Policy-basiertes Zugriffs-Management für Webservices unter Node.js mit Bezug auf ODRL und XACML

Bachelorarbeit

Abgabe: 28. Mai 2019

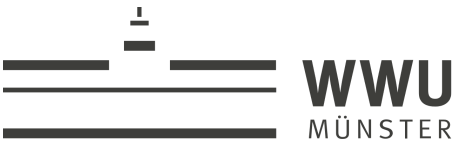
Autor: Simon Petrac

Matrikelnummer: 383 509

E-Mail-Adresse: s_petr02@uni-muenster.de

Studiengang: B.Sc. Informatik (Nebenfach Mathematik)

13. Fachsemester



Inhaltsverzeichnis

1. Motivation und Zielsetzung

Vor einem Jahr ist ein Freund mit einem spannenden Projekt an mich herangetreten. Der Traditionsbetrieb[1] der Familie hatte Probleme mit dem Verwaltungsaufwand unterschiedlicher Standorte, steigender Mitarbeiterzahlen und produzierter Waren, da bisher kaum digitale Verwaltung in dem Betrieb verwendet wurde. Die Mittel für eine professionelle ERP Software lagen außerhalb der Möglichkeiten, deshalb entschieden wir uns einen Webservice zu hosten und eine Applikation zu schreiben, welche die neu erworbenen Warenscanner nutzt um Details über die Betriebsabläufe aufzuschlüsseln, sowie Optionen zur Verwaltung bietet. Wir stellten fest, dass die am System beteiligten Entitäten (Scanner, Mitarbeiter, privilegierte Nutzer) unterschiedliche Aufgaben hatten und die erlaubten Aktionen aus Sicherheitsgründen durch eine Autorisierung eingeschränkt werden mussten, damit zum Beispiel die Scanner nicht missbraucht werden konnten, um sich auf der Weboberfläche einzuloggen.

Ich lernte die Open Digital Rights Language kennen und erkannte in der Syntax eine Möglichkeit, die Entitäten und Ressourcen des Servers abzubilden und mithilfe von ODRL-Dateien die Nutzungsrechte zu definieren, auch wenn mir der volle Umfang der ODRL zu dem Zeitpunkt noch nicht klar war. Schnell merkte ich jedoch, dass die autorisierte Ausführung von Aktionen, wie dem Einloggen am Dashboard oder dem Auswerten der Scannerdaten, keine leichte Aufgabe ist. Hinzu kommt, dass es sich nicht um statische Abläufe handelt, sondern Daten hinzukommen, gelöscht oder geändert werden. Auch wenn ich die Applikation letztendlich zum Laufen bekommen habe, war ich nicht überzeugt von meinem Quellcode. Ein fehlendes Grundkonzept hatte dazu geführt, dass der Code zunehmend unübersichtlicher wurde und immer mehr Sonderfälle betrachtet werden mussten, was die Wartung und Erweiterung des Codes schwierig gestaltete.

Ich konnte also noch nicht mit dem Thema abschließen. Andere Projekte warteten außerdem schon, welche genau dieselben Schwierigkeiten aufwiesen.

[1] Details wurden weggelassen oder abgeändert, um keine Verbindung zum tatsächlichen Betrieb herstellen zu können.

Die einzige, wirklich gute Lösung bedeutete für mich einen Policy-Agenten, welcher nicht nur die Autorisierung ausspricht, sondern die Entitäten am System verwaltet und Aktionen voll integriert und einheitlich ausführen kann. Ob es darum geht ein Dokument aufzurufen, Daten zu speichern oder einen Service zu nutzen, soll sich der Agent vollständig darum kümmern. Performanz und Zuverlässigkeit sind wichtige Faktoren, deshalb ist die Wahl eines guten Datenflussmodells und eines guten Informationsmodells entscheidend.

Genau diesem Ziel soll sich diese Arbeit widmen: Der Entwicklung eines abgeschlossenen Moduls, welches die Aufgaben des Zugriffs-Managements auf definierte Service-Aktionen und die Autorisierung in Bezug auf Akteur und Ressource übernimmt, sowie die Untersuchung der Eignung von ODRL und XACML für dieses Vorhaben.

2. Use-Cases

2.1. Internetforum

Internetforen sind ein einfaches Beispiel um Rollen-basiertes Zugriffsmanagement zu demonstrieren. In der Regel verwaltet ein Forum verschiedene Themen, in denen Beiträge erstellt werden können. Innerhalb dieser Beiträge können Nutzer kommentieren und sich austauschen. Das kann anonym geschehen oder mit einem registrierten Benutzerkonto, wobei sich die Konten durch die Rolle unterscheiden, die sie in dem Forum erfüllen. Klassischerweise sind das Administratoren, Moderatoren und einfache User. Diese Rollen werden benötigt, um verschiedenen Benutzern verschiedene Rechte zu geben, wie das Erstellen und Kommentieren von Beiträgen für User, das Löschen von Beiträgen für Moderatoren und das Bearbeiten der Foren-Einstellungen für Administratoren.

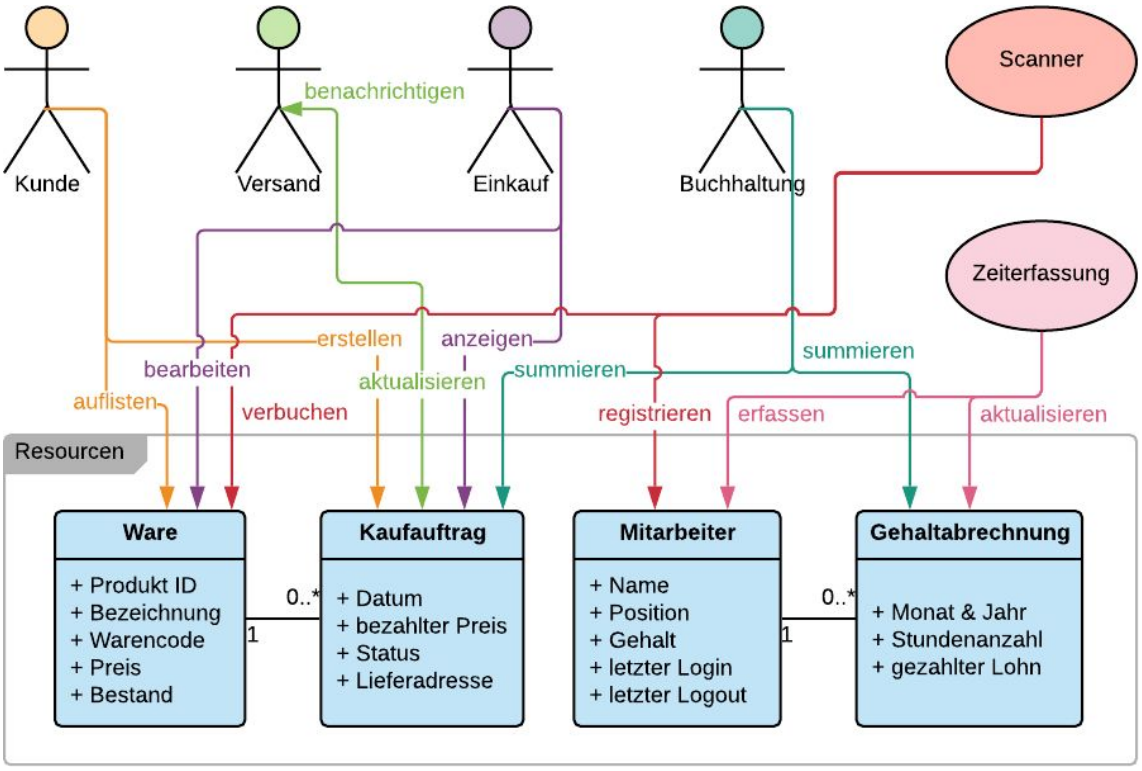
2.2. Einzelhandel

Im Zeitalter des Internets der Dinge ist es jedoch nicht ausreichend, nur Benutzer einer Weboberfläche für das Zugriffsmanagement zu betrachten. Stellen wir uns eine Firma vor, die im Einzelhandel tätig ist und eine Software zur Verwaltung der Warentransporte und der Firmenabläufe einführen will:

- *Kunden* sollen über eine Weboberfläche bequem den aktuellen Warenkatalog einsehen können, um online ihre Bestellung zu machen.
- Mitarbeiter im *Versand* müssen über neue Kaufaufträge benachrichtigt werden, um die Waren zeitgerecht losschicken zu können.
- Um den Ein- und Ausgang von Waren nachzuverfolgen, sollen *Scanner* eingesetzt werden, die den Warencode erfassen und verbuchen.
- Mitarbeiter im *Einkauf* müssen die Lagerbestände und Verkaufszahlen einsehen können, um Produkte nachzubestellen oder Preise anzupassen.
- Die *Buchhaltung* muss über die Einnahmen des Tagesgeschäfts und die Ausgaben durch die Löhne informiert sein, um den Gewinn der Firma abschätzen zu können.

- Für die Abrechnung der Stunden soll eine *Zeiterfassung* mit digitaler Stempelkarte zum Einsatz kommen.

Die *folgende Grafik* gibt einen ungefähren Zusammenhang von Akteuren, Ressourcen und Aktionen wieder, auch wenn eine tatsächliche Implementation in der Realität umfangreicher aufgebaut wäre:



Grafik 1 - Use-Case Diagramm

Um keine Sicherheitslücke in ihrem System zu erzeugen, zum Beispiel dadurch dass der Scanner Zugriff auf die Mitarbeiterliste erhalten kann, will die Firma die Zugriffsrechte mittels Policies einstellen zu können. Policies (in diesem Sinne) sind Sammlungen von Regeln, die Akteure und Ressourcen in Beziehung setzen und bestimmen, unter welchen Bedingungen welche Aktion ausgeführt werden darf oder zusätzlich ausgeführt werden muss. Durch einen genauen Beschrieb der Zugriffsrechte außerhalb irgendeines Quellcodes erhofft sich die Firma außerdem mehr Transparenz und Wartbarkeit des Systems.

3. Informationsmodell

3.1. Open Digital Rights Language (ODRL)

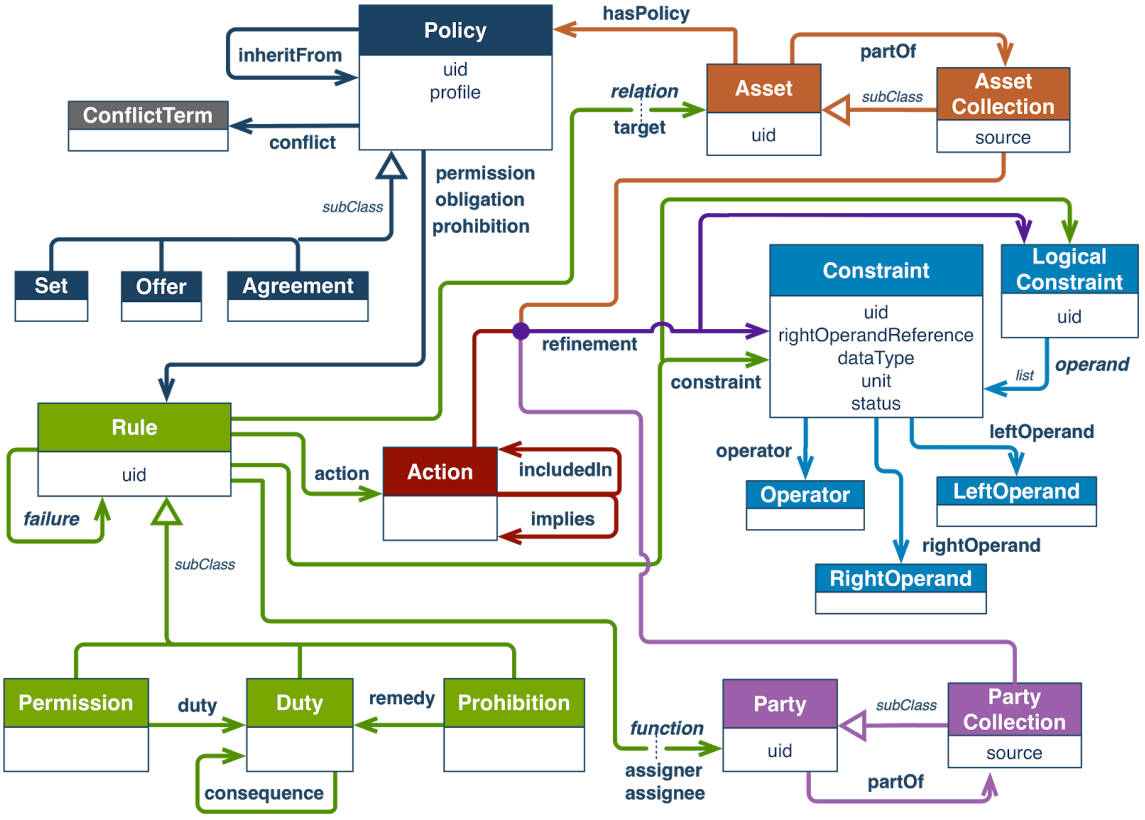
Um die Jahrtausendwende herum und im Zuge der Digitalisierung und digitalen Verbreitung von Medien begann ein Bedarf nach Kontrolle für die Nutzung digitaler Inhalte[2]. Die Entwicklung von DRM-Systemen (Digital Rights Management) hatte begonnen, doch bisher gab es keine einheitliche Semantik diese Rechte auszudrücken, um Interoperabilität zwischen verschiedenen DRMs zu gewährleisten[3]. Die ODRL war ein erster Schritt in Richtung einer nicht lizenzierten, freien Sprache und hat sich im Verlauf der nächsten Jahre als primäre Sprache zum Ausdruck digitaler Rechte durchgesetzt.
Dank des universellen XML-Formats war die ODRL nicht an eine bestimmte Plattform oder Implementation gebunden, jedoch existieren seit Version 2.0 zusätzlich offizielle Serialisierungen für RDF/OWL und JSON-LD, welche heute im Semantic-Web eine große Rolle spielen. Im Folgenden werde ich mich jedoch nur noch auf die JSON-Repräsentation (JavaScript Object Notation) beziehen, da sie leichter zu lesen ist und für die Entwicklung unter JavaScript offensichtliche Vorteile hat.

3.1.1 Grundbegriffe

Die ODRL baut grundsätzlich auf sogenannten *Policies* auf, die die rechtmäßige Nutzung von Ressourcen oder Services[4] beschreiben, wie zum Beispiel eines eBooks oder einer MP3 in einer Online-Mediathek. Je nach Zweck der Policy werden die Unterklassen *Offer*, für ein angebotenes Medium, *Agreement*, für eine vereinbarte Nutzung eines Medium zwischen zwei Parteien, oder *Set* als Standardklasse, für generelle Definition von Nutzung für ein Medium, verwendet. Diese *Policies* enthalten beliebig viele Regeln (*Rules*), die einem Akteur (*Party*) die Erlaubnis, das Verbot oder die Pflicht einer Aktion aussprechen, die er auf einer Ressource (*Asset*) ausführen darf, kann oder

[2] vgl. [13] OASIS: *XML and Digital Rights Management (DRM)*, Unten
[3] vgl. [14] Renato Iannella: *Open Digital Rights Language (ODRL)*, Seite 1
[4] vgl. [1] W3C: *ODRL Information Model 2.2*, Abstract

muss. Bis auf die top-level Aktionen *use* und *transfer* besitzt eine Aktion dabei immer eine *inludedIn* Aktion, von der sie ableitet. Außerdem kann eine Aktion beliebig viele *implies* Aktionen haben, die für ihre Ausführung benötigt werden. *Party*- und *Asset-Collections* dienen dabei der Generalisierung einer Regel auf mehr als einen Akteur oder eine Ressource. Je nach Anwendung kann das schon ausreichen, um zum Beispiel allen eingeloggten Usern auf einer Seite Zugriff auf die angebotenen Medien zu gewähren.



Grafik 2 - ODRL Informationsmodell [5]

Für komplexere Bedingtheiten werden die *Constraints* benutzt, welche die Policies um logische Ausdrücke erweitern und so die Auflösung von Regeln beeinflussen. Ein *Constraint* in Kombination mit einer *Party*- oder *AssetCollection* schränkt die für diese Regel zutreffenden Gruppenmitglieder ein, wie zum Beispiel alle Mitarbeiter eines Unternehmens, die einen aktuellen Zeitstempel haben. Ein *Constraint* in Kombination mit einer *Action* definiert eine Einschränkung der Ausführung dieser Aktion, zum Beispiel dass für einen Kunden die Auflistung der Waren nur den Namen und den Preis enthalten darf. Ein *Constraint* in Kombination mit einer *Rule* definiert die Gültigkeit dieser

[5] [1] W3C: *ODRL Information Model 2.2* - Figure 1

Regel, sodass diese nur angewendet werden kann, wenn alle ihre *Constraints* erfüllt sind, wie zum Beispiel dass das Erstellen einer Gehaltsabrechnung nur möglich ist, wenn im selben Monat nicht schon eine erstellt wurde.

“Das ODRL Standard-Vokabular bietet eine generische Menge an gebräuchlichen Aktionen für *Permissions*, *Prohibitions* und *Duties*, sowie *Policy* Unterklassen, *Constraint* Operanden und Operatoren, *Asset relations* und *Party functions*.”[6] Es ist jedoch auch möglich das Standard-Vokabular durch ein sogenanntes ODRL-*Profile* zu erweitern oder zu ändern, um notwendige Ausdrücke für die eigenen Bedürfnisse zu ergänzen.

3.1.2 Beispiel-Policy

Es gibt meistens verschiedene Wege, ein und denselben Sachverhalt in einer ODRL Policy auszudrücke. Die folgende Policy soll daher hauptsächlich die Syntax verdeutlichen, auch wenn sie in der Art so nicht eingesetzt wurde:
(Für eine Beschreibung der Policy bitte die Kommentare beachten.)

```
{
  // Alle Attribute mit einem @ gehören zur JSON-LD
  // Notation. Durch den @context wird beschrieben,
  // wie die vorliegenden Daten aufgebaut sind.
  "@context": "http://www.w3.org/ns/odrl.jsonld",
  "@type": "Policy",
  // uid: IRI der Policy
  "uid": "http://example.com/policy:001",
  "profile": "http://example.com/odrl-profile",
  "permission": [{

    // assignee: Die Erlaubnis gilt für alle Parties aus der
    // Party Collection 'kunden'.
    "assignee": "http://example.com/party-collection/kunden",

    // target: Für den Kunden sollen alle Assets aus der
    // Asset Collection 'waren' aufgelistet werden,
    // die einen vorhandenen Bestand haben (refinement).
    "target": {
      "@id", "http://example.com/asset-collection/waren",
      "refinement": [{
        "leftOperand": "Bestand",
        "operator": "gt", // = greater than
        "rightOperand": { "@value": "0", "@type": "xsd:integer" }
      }]
    },
```

[6] [1] W3C: *ODRL Information Model 2.2*, Kapitel 3.1: *ODRL Profile Purpose* (übersetzt)

```
    // action: Für den Kunden sollen nur diejenigen Details
    // aufgelistet werden, die als Info markiert sind (refinement).
    "action": [{
      "rdf:value": { "@id": "http://example.com/action/auflisten" },
      "refinement": [{
        "leftOperand": "istKundenInfo",
        "operator": "eq", // = equal
        "rightOperand": { "@value": "true", "@type": "xsd:boolean" }
      }]
    }]

  }]
}
```

Beispiel 1 - Entwurf einer Policy

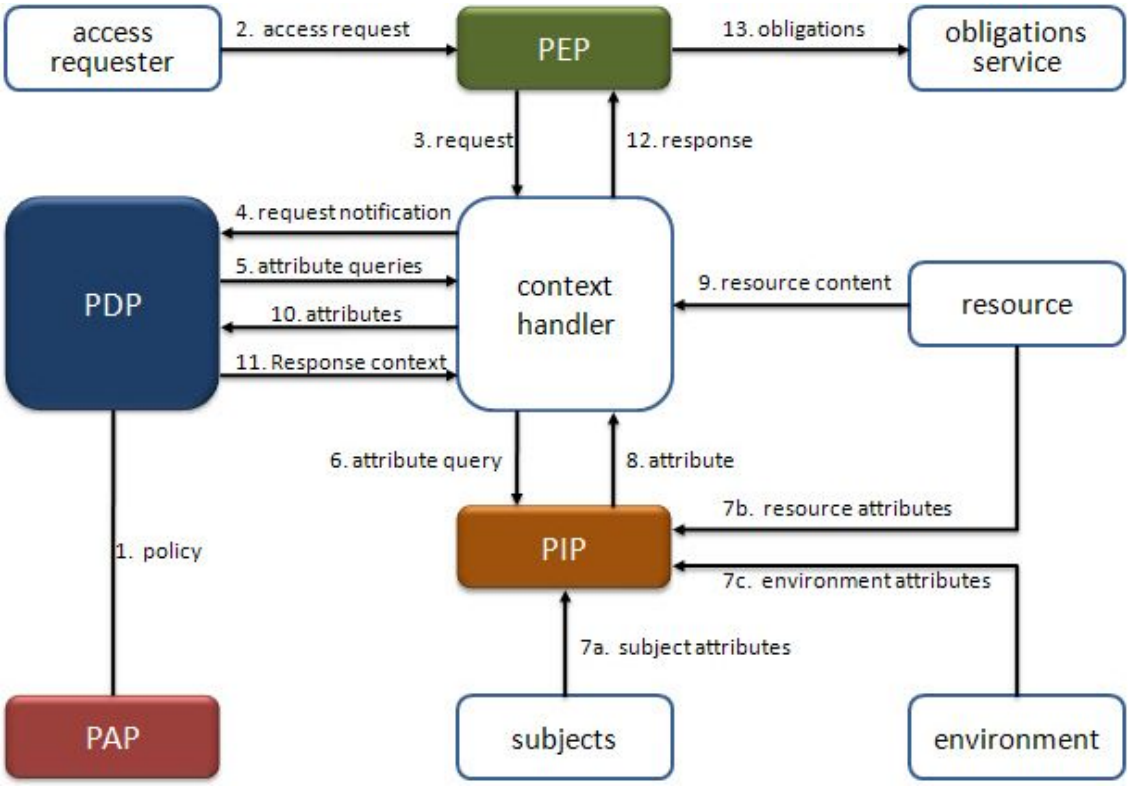
3.2. eXtensible Access Control Markup Language (XACML)

Das XACML Informationsmodell umfasst genau wie die ODRL eine ganze Sprache zum Ausdruck von Policies, allerdings möchte ich nicht weiter darauf eingehen, da sie nur eine Alternative zur ODRL darstellt und das zusätzliche befassen mit einem weiteren Informationsmodell den Rahmen dieser Arbeit sprengen würde. Stattdessen bietet die XACML für diese Arbeit einen anderen Nutzen und das ist ein Datenflussmodell, welches in vielen anderen Bereichen bereits adaptiert wurde, darunter Axiomatics ABAC Autorisierung[7], die API Integrationsplattform des WSO2[8] und das neue Sicherheitsframework des Fraunhofer IESE[9].

Die Hauptbestandteile des Modells sind vier Module, welche durch einen Request auf eine Ressource angefahren werde. Der *Policy Enforcement Point* (PEP) hat die Aufgabe, jeden Request entgegenzunehmen und die Zugriffskontrolle durchzuführen. Dafür erstellt der PEP eine Anfrage, welche an den *Policy Decision Point* (PDP) weitergeleitet wird. Der PDP wertet die Anfrage aus und ermittelt eine Entscheidung für den Zugriff. Um das zu tun benötigt der PDP als erstes Informationen über die angeforderte *Ressource* und das anfordernde *Subjekt*. Diese erhält er von dem sogenannten *Policy Information Point* (PIP), welcher außerdem dazu genutzt werden kann, um in etwaigen Policies vorkommende Umgebungsvariablen aufzulösen. Sämtliche Policies werden dem PDP durch den Policy Administration Point (PAP)

[7] vgl. [4] Axiomatics: *XACML Reference Architecture*
[8] vgl. [5] WSO2: *Access Control and Entitlement Management*
[9] vgl. [6] Fraunhofer IESE: *IND²UCE Sicherheitsframework für Datennutzungskontrolle*

bereitgestellt, welcher auch eine Schnittstelle zum Erstellen und Ändern von Policies bieten muss. Der PDP beendet seine Ausführung mit der Entscheidung auf Zugriff oder Ablehnung, welche der PEP dazu nutzt, die Ressource freizugeben oder nicht.[10]



Grafik 3 - XACML Datenflussmodell [11]

Eine fünfte Komponente ist der *Context handler*, welcher im Mittelpunkt steht und die Vermittlung der Daten koordiniert. Bei einer verteilten Datenlandschaft und skalierenden Systemen, spielt der *Context handler* eine wichtige Rolle. Da in kleineren lokalen Anwendungen die Policy Points auch direkt miteinander kommunizieren können, kann in dem Fall darauf verzichtet werden. Funktionen die der *Context handler* koordiniert hätte, werden dann logischerweise direkt aufgerufen.

Neben den *Policy Points* definiert die XACML auch den sogenannten *Request-* und *Response Context*. Der *Request Context* wird vom PEP erstellt und enthält die für den Request benötigten Attribute über die Aktion, welche ausgeführt werden soll, die Ressource, auf der die Aktion ausgeführt werden soll, das Subjekt, welches den Request anfordert, und etwaige Umgebungsvariablen, die

[10] vgl. [3] OASIS: *eXtensible Access Control Markup Language (XACML) Version 3.0*
[11] [23] Yoon Jae Kim: *Access Control Service Oriented Architecture Security* - Figure 3

für den Request wichtig sein könnten. Der *Response Context* ist die Antwort des PDP und enthält zum einen die Entscheidung, welche “Permit”, “Deny”, “NotApplicable” oder “Indeterminate” sein kann, zum anderen zusätzliche Statusinformationen und Obligationen, welche ausgeführt werden müssen bevor der Zugriff auf die angeforderte Ressource gewährt werden kann.[12]

4. Entwicklungsumgebung

4.1. Node.js

Wer anfang der 2000er Jahre selber einen Webservice entwickeln wollte, würde wahrscheinlich entweder die Java API für XML Web Services (JAX-WS) oder C# und das ASP.NET Framework nutzen. Weil Service-Requests hier in einem Thread-Pool verarbeitet werden, muss man als Entwickler darauf achten, dass IO-Operationen den Thread nicht blockieren. Bei einer hohen Rate an Anfragen kann das zur Schwierigkeit werden. Ryan Dahl erkannte in JavaScript als Event-basierte, nicht-blockierende Programmiersprache das Potenzial, dieses Problem in den Griff zu bekommen und begann 2009 mit der Entwicklung von Node.js[13]. Erst 6 Jahre später jedoch rückte Node.js vollständig in den Fokus der Entwickler durch die Zusammenführung mit io.js, welche sich ehemals von dem Projekt abgespalten hatten um schnellere Fortschritte zu machen.
Im Gegensatz zu C# läuft JavaScript single-threaded und behandelt die vielen Serveranfragen durch eine Event-loop, welche blockierende IO-Operationen in eine Event-queue einreiht und andere Anfragen in der Zwischenzeit bearbeiten kann, bis ein Callback die Ausführung des Scripts an der unterbrochenen Stelle fortsetzt. Skalierbarkeit auf mehrere Prozessoren kann in Node.js dennoch realisiert werden und könnte sinnvoll mit Mechanismen der Lastverteilung kombiniert werden, sodass dieselbe Kommunikation zwischen Prozessoren wie zwischen Maschinen genutzt werden würde.

[12] vgl. [23] Yoon Jae Kim: *Access Control Service Oriented Architecture Security*, Kapitel 3
[13] vgl. [17] Dr. Holger Schwichtenberg, Martin Möllenbeck: *JavaScript überall - Webserverprogrammierung mit Node.js*

Ein weiterer Grund für die wachsende Beliebtheit von Node.js ist der Umstand, dass die Programmiersprache JavaScript ohnehin in fast jedem Web-Frontend Anwendung findet und für neue Entwickler einen leichten Einstieg in das Programmieren bieten, durch die fehlende statische Typisierung, Kompilierung zur Laufzeit, geringen Boilerplate-code und eine große Entwickler-Community.
Der Node Package Manager (npm) beinhaltet aktuell über 800 000[14] Module und steht dadurch um mehr als das Doppelte auf dem ersten Platz an Developer Content. Selbstverständlich ist dies kein Indikator für die Qualität der Entwicklungsumgebung, jedoch ermöglicht einem eine Vielzahl an Modulen erst die Wahl, das passende anhand von Zuverlässigkeit, Dokumentation, Funktionalität, Performance oder anderer Aspekte auszuwählen.

4.2. Neo4j

Neo4j gehört zu den Graphendatenbanken und somit zur Kategorie der NoSQL-Datenbanken[15], auch wenn sich Neo4j von klassischen NoSQL-Datenbanken wie Redis oder MongoDB distanziert. Im Gegensatz zu relationalen Datenbanken, die Tabellen und Zeilen nutzen um Datensätze und ihre Attribute zu speichern, werden in Graphendatenbanken Knoten und Kanten verwaltet, um Datenstrukturen zu speichern. Zwar lässt sich jede Graphendatenbank auch in ein entsprechendes relationales Datenbankschema übertragen, allerdings hat die Abbildung im Graphen den Vorteil, gerade bei stark vernetzten Daten wesentlich intuitiver zu sein, wodurch Datenbankabfragen leichter formuliert werden können. Abfragen über die Relation von Knoten und Extraktion von Teilen des Graphen lassen sich somit erheblich einfacher konstruieren und sind dank der expliziten Graphenstruktur sogar performanter als die implizite Verknüpfung durch JOINs in SQL-Datenbankabfragen. Neo Technologies hat für Abfragen an Neo4j dazu die *Cypher Query Language* entwickelt, dessen Semantik statt auf Datensätze an die Struktur von Knoten und Kanten angelehnt ist.

[14] <http://www.modulecounts.com/>, 13.05.2019
[15] vgl. [18] Stefan Luber, Nico Litzel: *Was ist eine Graphdatenbank?*

Eine umfassende Dokumentation und eine kostenlose Community-Edition machen den Einstieg in Neo4j für Entwickler einfach, wenngleich die Lernkurve für komplexere Datenbankabfragen wahrscheinlich ähnlich hoch ist, wie bei SQL.

4.3. MongoDB

MongoDB ist eine in der Handhabung einfache NoSQL-Datenbank, die Daten in flexiblem, JSON-ähnlichen Dokumenten speichert.[16] Das Finden von Objekten anhand von bekannten key-values ist sehr komfortabel und bietet somit eine denkbar einfache Lösung um fehlende Attribute von Subjekten und Ressourcen aus einer Datenquelle aufzufüllen, um mit den vollständigen Angaben über eine Entität weitermachen zu können. Zusätzlich können die Daten durch Collections gruppiert werden, um zum Beispiel Typen zu unterscheiden und unterschiedliche Indizierung auf den Daten zu ermöglichen.
Dank einer kostenlosen Community-Edition und dem für JavaScript komfortablen JSON Format, ist MongoDB eine attraktive Wahl für viele Node.js Anwendungen.

[16] vgl. [9] MongoDB, Inc.: *What is MongoDB?*

5. Umsetzung

5.1. Kombination der Modelle

Die ODRL ist eine sehr umfangreiche Sprache um Rechte über Ressourcen oder Services auszudrücken, jedoch kann ihre Vielseitigkeit auch ein Nachteil sein. Es wird nicht vorgeschrieben, wie man die Daten verarbeiten oder Rechtmäßigkeiten feststellen soll, stattdessen muss sich ein Service selber dadurch auszeichnen, ein vorgegebenes *ODRL Profile* zu erfüllen. Wie er das macht, wird nicht angegeben. Der Mangel an Algorithmen ist eine große Herausforderung für eine Implementation, jedoch beinhaltet die XACML ein oft eingesetztes Datenflussmodell, welches Abhilfe schaffen kann.

Da die beiden Modelle teilweise unterschiedliche Bezeichnungen haben, möchte ich zum Verständnis zwei Begriffe erklären, welche die beiden Sprachen unterschiedlich definieren:

- Um einen Service oder eine Ressource zu beschreiben, auf der eine Aktion ausgeführt werden soll, vereinheitlicht die XACML dies unter dem Begriff *Resource*, wohingegen die ODRL den Begriff *Asset* benutzt, meistens als *Target* eines Requests.
- Um einen Akteur zu beschreiben, der eine Aktion auf einer Ressource ausführen will, verwendet die XACML den Begriff *Subject*, wohingegen die ODRL den Begriff *Party* benutzt, meistens als *Assignee* für den anfordernden Akteur oder *Assigner* für den Berechtigung erteilenden Akteur.

Obwohl die ODRL die zentrale Rolle in der Umsetzung spielt, da sie die verwendete Policy-Sprache beinhaltet und somit die Syntax für die Policy-Auflösung vorgibt, ähnelt der Aufbau des Codes eher der XACML.

Ein *PEP* dient als Gateway für alle Requests, welche er in einem *Request Context* verpackt und an den *PDP* sendet. Dieser holt sich alle Informationen über die beteiligten Ressourcen und Subjekte von einem *PIP*. Dadurch erhält er auch die benötigten UIDs der Assets und Parties für die ODRL Policies. An einem *PAP* werden alle benötigten Policies aus dem Datenbestand mittels der UIDs vom PIP angefordert und dem PDP in gültigem ODRL-JSON-Format zur

Verfügung gestellt. Sollte der PDP weitere Informationen benötigen, welche in den Policies referenziert werden, kann er diese am PIP anfordern. Ansonsten wendet der PDP seine Algorithmen zur Policy-Auflösung auf die gesammelten Daten an und speichert Entscheidungen und Anweisungen in einem *Response Context*. Diesen erhält der PEP als Antwort und muss daraus die angeforderten Aktionen ausführen und eine Antwort auf den ursprünglichen Request verfassen, alles im Rahmen des ODRL-Modells (vgl. *Grafik 4* und *Grafik 5*).
Da Assets, Parties und Policies ohnehin im PAP verzeichnet sind, liegt es nahe die Abhängigkeiten der Aktionen auch dort zu verwalten. Weil die Aktionen jedoch am PEP definiert werden hat sich in der Praxis gezeigt, dass dieser die *includedIns* und *implies* selber verwalten und alle benötigten Aktionen in einem *Request Context* gleichzeitig anfordern sollte.

5.2. Abweichungen vom Modell

Eine zentrale Rolle in der XACML besitzt der *Context Handler*. Wie schon weiter oben erklärt, ist diese Komponente vor allem für verteilte Systeme eine Hilfe. Auch mit dieser Implementation soll eine gewissen Skalierbarkeit erhalten bleiben, jedoch ist eine abgesicherte Kommunikation zwischen den einzelnen Modulen notwendig für einen Policy Agenten, der über mehrere Node.js-Instanzen verteilt ist. Da dies nicht der Fokus der Arbeit sein sollte, beschränkte ich mich auf die Skalierbarkeit in einer Node.js-Instanz, was bedeutet, dass mehr als ein PEP, PDP etc. instanziiert und verbunden werden können. Ein *Context Handler* ist dafür vernachlässigbar, da die JavaScript Laufzeitumgebung ausreichend ist.
Eine weitere Abweichung ist der Umgang mit den für Assets vorgesehenen Aktionen. In der ODRL schreibt das *Policy Profile* vor, welche Aktionen vorhanden sind und von Policies bzw. Anfragen benutzt werden können. Das Basisprofil besitzt eine Fülle an gebräuchlichen Aktionen, jedoch stehen laut Dokumentation nur die Top-Level-Aktionen *use* und *transfer* fest. Statt feste Aktionen für Ressourcen vorzugeben, deren Nutzung aus aktueller Perspektive unklar ist, und somit die mögliche Semantik für Ressourcen einzuschränken, sollen alle unterstützten Aktionen vom Anwendungsentwickler am PEP selber definiert werden und nur die Top-Level-Aktionen vom Policy Agenten gestellt

werden. Auch wenn die autoritative Quelle für Assets und Parties der PAP ist, entsteht dadurch kein Problem, da der PEP die abhängigen Aktionen nur durchführen darf, wenn er durch einen erfolgreichen *Decision Request* die Erlaubnis dazu bekommen hat.

Da die ODRL jedoch als *Digital Rights Language* ursprünglich dazu konzeptioniert wurde, Medien mit Nutzungsbedingungen auszustatten, ist es normalerweise unerlässlich ein öffentlich gültiges ODRL-Profil zu implementieren, um Medien zu verwalten, die dieses Profil referenzieren. Da jedoch nicht nur Medien mit Nutzungsbedingungen, sondern jegliche Form von Daten oder Services mit unter Umständen eigener Semantik verwaltet werden sollen, die eine Webanwendung oder ein Web Service benötigt, wird es dem Anwendungsentwickler freigestellt, ein bestimmtes ODRL-Profile zu implementieren oder völlig eigene Aktionen zu definieren.

5.3. Besondere Anforderungen

5.3.1. Implementation als Node.js-Package

Im Node.js-Umfeld zu entwickeln bedeutet nicht nur, dass der eigene Code in einem Projekt eingesetzt wird und keine andere Verwendung findet. Durch den Node Package Manager ist es möglich Programmcode zu entwickeln, der für eine Vielzahl von Projekten eingesetzt werden kann. Das ist vergleichbar mit einer Studie, die publiziert wird und deren Resultate in wissenschaftlichen Arbeiten auf der ganzen Welt genutzt werden können. Dieser Mehrwert der Wiederverwendbarkeit soll auch für diese Arbeit einen großen Aspekt spielen und ist der Grund, warum der Programmcode als Node.js-Package konzeptioniert wurde. Aus diesem Grund soll auch auf die folgenden Punkte großen Wert gelegt werden:

- Alle Klassen müssen ein leicht verständliches Interface und an das Informationsmodell (ODRL & XACML) angelehnte Methoden haben, damit eine konsistente Dokumentation zustande kommen kann.
- Um Verwirrung und Unklarheit in der Nutzung zu vermeiden, soll es möglichst keine unerwarteten Fehler geben, wenn der Nutzer etwas falsch implementiert hat. Aus diesem Grund müssen alle möglichen

Fehlerzustände abgefangen und durch dokumentierte Exceptions ersetzt werden. Da es bei längerer Nutzung auf einem Server immer wieder zu Fehlerzuständen kommen kann, könnte ein allgemeines Auditieren zusätzlich hilfreich sein, um eine Anwendung leichter debuggen zu können.

- Um auch nachträglich neue Funktionen implementieren zu können ohne die Abwärtskompatibilität zu verletzen, ist es möglich ein striktes Basismodell zu schaffen, welches durch Vererbung erweitert werden kann. Dies bedeutet konkret, dass zum Beispiel der PEP im Basismodell nur den *Decision Request* zum PDP beherrscht, jedoch eine Klasse später vom PEP ableiten kann, die nativ eine Socket.io Schnittstelle stellt oder eine Webanwendung hostet. Auch ein PIP wäre denkbar, der im Basismodell nur Meta-Informationen verwaltet, dessen Ableitung zusätzlich direkten Zugriff auf eine referenzierte Ressource gewähren kann.

5.3.2. Steigerung der Produktivität

Für einen Webservice nimmt Sicherheit und Zuverlässigkeit eine wichtige Rolle ein. Auch können sich die Anforderungen mit der Zeit ändern, sodass immer wieder Anpassungen notwendig sind. Sollten dennoch Fehler vorkommen, müssen diese nachverfolgbar und reproduzierbar sein. Doch die Art und Weise wie wir unseren Code schreiben, hat einen enormen Einfluss auf diese Probleme. Sich ändernde Anforderungen führen oft dazu, dass mehrere Entwickler am selben Code immer wieder Verbesserungen durchführen: hier eine if-Anweisung um eine neue Beschränkung zu implementieren, dort ein geänderter Funktionsaufruf, weil ein zusätzliches Argument beachtet werden muss. Die schlimmste Eigenart tritt zum Vorschein, wenn Code an anderer Stelle fast die Aufgabe erfüllt, die man neu implementieren möchte. Dann wird kopiert und eingefügt und hier und da ein paar Details verändert. Nach gewisser Zeit nimmt so die Zuverlässigkeit eines Programms stark ab, weil kein Entwickler mehr einen Überblick hat, wie es überhaupt funktioniert.

Auch für dieses Problem soll der Policy Agent eine Verbesserung sein. Sämtliche verwalteten und in der Art des ODRL definierten Aktionen, müssen mit samt Callback-Funktion am PEP angegeben werden. Wie wir aus der ODRL

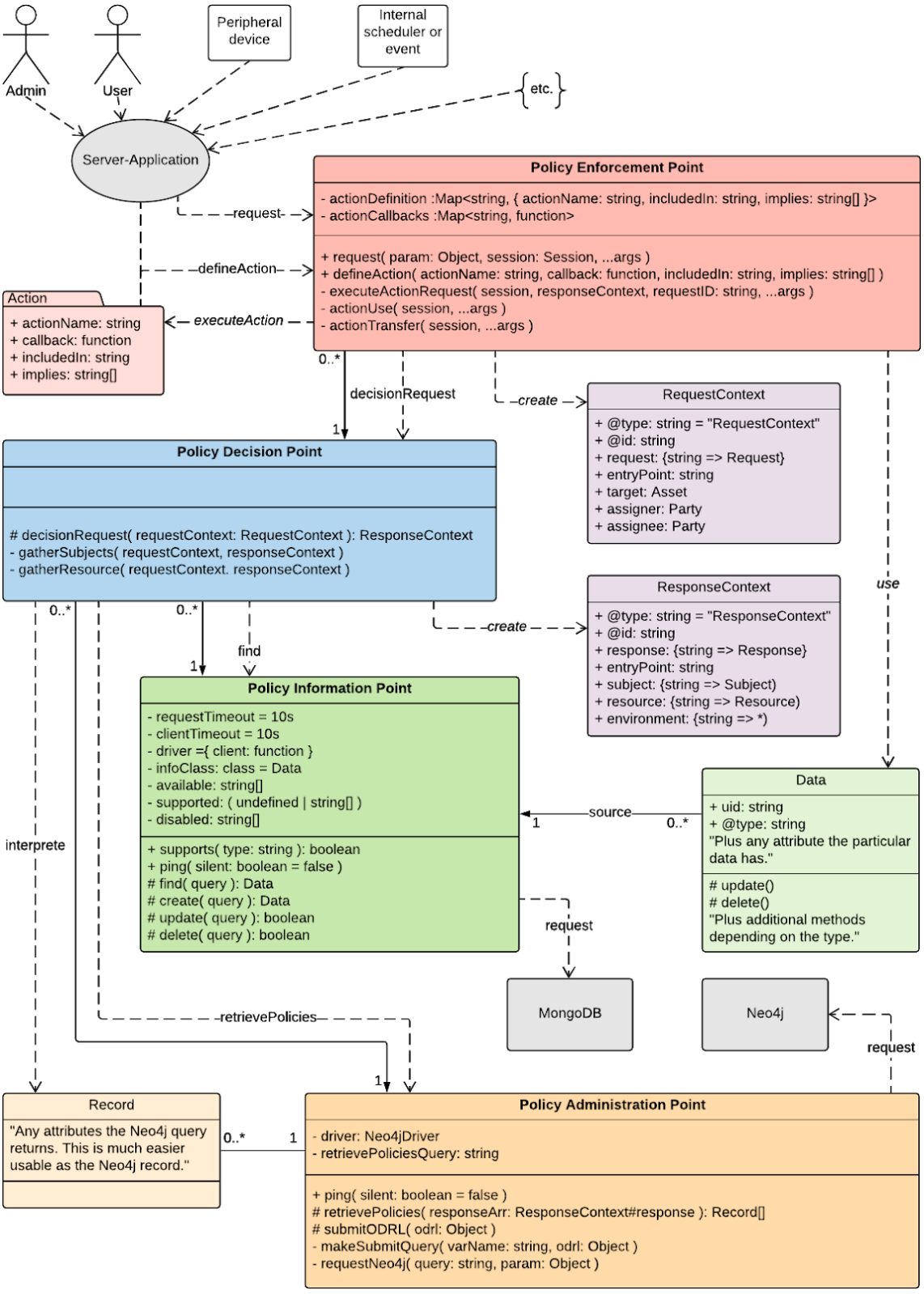
wissen, gehört dazu ein *includedIn*, welches in diesem Fall die Vererbung einer Aktion anzeigt, wohingegen *implies* die zusätzlich verwendeten Aktionen beschreibt. Durch geeignete Überprüfungen am PEP müssen zirkuläre Definitionen ausgeschlossen werden, jedoch verspricht diese Herangehensweise, dass alle möglichen Aktionen vor einem Request bekannt sind und deren Ausführung sichergestellt werden kann. Durch die autonome Ausführung der angeforderten Aktionen nach dem Schema der ODRL, kann der PEP nach einem erfolgreichen *Decision Request* nicht nur ein gültiges Zugriffsschema bereitstellen, sondern auch die Integrität der Ausführung laut Policy-Definition gewährleisten, indem jeder Aktion nicht mehr und nicht weniger als das bereitgestellt wird, zu dem sie berechtigt worden ist.

Dieses Vorgehen reduziert die durch eigene Fehler oder Nachlässigkeiten begangenen Datenschutzverletzungen und legt die Verantwortung zur Einhaltung sich eventuell ändernder Anforderungen in den Policy-Bestand, was den Programmcode unangetastet lässt. Dieser dient im Gegensatz nun nur noch dazu, die tatsächlich definierte Aktion auszuführen.

Um dieser Verantwortung bei gleichbleibender Flexibilität der Aktionen gerecht zu werden, ist eine statische Verwaltung der Policies für den PAP, wie sie für das Digital Rights Management vielleicht ausreichend wäre, nicht genug, denn ein Service macht unter Umständen stetige Änderungen am Datenbestand und muss selbstständig Berechtigungen aktualisieren. Eine relationale Datenbank wäre für dieses Vorhaben eine valide Option, jedoch bieten Graphendatenbanken erheblich mehr Komfort und Übersicht, was diese Aufgabe betrifft. Die Policies drücken eine logische Verbundenheit von *Rules*, *Actions*, *Assets*, *Parties* und *Constraints* aus, welche sich im Graphen durch die Verbindungen der Nodes widerspiegelt und dadurch einen einfachen und intuitiven Überblick gewährt, der große Ähnlichkeit mit der Modellabbildung aus der ODRL-Dokumentation hat. Da eine Abfrage all diese Typen benötigt, würden bei einer herkömmlichen SQL-Datenbank viele JOINS gemacht werden müssen. Aus diesem Grund verspricht Neo4j nicht nur eine bessere Übersichtlichkeit der Daten, sondern auch eine einfachere Abfrage der Assets und Parties inklusive der Suche aller beteiligter Policies und eine höhere Performanz der Verarbeitung dieser Abfragen.

5.4. Implementation

5.4.1. Aufbau des Modells



Grafik 4 - Klassendiagramm des PolicyAgent

Der Vollständigkeit halber und zur leichteren Orientierung in der aktuellen Implementation, sieht man in *Grafik 4* ein Klassendiagramm über den Großteil der Anwendung, lediglich das Logging und Error-Handling wurde aus Gründen der Übersichtlichkeit weggelassen. In Kombination mit *Grafik 5* lässt sich der Ablauf eines Requests sehr gut nachverfolgen und bestätigt das Vorgehen des in Kapitel 3.2 beschriebenen Datenflussmodells der XACML. Es lassen sich jedoch auch feine Unterschiede feststellen, wie zum Beispiel das sammeln der beteiligten Policies, welche in *Grafik 3* zwischen Schritt 10 und 11 aktiv vom PDP angefordert werden würden, statt schon zu Beginn vom PAP zur Verfügung gestellt zu werden, oder das anfordern von Subjekten und Ressourcen direkt am PIP, statt über den fehlenden *Context handler*.
Durch die Verarbeitung der Aktionen am PEP (siehe Kapitel 5.4.4) ergibt sich ein weiterer Unterschied: Obwohl der *Response Context* in der XACML mit der Entscheidung und den Obligationen auskommt, werden hier zusätzlich die beteiligten Ressourcen und Subjekte aufgeführt, um die Ausführung der Aktionen mit den entsprechenden Daten zu ermöglichen.

5.4.2. Request- und Response-Context

Tatsächlich vom Policy Agent generierter Request Context und passender Response Context.

(Für eine Beschreibung der Kontexte bitte die Kommentare beachten.)

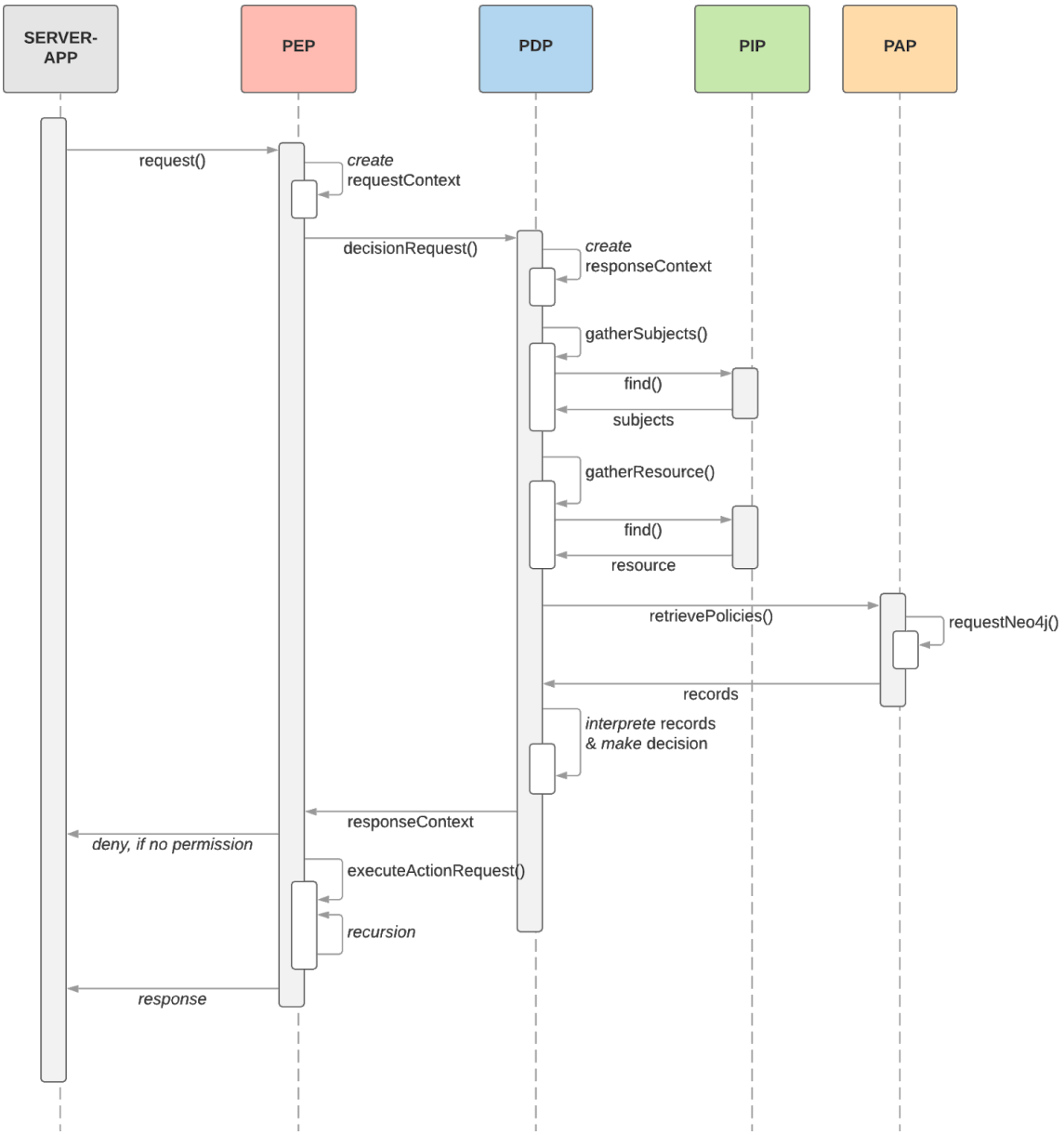
```
{
  "@type": "RequestContext",
  "@id": "7f291b63-5961-4b04-aeb8-...",

  // request enthält alle beteiligten Aktionen
  // und spezifischen Definitionen.
  "request": {
    // Die id wird für jeden Request generiert und dient der
    // späteren Koordination der Abhängigkeiten.
    "readFile-78cede71-5c7a-4da5-8923-...": {
      "id": "readFile-78cede71-5c7a-4da5-8923-...",
      "action": "readFile",
      // implies und includedIn werden über die id eines
      // selbstständigen requests angegeben, da eine implizite
      // Aktion unterschiedliche Ressourcen benötigen könnte.
      // Außerdem können über diese Trennung separate
      // Obligationen beim response vergeben werden.
      "includedIn": "use-db0c825e-1d18-444e-ab6a-...",
      "implies": []
    },
    "use-db0c825e-1d18-444e-ab6a-...": {
      "id": "use-db0c825e-1d18-444e-ab6a-...",
      "action": "use",
      "implies": []
    }
  },

  // Hier ist das target allgemeingültig, es kann allerdings durch
  // eine spezifische Definition in den requests
  // überschrieben werden.
  "target": {
    "@type": "File",
    "@id": "/"
  },

  // Der entryPoint kennzeichnet die initiale Aktion.
  "entryPoint": "readFile-78cede71-5c7a-4da5-8923-..."
}
```

Beispiel 2 - RequestContext

```
{
  "@type": "ResponseContext",
  "id@": "0864f263-f34f-4cd0-8570-...",

  // Für jeden request entsteht hier ein response
  // mit identischer id.
  "response": {
    "readFile-78cede71-5c7a-4da5-8923-...": {
      "id": "readFile-78cede71-5c7a-4da5-8923-...",
      "action": "readFile",
      "includedIn": "use-db0c825e-1d18-444e-ab6a-...",
      "implies": [],
      // Subjekte und Ressourcen werden hier nur noch
      // durch ihre uid angegebe.
      "target": "/index.html",
      "decision": "Permit"
    },
    "use-db0c825e-1d18-444e-ab6a-...": {
      "id": "use-db0c825e-1d18-444e-ab6a-...",
      "action": "use",
      "implies": [],
      "target": "/index.html"
    }
  },

  // Hier tauchen die tatsächlichen Daten von Ressourcen,
  // Subjekten und Umgebung auf, abhängig ihrer uid. Dieses
  // “cachen” der Daten spart Datenübertragung bei multipler
  // Referenzierung von Ressourcen.
  "subject": {},
  "resource": {
    "/index.html": {
      "@type": "File",
      "@id": "/",
      "path": "/index.html",
      "mimeType": "text/html"
    }
  },
  "environment": {},

  "entryPoint": "readFile-78cede71-5c7a-4da5-8923-...",
  // Dies ist die Entscheidung, welche allerdings auch auf
  // jedem response-Eintrag gesondert vergeben werden kann.
  "decision": "Permit"
}
```

Beispiel 3 - ResponseContext

5.4.3. Ein Request im Detail



Grafik 5 - Sequenzdiagramm eines Requests

Wie schon in *Kapitel 5.4.1* erwähnt, folgt das Sequenzdiagramm größtenteils dem Verlauf des Datenflussmodells der XACML und lässt sich leicht verstehen, deshalb möchte ich nicht näher auf den Ablauf eingehen und stattdessen eine Stelle erläutern, die in der restlichen Arbeit kaum Erwähnung findet: *interprete records & make decision*.

In *Kapitel 5.5.1* wird bereits gesagt, dass Constraints nicht implementiert wurden, jedoch ergeben diese nur einen weiteren Faktor, ob und unter welchen Bedingungen eine Regel angewendet wird. In erster Linie lässt sich die Entscheidungsfindung allerdings auf ein einfaches Grundkonzept zurückführen:

Für jeden Request werden die beteiligten Policies angefordert und separat ausgewertet. Der Status der Entscheidung beginnt mit “NotApplicable” und kann nur dann zu “Permit” werden, falls eine *Permission* gefunden wurde. Wird eine *Prohibition* gefunden, wird der Status zu “Deny”, auch wenn zuvor eine *Permission* ausgesprochen wurde. Dies entspricht dem “prohibit”[17] *ConflictTerm* per Default, da auch der *ConflictTerm* bisher nicht implementiert wurde (siehe *Kapitel 5.5.2*). Der Status “Indeterminate” wird derzeit nicht verwendet, was jedoch einen implementierten *ConflictTerm* oder ein anderes Default-Vorgehen voraussetzt.[18]

5.4.4. Verarbeitung der Aktionen

Die Aktionen in der ODRL legen ganz klar fest, was mit einem Asset gemacht werden kann, da alle übrigen, nicht definierten Aktionen nicht vom System verwaltet werden und somit auch nicht *enforced* werden können. Laut XACML hat der PEP seine Aufgabe erfüllt, wenn er “den Zugriff auf eine Ressource zulässt, falls es erlaubt wurde, und ansonsten den Zugriff verweigert.”[19] Es kann allerdings nicht sichergestellt werden, was der Anwender tatsächlich mit der erhaltenen Ressource macht. Genauso wenig kann sichergestellt werden, dass eine Änderung der Policies auch eine Änderung in den ausführenden Aktionen bewirkt, falls für alle Aktionen freier Zugriff auf die Ressource gestattet wird.
Um das *Enforcement* von Nutzungsrechten robuster zu machen, wird dem PEP in diesem Fall auch die Aufgabe über die Ausführung der Aktionen übertragen. Eine Registrierung von Callback-Funktionen für alle benötigten Aktionen am PEP ermöglicht die Kontrolle über den *Scope* und die *Argumente* des Funktionsaufrufs. Nur die Top-Level-Funktionen erhalten Zugriff auf das tatsächliche *Target* eines Requests, alle anderen Funktionen bekommen als Target nur das Resultat des *includedIns*. Die *implies* werden der Aktion als einfache Funktionsaufrufe zur Verfügung gestellt, da man im Voraus nicht wissen kann, ob und wann ein implies tatsächlich gebraucht wird.

---

[17] vgl. [1] W3C: *ODRL Information Model 2.2*, Kapitel 2.10: *Policy Conflict Strategy*
[18] Entscheidungsstatus: vgl. [3] OASIS: *eXtensible Access Control Markup Language (XACML) Version 3.0*, Kapitel 1.1: *Glossary - Authorization decision*
[19] [3] OASIS: *eXtensible Access Control Markup Language (XACML) Version 3.0*, Kapitel 3.1: *Data-flow model*, Step 14

Auch *Obligations* können auf diese Art und Weise ausgeführt werden. Zum Beispiel könnte die Voraussetzung für das Erstellen eines Kaufauftrags von einem Kunden die Überweisung des Preises sein. Die Aktion *Kaufauftrag erstellen* des Kunden wird folglich nur dann ausgeführt, falls die Obligation des *Geldüberweisens* erfolgreich beendet wurde.
Erst durch die zentrale Verwaltung und Deklaration der Aktionen mithilfe von ODRL Syntax, kann eine automatisierte Abarbeitung und Einhaltung der strukturellen oder durch ODRL Policies gegebenen Bedingtheiten von Aktionen gewährleistet werden.

5.4.5. Use-Case Setup

Um eine bessere Vorstellung für die Aktionen zu bekommen, zeigt das *Beispiel 4* einen Ausschnitt aus der Aktionen-Definition des Use-Cases aus *Grafik 1*. Genauer gesagt beinhaltet das Beispiel die *readFile* Aktion, welche zum darstellen der Website notwendig ist und die *kunde:auflisten* Aktionen zum auflisten der verfügbaren Waren.
(Das Resultat ist auch im Live-Beispiel zu beobachten.)

```
pep.defineAction( "readFile" ,
async function (session, response) {
    // readFile ist includedIn use, deshalb reflektiert this.target
    // die Daten zur aktuellen Ressource.
    let target = await this.target();

    // [...] <- error-handling
    let fileBuffer = await _promify(
        Fs.readFile,
        Path.join(__dirname, target.path)
    );

    response.type(target.mimeType).send(fileBuffer);
} , "use", [] );

pep.defineAction( "listMembers" ,
async function (session) {
    let target = await this.target();

    // [...] <- error-handling

    let
        memberIDs = _listMembers[target["@id"]],
```

```
        // Aktuell ist es noch nicht möglich, die Mitglieder
        // einer Collection zu ermitteln, deshalb ist
        // folgende Methode ein bisschen unsauber.
        members = await Promise.all(memberIDs.map((id) =>
            pep.request({
                "action": "use",
                "target": {
                    "@type": target["@id"],
                    "@id": id
                },
                // Auch das Subject ist von hier erreichbar.
                "assignee": this.assignee
            }, session)
            // [...] <- try catch
        ));

    return members.filter(val => val);
} , "use" , [] );

pep.defineAction( "kunde:auflisten" ,
async function (session, response) {
    // Da die Aktion von listMembers erbt, ist das Target
    // ein Array der Mitglieder der Collection.
    let members = await this.target();

    let output = members
        .filter(ware => parseInt(ware.bestand) > 0)
        .map(ware => ({
            "Bezeichnung": ware.bezeichnung,
            "Preis": ware.preis
        }));

    response.type("json").send(output);
} , "listMembers" , [] );
```

Beispiel 4 - Ausschnitt aus dem Setup der Aktionen

5.5. Einsparung aus Komplexitätsgründen

Leider musste ich aus Zeitgründen die Implementation und genaue Planung einiger Komponenten hintenan stellen, auch wenn ich sie nie gänzlich vergessen habe.

5.5.1. Constraints

Die Constraints sind eigentlich ein sehr wichtiger Bestandteil der ODRL, da sie erst die Attribut-basierte Autorisierung möglich machen. In einer vollständigen Implementation würden diese im PDP nach dem Erhalten der Policies aufgelöst werden, wo bisher nur die Art der Regel (Permission/Prohibition) analysiert wird. Da Constraints sehr umfangreich sind und ihre Implementation das Modell weder groß verändern noch verbessern, sondern nur die Policy-Auflösung komplizierter machen, habe ich ihre Umsetzung für einen späteren Zeitpunkt nach dieser Arbeit festgelegt. Ich bin der Meinung, dass die Rollenbasierte Autorisierung durch die Collections zur Demonstration dieser Arbeit ausreichend ist.

5.5.2. ConflictTerm

Angenommen mehrere Policies widersprechen sich: die eine erlaubt die Zeiterfassung für alle Mitarbeiter, die andere verbietet aber generell eine Zeiterfassung nach 20 Uhr. Um ein Kriterium festzulegen, ob Permissions oder Prohibitions den Vorrang haben, kann die Angabe eines geeigneten *Conflict Terms* helfen. Es mag sicherlich Umstände geben, wo dies notwendig ist, jedoch habe ich mich der Einfachheit halber für die Intuition entscheiden, dass eine Prohibition immer überwiegt. Bei gegebenem Bedarf sollte sich dieses Feature allerdings relativ einfach integrieren lassen.

5.5.3. Inheritance

Eine naive Vererbung der Regeln einer Policy ließe sich entweder durch eine zusätzliche Zeile in der Abfrage aus der Policy-Datenbank oder eine zusätzliche Verknüpfung einer Policy mit allen geerbten Regeln beim Einlesen realisieren. Auch eine Kopie von geerbten Regeln käme in Betracht. Ich sehe Vererbung in

diesem Stadium der Arbeit noch nicht als wichtig an, da sie erst für größere Projekte eine Rolle spielt. Außerdem können dieselben Regeln einer anderen Policy hinzugefügt und somit eine eventuell notwendige Vererbung umgangen werden.

5.5.4. Obligations

Die Auflösung und Verwaltung der Aktionen im PEP war das letzte, an dem ich gearbeitet habe. In anderen Modellen wird sogar ein zusätzlicher *Policy Execution Point* definiert, um obligatorische Aktionen auszuführen. Der PEP sollte in der Lage sein Obligation, die in einem Response Context definiert werden, selbstständig auszuführen und abzuwarten. Die Entwicklung von Obligations wurde dennoch vernachlässigt, da sie nicht unbedingt notwendig für das Zugriffsmanagement sind. Außerdem ist dafür eine zusätzliche Syntax für den Response Context notwendig, was zum Ende dieser Arbeit aus Zeitgründen nicht möglich war.

5.6. Mögliche Verbesserungen für die Zukunft

5.6.1. Datenflussmodell optimieren

Das Datenflussmodell der XACML war ein guter erster Ansatz, jedoch haben sich durch die strikte Aufteilung der Zuständigkeiten der einzelnen Stationen und das Bestreben einer ebenso getrennten Datenwelt, zu einigen Schwierigkeiten geführt, die durch ein passenderes Datenflussmodell vermieden werden können. Des weiteren verschwimmen durch die Features der aktuellen Implementation ohnehin die Grenzen der Aufgaben, die jeder Teil des Datenflussmodells erfüllen soll, wie zum Beispiel der Umstand, dass der PIP nicht nur Informationen, sondern auch Methoden liefern muss, die dem PEP für die Aktionen zur Verfügung gestellt werden. Eine häufig angewendete Lösung ist die Einführung zusätzlicher Stationen, wie dem Policy Execution Point (PXP) für die tatsächliche Ausführung der Aktionen und dem Policy Repository Point (PRP) für die Speicherung der Policies.
Was den Zusammenhang der Aufteilung von Zuständigkeiten und der Aufteilung von Datenzugriffen betrifft, scheint dies die Verarbeitung nur zu

verkomplizieren, ohne einen nennenswerten Mehrwert zu liefern. Durch die Beschränkung auf Neo4j als einzige Persistenz und ein Zugriffsrecht aus allen Stationen heraus, nur mit einem unterschiedlichen Fokus auf die Daten, erhoffe ich mir eine schnellere Abarbeitung des Requests, bessere Integration der Aktionen und konsistentere Nutzung der Daten. Dies könnte ein notwendiger Kompromiss sein, um genügend Flexibilität zur Umsetzung vollständig integrierter Aktionen zu gewährleisten.

5.6.2. ODRL Profile definieren

Um Entwicklern die Möglichkeit zu geben ihre eigenen Aktionen zu definieren, ist eine Beschreibung der Semantik unerlässlich. Bisher bietet das Framework allerdings nur die Minimalanforderungen an Aktionen, *use* und *transfer*. Zur Manipulation der Datenbank sollten zusätzliche Aktionen definiert werden, wie zum Beispiel das Auflisten, Hinzufügen und Entfernen von Mitglieder einer *Collection* oder das Bearbeiten von Attributen einer *Party* oder eines *Assets*. Die genauen Spezifikationen in einem ODRL-Profile festzuhalten würde nicht nur einen guten Semantik-Beschrieb liefern, sondern auch die Unterstützung für ODRL verdeutlichen.

5.6.3. GraphQL statt CypherQL

GraphQL ist eine allgemeine Query-Sprache zur Abfrage von Daten, allerdings bietet sie in Kombination mit der “neo4j-graphql”[20] Extension eine Alternative zur *Cypher Query Language*. Ob durch einen Wechsel tatsächlich ein Mehrwert erreicht werden kann, muss noch untersucht werden, allerdings verspricht GraphQL eine einfachere Schnittstelle zu Neo4j, dank der an JSON angelehnten Semantik von Abfragen und Antworten. Möglicherweise wird durch diese Strukturierung der Code bei Datenbankzugriffen etwas übersichtlicher als wenn Cypher-Queries verwendet würden, gerade wenn in Zukunft alle Module auf Neo4j zugreifen sollten.

[20] [20] Neo Technology, *Neo4j and GraphQL*

5.6.4. Der PEP im Web

Da wahrscheinlich ein großer Anteil des Policy Enforcements nicht nur durch Maschinen, sondern durch Autorisierung von menschlichen Nutzern entsteht, wäre eine Erweiterung des Frameworks in Richtung Web-Applikationen äußerst sinnvoll. Die Einführung eines Policy Enforcement Points im Browser könnte helfen, eine einfache Anbindung an den Service zu schaffen. Man könnte meinen, es handelt sich nur um einen Proxy zum PEP auf dem Server, allerdings lassen sich somit unter Umständen auch gewisse Sicherheitsrichtlinien nur durch das Design umsetzen, wie zum Beispiel bestimmte Obligationen, die clientseitig ausgeführt werden müssen, oder Umgebungsvariablen aus dem Browser, die direkt in Constraints einfließen können.

5.6.5. Skalierbarkeit und Trust-Management

Unter Skalierbarkeit versteht man im allgemeineren Sinne das Vermögen einer Anwendung ihre Leistung zu steigern, meist wird dies durch eine Verteilung der Aufgaben einer Anwendung auf multiple Systeme erreicht. Auch das Datenflussmodell der XACML ist darauf ausgelegt, skalieren zu können. Die Teilung in *Enforcement*, *Decision*, *Information* und *Administration* führt dazu, dass diese Aufgaben separat voneinander durchgeführt werden können. Eine Verteilung dieser Aufgaben auf multiple Stellen ermöglicht unter anderem Load-Balancing für den PDP, größere Informationslandschaft für den PIP, unterschiedliche Schnittstellen für den PEP und eine Abschirmung des PAP für erhöhte Sicherheit. Innerhalb einer einzigen Node.js-Instanz geht die Skalierbarkeit auf verteilte Systeme leider verloren. Es handelt sich zwar um keine große Schwierigkeit, eine REST Api für verteilte Zugriffe zu erstellen, allerdings sind in sicherheitskritischen Systemen andere Hürden zu überwinden, wie das Trust-Management der Verbindungen, also der Authentifizierung der Gegenstellen (zum Beispiel von PEP zu PDP). Dies ist kein triviales Problem, dennoch sollten Optionen zu Skalierbarkeit und Trust-Management in zukünftigen Implementationen überdacht werden.

6. Resümee

Das verwendete Informationsmodell der ODRL und das Datenflussmodell der XACML haben einen guten Einblick in das Feld der autorisierten Datenverarbeitung gewährt. Für die Entwicklung eines Moduls für das Zugriffsmanagement war die ODRL ein mehr als geeigneter Kandidat. Diese Arbeit hat zwar nicht den Umfang für eine Analyse der ODRL, jedoch schien die Nutzung für Services keine Gefahr für deren Fähigkeiten zu sein, sondern vielmehr einen stabilen Rahmen zu bieten. Es ist schade, dass im Node Package Manager keine Implementation der ODRL zu finden ist, was womöglich mit der Schwierigkeit zusammenhängt, die solch eine Implementation darstellt. Das Datenflussmodell der XACML auf anderer Seite hat teilweise das Gefühl vermittelt, durch die Semantik der Verarbeitung eingeschränkt zu werden, auch wenn es für den Anfang ein guter Start war. Definitiv kann man aber aus der Verarbeitung eines Requests lernen um ein besseres Datenflussmodell zu finden.

Ein Grund für die Schwierigkeiten mit der XACML liegt womöglich an der objektorientierten Herangehensweise. Ohne voreilige Schlüsse ziehen zu wollen, könnte eine prozedurale Herangehensweise einfacher sein, um mit dem Modell umzugehen. Das merkt man vor allem daran, dass die Klassen wenig zu nutzen scheinen, wenn man sich vorstellt in einer verteilten Datenlandschaft zu agieren, und mehr dazu dienen, die wenigen und eher aufwändigen Funktionsaufrufe zu gruppieren. Das ewige Hin-und-Her-Überlegen über die Herangehensweise war auch ein großer Faktor, der die Entwicklung einer Implementation verlangsamt hat.

Der Wunsch ein Node.js Package zu programmieren hat sich leider nicht erfüllt, weil bei näherer Betrachtung das Setup des Policy Agenten zu aufwändig ist und automatisiert werden müsste, damit andere Nutzer es gebrauchen könnten. Mit dem Ergebnis dieser Arbeit kann ich allerdings dennoch sehr zufrieden sein, da sie viele der geforderten Bedingungen erfolgreich umsetzt. Das Zugriffsmanagement auf Service-Aktionen, wie *readFile* oder *auflisten* von Waren, und die Autorisierung bezüglich Ressource und Subjekt funktioniert genau wie es in den Policies festgelegt. Auch die autonome Ausführung dieser

Service-Aktionen, inklusive ihrer Abhängigkeiten, funktioniert einwandfrei, auch wenn der Support für einige Features noch eingeschränkt ist.

Wenn es darum geht einen Service unter Node.js zu schreiben, nimmt ein Policy Agent, wie in dieser Arbeit vorgestellt, einen Großteil des Programmier- und Verwaltungsaufwandes ab. Da sich der Quellcode einer möglichen Anwendung der ODRL und den Mechanismen des Agenten unterwerfen muss, wird dieser automatisch strukturierter und leichter zu warten. Für mich ist das ein guter Grund, die Entwicklung einer verbesserten Version voranzutreiben und den Policy Agenten in zukünftigen Service-Anwendungen einzusetzen.

7. Literaturverzeichnis

[1] W3C: *ODRL Information Model 2.2 - W3C Recommendation 15 February 2018*, <https://www.w3.org/TR/odrl-model/>, 01.04.2019

[2] W3C: *ODRL Vocabulary & Expression 2.2 - W3C Recommendation 15 February 2018*, <https://www.w3.org/TR/2018/REC-odrl-vocab-20180215/>, 01.04.2019

[3] OASIS: *eXtensible Access Control Markup Language (XACML) Version 3.0 - OASIS Standard 22 January 2013*, <http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html>, 01.04.2019

[4] Srijith Nair, Axiomatics: *XACML Reference Architecture*, <https://www.axiomatics.com/blog/xacml-reference-architecture/>, 01.04.2019

[5] WSO2: *Access Control and Entitlement Management*, <https://docs.wso2.com/display/IS570/Access+Control+and+Entitlement+Management>, 01.04.2019

[6] Fraunhofer IESE: *IND²UCE Sicherheitsframework für Datennutzungskontrolle*, <https://www.iese.fraunhofer.de/de/competencies/security/ind2uce-framework.html>, 01.04.2019

[7] Node.js Foundation: *Über Node.js®*, <https://nodejs.org/de/about/>, 15.04.2019

[8] Neo Technology: *The Internet-Scale Graph Platform*, <https://neo4j.com/product/>, 01.04.2019

[9] MongoDB, Inc.: *What is MongoDB?*, <https://www.mongodb.com/what-is-mongodb>, 01.04.2019

[10] OASIS: *Open Digital Rights Language (ODRL)*, <http://xml.coverpages.org/odrl.html>, 03.04.2019

[11] IPTC Rights Expressions Working Group: *IPTC RightsML Standard 2.0*, <https://iptc.org/std/RightsML/2.0/RightsML_2.0-specification.html>, 03.04.2019

[12] Kasten, Andreas & Grimm, Rüdiger: *Making the Semantics of ODRL and URM Explicit Using Web Ontologies*, <https://www.researchgate.net/publication/268434167_Making_the_Semantics_of_ODRL_and_URM_Explicit_Using_Web_Ontologies>, 03.04.2019

[13] OASIS: *XML and Digital Rights Management (DRM)*, <http://xml.coverpages.org/drm.html>, 03.04.2019

[14] Renato Iannella, W3C: *Open Digital Rights Language (ODRL)*, <https://www.w3.org/2012/09/odrl/archive/odrl.net/ODRL-07.pdf>, 03.04.2019

[15] WSO2: *WSO2 Balana Implementation*, <https://svn.wso2.org/repos/wso2/trunk/commons/balana/>, 11.04.2019

[16] centron, 04.07.2013: *Was ist Node.js?*, <https://www.centron.de/was-ist-node-js/>, 15.04.2019

[17] Dr. Holger Schwichtenberg, Martin Möllenbeck, 31.03.2014: *JavaScript überall – Webserverprogrammierung mit Node.js*, <https://entwickler.de/online/javascript-ueberall-webserverprogrammierung-mit-node-js-159630.html>, 16.04.2019

[18] Stefan Luber, Nico Litzel, 10.01.2019: *Was ist eine Graphdatenbank?*, <https://www.bigdata-insider.de/was-ist-eine-graphdatenbank-a-788834/>, 16.04.2019

[19] Renato Iannella, 15.02.2018: *ODRL: a path well travelled*, <https://www.w3.org/blog/2018/02/odrl-a-path-well-travelled/>, 30.04.2019

[20] Neo Technology: *Neo4j and GraphQL*, <https://neo4j.com/developer/graphql/>, 19.05.2019

[21] Facebook Inc.: *Introduction to GraphQL*, <https://graphql.org/learn/>, 19.05.2019

[22] Grigor Khachatryan, 16.06.2018: *What is GraphQL?*, <https://medium.com/devgorilla/what-is-graphql-f0902a959e4>, 19.05.2019

[23] Yoon Jae Kim, CSE at Washington University in St. Louis: *Access Control Service Oriented Architecture Security*, <https://www.cse.wustl.edu/~jain/cse571-09/ftp/soa/index.html>, 20.05.2019

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit über

Policy-basiertes Zugriffs-Management für Webservices unter Node.js mit Bezug auf ODRL und XACML

selbstständig verfasst habe, dass ich keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe und dass die Stellen der Arbeit, die anderen Werken – auch elektronischen Medien – dem Wortlaut oder Sinn nach entnommen wurden, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht worden sind.

_____ _____

Ort, Datum Unterschrift