi07] R. Shirey. *RFC 4949-Internet Security Glossary*. 2007 (zitiert auf S. 44).

[SS94] R. S. Sandhu, P. Samarati. „Access control: principle and practice“. In: *IEEE communications magazine* 32.9 (1994), S. 40–48 (zitiert auf S. 26).

[SYN18] Yang Luo. *GitHub: Casbin -Syntax for Models*. 2018. URL: https://casbin.org/docs/en/syntax-for-models (zitiert auf S. 29).

[TSM+13] B. Trninić, G. Sladić, G. Milosavljević, B. Milosavljević, Z. Konjović. „Policydsl: Towards generic access control management based on a policy metamodel“. In: *2013 IEEE 12th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT)*. IEEE. 2013, S. 217–223 (zitiert auf S. 13).

Alle URLs wurden zuletzt am 31. 05. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift