hypercontract

Selbstbeschreibende REST APIs für Mensch und Maschine

MASTERTHESIS

Marvin Luchs

REST APIs must be hypertext-driven

— Fielding 2008

Abstract

Die Interaktion mit einer REST-basierten Web-API ähnelt der Nutzung einer Website durch den Menschen: Eine URL identifiziert den Einstiegspunkt und typisierte Hyperlinks in den Nachrichten des Servers verweisen auf weiterführende Ressourcen und somit auf die möglichen nächsten Schritte. Damit dies funktioniert, muss der menschliche Betrachter die Website im fachlichen Kontext interpretieren. In der Machine-to-Machine-Kommunikation wird der Kontext über die Aushandlung von Media Types und Profilen vermittelt. Dadurch wird dem Konsumenten einer REST-API das Verständnis für die Funktionsweise und Semantik der Schnittstelle vermittelt. Die Spezifikation von Media Types und Profilen erfolgt jedoch meist in Form entkoppelter, für Menschen verfasster Dokumentation, die für die maschinelle Verarbeitung nicht geeignet ist. Andere populäre Ansätze zur API-Beschreibung sind wiederum nicht mit den Prinzipien von REST vereinbar.

Mit hypercontract wird in dieser Arbeit ein Konzept für die Beschreibung REST-konformer Web-APIs vorgestellt, das im Einklang mit der Architektur des Webs und den Prinzipien von REST steht und die Schnittstelle maschinenlesbar sowie für den Menschen intuitiv verständlich im Kontext der Nutzung dokumentiert. Illustriert wird dies anhand einer für diesen Zweck entwickelten, beispielhaften Anwendung. Ausgehend vom fachlichen Kontext der Anwendung wird die applikationsspezifische Schnittstellenmechanik und fachliche Semantik mit Hilfe des Vokabular von hypercontract in Form eines RDF-basierten Profils definiert. Die Bereitstellung des Profils als RDF-Serialisierung und HTML-Dokument erlaubt eine empfängergerechte Aufbereitung. Die eindeutige Identifikation von Nachrichtenelementen über URIs ermöglicht zudem den direkten Zugriff auf relevante Abschnitte der Spezifikation.

Die abschließende Evaluation zeigt die Vorteile auf, die sich durch den Einsatz von RDF in Hinblick auf die Vereinbarkeit mit der Webarchitektur und REST ergeben, betont jedoch auch, dass die Abbildung fachlicher Semantik in maschinenlesbarer Form auch weiterhin eine Herausforderung darstellt.

Schlagworte: REST, Web-APIs, Schnittstellenspezifikation, RDF

Inhaltsverzeichnis

ABBILDUNGSVERZEICHNIS IV

TABELLENVERZEICHNIS V

LISTINGVERZEICHNIS VI

1 EINLEITUNG 1

1.1 Problembeschreibung 3

1.2 Zielsetzung 4

1.3 Abgrenzung 5

1.4 Vorgehensweise 5

2 GRUNDLEGENDE KONZEPTE UND TECHNOLOGIEN 7

2.1 Architekturprinzipien und Konzepte des Webs 7

2.1.1 Identifikation durch URIs 9

2.1.2 Interaktion via HTTP 9

2.1.3 Datenformate und Hypermedia 11

2.2 Webservices 13

2.2.1 XML-RPC und SOAP 13

2.2.2 Webservice-Architektur 14

2.2.3 Architekturmodelle und Konzepte 15

2.2.4 Contracts in Webservices 17

2.3 REST als Architekturstil 19

2.3.1 Prinzipien 20

2.3.2 Uniform Interface 21

2.3.3 Contracts in REST 25

2.4 RDF und Linked Data 33

2.4.1 Ressourcenbeschreibung mit RDF 35

2.4.2 Serialisierung von RDF-Daten 37

2.4.3 Vokabulare mit RDFS und OWL 39

2.4.4 REST und Linked Data 41

3 EXISTIERENDE LÖSUNGEN 44

3.1 OpenAPI 44

3.2 hRESTS 48

3.3 ALPS 52

3.4 Hydra 55

3.5 Zusammenfassung 61

4 HYPERCONTRACT 63
4.1 Anwendungsbeispiel: hypershop ... 63
4.1.1 Fachlicher Kontext ... 64
4.1.2 Semantische Deskriptoren ... 66
4.1.3 Zustandsübergänge ... 67
4.1.4 Einheitliches Vokabular ... 69
4.1.5 Repräsentationsformate ... 69
4.2 Profildefinition mit hypercontract ... 75
4.2.1 RDF als Beschreibungssprache ... 76
4.2.2 Schnittstellenmechanik ... 76
4.2.3 Fachliche Schnittstellensemantik ... 85
4.3 Integration in die Schnittstelle ... 86
4.3.1 Bereitstellung des Profils ... 87
4.3.2 Verknüpfung mit der Schnittstelle ... 89
4.3.3 Semantische Annotation von HTML-Elementen ... 92
4.3.4 Aushandlung des Contracts ... 93
5 EVALUATION 95
5.1 Vereinbarkeit mit Webarchitektur ... 95
5.2 Vereinbarkeit mit REST ... 96
5.3 Integration in REST-konforme Web-APIs ... 97
5.4 Verfügbarkeit im Kontext der Nutzung ... 98
5.5 Aufbereitung für Mensch und Maschine ... 99
5.6 Beschreibung der Schnittstellenmechanik ... 99
5.7 Beschreibung der fachlichen Semantik ... 100
6 ZUSAMMENFASSUNG UND AUSBLICK 101
A LITERATURVERZEICHNIS XI
B NAMENSRAUMPRÄFIXE XIX
C SEMANTISCHE DESKRIPTOREN VON HYPERSHOP XX
D ZUSTANDSÜBERGÄNGE VON HYPERSHOP XXIV
E HYPERCONTRACT VOKABULAR XXVI

Abbildungsverzeichnis

Abbildung 1: Architekturmodelle der Web Services Architecture nach BOOTH U. A. 2004 15
Abbildung 2: Contracts als Komposition aus Media Types und Protokollen nach WEBBER, PARASTATIDIS, ROBINSON 2010, S. 109 26
Abbildung 3: Die Spezifikationen von HTML 2.0 und 5.3 im Internet Explorer 3 und Edge 79 29
Abbildung 4: Der ursprüngliche Semantic Web Stack nach BERNERS-LEE 2000 und der erweiterte nach BRATT 2007 33
Abbildung 5: Darstellung eines RDF-Graphen 37
Abbildung 6: Darstellung eines RDFS-basierten Vokabulars 40
Abbildung 7: Zuordnung von Ressourcen zu Klassen über `rdf:type` 40
Abbildung 8: Definition von Eigenschaften mit OWL 41
Abbildung 9: Beispiel für die aus einer OpenAPI-Spezifikation generierte Dokumentation 46
Abbildung 10: Der in hypershop abgebildeten Aktivitäten 64
Abbildung 11: Das semantische Datenmodell von hypershop 65
Abbildung 12: Zustandsdiagramm der hypershop REST-API 68
Abbildung 13: Darstellung der HTML-Repräsentation eines Produkts im Webbrowser 70
Abbildung 14: Die Produkt-Ressource als RDF-Graph 74
Abbildung 15: Die `Product`-Klasse und ihrer Eigenschaften als RDF-Graph 76
Abbildung 16: Die `Product`- und die `ShoppingCartItem`-Klasse mit gemeinsamen Eigenschaften 77
Abbildung 17: Definition von Eigenschaften am Beispiel von Warenkorb und Warenkorbposition 78
Abbildung 18: Darstellung des `product`-Links von der Warenkorbposition auf das Produkt als RDF-Graph 79
Abbildung 19: Beschreibung der `addToShoppingCart`-Operation mit Hilfe des hypercontract-Vokabulars 80
Abbildung 20: Beschreibung der `searchCatalog`-Operation mit Hilfe des hypercontract-Vokabulars 81
Abbildung 21: Beschreibung der Einstiegsressource der hypershop-API als `EntryPoint` 81
Abbildung 22: Verknüpfung der `quantity`-Eigenschaft mit einem JSON-Schema 82
Abbildung 23: Verknüpfung der `AdditionToShoppingCart`-Klasse mit einem JSON-Schema 83
Abbildung 24: Verknüpfung der `SearchQuery`-Klasse mit einem URI-Template 84
Abbildung 25: Beschreibung der `AdditionToShoppingCart`-Klasse und ihrer Eigenschaften mit `rdfs:comment` 85
Abbildung 26: Mehrsprachige Beschreibung der `AdditionToShoppingCart`-Klasse 85
Abbildung 27: Abbildung der Vorbedingung für das Bestellen eines Warenkorbs als `hyper:Precondition` 86
Abbildung 28: Darstellung des hypershop-Profils als HTML-Seite 88

Tabellenverzeichnis

Tabelle 1: Beispiel für die Beschreibung semantischer Deskriptoren ... 66
Tabelle 2: Beispiel für die Beschreibung semantischer Deskriptoren mit Definitions- und Wertebereichen für Eigenschaften ... 67

Listingverzeichnis

Listing 1: Clientanfrage und Serverantwort mit Aushandlung des Media Types via Content Negotiation 11
Listing 2: Angabe eines Link-Relationstyps für einen HTML-Verweis 12
Listing 3: Verweis auf das Profil über einen `Link`-Header 31
Listing 4: Verweis auf das Profil über den `profile`-Parameter des JSON-HAL Media Types 32
Listing 5: Berücksichtigung des Profils bei der Content Negotiation über den `Accept-Profile`-Header 32
Listing 6: Beispiele für RDF-Statements 35
Listing 7: Verwendung von Namensraumpräfixen zur kompakteren Darstellung von URIs in RDF-Tripeln 36
Listing 8: RDF-Statements über eine Eigenschaft 36
Listing 9: Abbildung von Daten als JSON-Struktur 37
Listing 10: Abbildung eines RDF-Graphen im JSON-LD-Format 38
Listing 11: Beispiel für eine Schnittstellenbeschreibung mit OpenAPI 46
Listing 12: Annotation einer HTML-basierten API-Dokumentation mit hRESTS 49
Listing 13: SAWSDL-Mapping eines Nachrichtenteils auf eine applikationsspezifische Ontologie 50
Listing 14: ALPS-Beschreibung für eine Produktrepräsentation 53
Listing 15: Beispiel für eine Schnittstellenbeschreibung mit Hydra (verkürzt) 56
Listing 16: Beispiel für die Repräsentation einer Produkt-Ressource in einer mit Hydra beschriebenen Web-API 56
Listing 17: Beispiel für die Beschreibung von Eigenschaften einer Hydra-Klasse (verkürzt) 57
Listing 18: Beispiel für die Repräsentation einer Sammlung von Product-Ressourcen als Hydra-Collection (verkürzt) 58
Listing 19: Clientanfrage (verkürzt) für die Abfrage eines Produkts als HTML-Repräsentation 69
Listing 20: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und HTML-Repräsentation 70
Listing 21: Clientanfrage für die Abfrage eines Produkts als JSON-Repräsentation 71
Listing 22: Antwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-Repräsentation 71
Listing 23: Clientanfrage für die Abfrage eines Produkts als JSON-HAL-Repräsentation 72
Listing 24: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-HAL-Repräsentation 72
Listing 25: Clientanfrage für die Abfrage eines Produkts als JSON-LD-Repräsentation 73
Listing 26: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-LD-Repräsentation 74
Listing 27: Serverantwort auf eine Produkt-Anfrage mit Verweis auf einen JSON-LD-Kontext via Link-Header 75
Listing 28: JSON-HAL-Repräsentation einer Warenkorbposition mit Link auf das Produkt 79

Listing 29: Clientanfrage für das Hinzufügen eines Produkts zum Warenkorb 80
Listing 30: Serverantwort auf das Anlegen einer Warenkorbposition mit Umleitung auf den Warenkorb 80
Listing 31: Schemadefinition für die `quantity`-Eigenschaft als JSON Schema 83
Listing 32: Schemadefinition für die `AdditionToShoppingCart`-Klasse als JSON Schema 84
Listing 33: Serverantwort (verkürzt) auf eine Abfrage des hypershop-Profils als JSON-LD 87
Listing 34: Clientanfrage und Serverantwort (verkürzt) für das JSON-Schema der `AdditionToShoppingCart`-Klasse 89
Listing 35: Serverantwort (verkürzt) mit `Link`-Header für das Profil 90
Listing 36: Serverantwort (verkürzt) mit `Link`-Header für Profil und JSON-LD-Kontext 90
Listing 37: Clientanfrage und Serverantwort (verkürzt) für die JSON-LD-Repräsentation der `AddToShoppingCart`-Operation 91
Listing 38: Clientanfrage und Serverantwort (verkürzt) für die HTML-Repräsentation der `AddToShoppingCart`-Operation 92
Listing 39: HTML-Repräsentation eines Produkts mit Link-Relationstypen und Microdata-Auszeichnungen 93
Listing 40: Clientanfrage und Serverantwort (verkürzt) mit Aushandlung von Media Type und Profil via Content Negotiation 94

1 Einleitung

Man stelle sich folgende Situation vor: Ein Nutzer möchte bei einem Internethändler ein Produkt bestellen. Da er den Bestellprozess des Anbieters nicht kennt, muss er im Handbuch des Online-Shops zunächst nachschlagen, wie der URL für eine Suchanfrage an den Katalog zusammengesetzt wird. Als Antwort erhält er eine Liste mit Suchergebnissen. Er notiert sich die ID des gewünschten Produkts und schlägt nun an anderer Stelle im Handbuch nach, wie er den Artikel seinem Warenkorb hinzufügen kann. Er erfährt von einem weiteren URL, an welchen er die Produkt-ID und die gewünschte Bestellmenge als POST-Anfrage sendet. Auf diese Weise arbeitet er sich Schritt für Schritt durch den Prozess, merkt sich die IDs von Adressen und Zahlungsoptionen, kombiniert sie mit den IDs der Warenkorbpositionen, fügt alle Informationen nach den Vorgaben des Handbuchs zusammen, sendet sie an den URL für neue Bestellungen – und erhält als Antwort ein Fehler: `422 Unprocessable Entity`. Ein erneuter Blick in die Dokumentation verrät: Mindestbestellwert unterschritten.

Wäre das die gängige Benutzererfahrung im Web, hätte der deutsche Internethandel im vergangenen Jahr keine 53 Milliarden Euro Umsatz gemacht (vgl. STATISTA 2019). Trotzdem ist diese Beschreibung nicht völlig abwegig. Bei der Anbindung von Webschnittstellen machen Softwareentwickler häufig ähnliche Erfahrungen.

Grundsätzlich sind die primären Konsumenten einer Schnittstelle andere Programme. Bei der Festlegung der Qualitätsziele genießt die Gebrauchstauglichkeit für den Menschen daher selten eine hohe Priorität. Tatsächlich werden Schnittstellen aber für Menschen geschrieben. Insbesondere bei öffentlichen APIs sind sie der primäre Kommunikationskanal zwischen den Entwicklern, die die Schnittstelle implementieren, und jenen, die sie anbinden (vgl. SPICHALE 2019, S. viii–ix). Ist eine Schnittstelle nicht verständlich gestaltet, erhöht sich das Risiko, dass die Nutzer unnötig viel Zeit für die Integration aufwenden müssen, die API falsch verwenden oder sich gleich für eine Alternative entscheiden. Daher ist, neben der Zuverlässigkeit einer Schnittstelle, die Gebrauchstauglichkeit ein wesentliches Qualitätsmerkmal. Ist die Schnittstelle intuitiv verständlich, leistet dies einen wichtigen Beitrag zu ihrer Gebrauchstauglichkeit (vgl. ebd., S. 13f.).

Im Kontext von Web-APIs mangelt es häufig an dieser intuitiven Verständlichkeit. Stattdessen verlässt man sich auf umfangreiche API-Dokumentationen, um die Usability der Schnittstelle zu gewährleisten (vgl. RICHARDSON, AMUNDSEN 2013, S. 134). Grundsätzlich ist eine gute API-Dokumentation wichtig und trägt nicht

unerheblich zur Gebrauchstauglichkeit einer Schnittstelle bei. Sie spielt ihre Stärken vor allem dann aus, wenn es darum geht, einen Überblick zu bekommen und grundlegende Prinzipien und Ideen zu vermitteln. In der Praxis wählen Entwickler jedoch beim Erlernen einer neuen API häufig eine explorative Herangehensweise. Eine von der Schnittstelle getrennte Dokumentation hilft hier nur bedingt. Sie stellt eine statische Abbildung dar, die nicht zwingend das widerspiegelt, was dem Entwickler in der Praxis begegnet. Darüber hinaus bedeutet sie für den Leser einen kostspieligen Kontextwechsel, da er die für ihn in dem Moment relevanten Teile ausfindig machen muss. Daher sollte die Schnittstelle selbst dem Nutzer alle Informationen zur Verfügung stellen, die für ihr Verständnis notwendig sind – ähnlich dem Besuch einer Website:

> *Lots of websites have help docs, but when was the last time you used them? Unless there's a serious problem (you bought something and it was never delivered), it's easier to click around and figure out how the site works by exploring the connected, self-descriptive HTML documents it sends you.*
>
> — RICHARDSON, AMUNDSEN 2013, S. 15

Es liegt also nahe, Web-APIs nach ähnlichen Prinzipien zu gestalten, wie sie bei Websites zum Einsatz kommen. Der im Umfeld von Webapplikationen populäre Architekturstil REST bietet hierfür die ideale Grundlage, da er die Architekturprinzipien des Webs generalisiert und so auf andere verteilte Systeme anwendbar macht (vgl. FIELDING 2000).

Die Nutzung einer REST-basierten Web-API erfolgt demnach ähnlich der einer Website. Der Konsument kennt nur den Einstiegs-URL. Typisierte Hyperlinks in den Nachrichten des Servers verweisen mit Hilfe von URIs auf weitere Ressourcen der API und informieren den Nutzer über die möglichen nächsten Schritte (vgl. TILKOV U. A. 2015, S. 71ff.). Durch die Angabe des Media Types in den Kopfzeilen der Nachrichten einigen sich Client und Server auf ein gemeinsames Nachrichtenformat. Damit sind die Nachrichten selbstbeschreibend und können von den beteiligten Systemen verarbeitet werden, ohne zusätzliche Informationen heranzuziehen (vgl. ebd., S. 158). HTTP als Applikationsprotokoll liefert die notwendigen Methoden, um durch den Austausch von Nachrichten API-Ressourcen abzufragen und zu manipulieren (vgl. ebd., S. 53ff.).

URIs, HTTP, selbstbeschreibenden Nachrichten und Hyperlinks bilden das so genannte *Uniform Interface* von webbasierten REST-APIs. Die Vorteile in Hinblick auf

Interoperabilität und Evolvierbarkeit verdankt der Architekturstil unter anderem dieser einheitlichen Schnittstelle (vgl. WILDE, PAUTASSO 2011, S. 2ff.). Ein Client, der die Interaktion mit Ressourcen über HTTP unterstützt und in der Lage ist, via Links zu weiteren Ressourcen zu navigieren, ist zu jeder REST-API kompatibel, sofern sich die beteiligten Systeme auf einen Media Type einigen können.

1.1 Problembeschreibung

Der Media Type ist ein zentraler Aspekt des Contracts, den Anbieter und Konsumenten einer Webschnittstelle eingehen. Über dessen Spezifikation wird nicht nur der Aufbau von Nachrichten definiert, sondern auch die fachliche Bedeutung der darin enthaltenen Auszeichnungselemente und Linktypen beschrieben. Die Aushandlung des Media Types ist somit ein wichtiger Mechanismus, um sicherzustellen, dass die beteiligten Systeme ein gemeinsames Verständnis für den Kontext besitzen, in dem sie sich bewegen. Es gibt jedoch keine deklarative Notation, mit welcher die verschiedenen Aspekte eines Media Types festgehalten werden können. Sie werden daher meist in Form separater Datenschemas und Spezifikationsdokumente beschrieben (vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, S. 108).

Etablierte Standards wie HTML, XML oder JSON sind zwar umfassend spezifiziert, allerdings ist es rein auf Basis dieser Formate nicht möglich, die Fachlichkeit einer Applikation zu vermitteln. Im Falle einer Website übersetzt der Server die fachliche Semantik eines Online-Shops in die dokumentenzentrierte Semantik von HTML. Der Informationsverlust, der dabei entsteht, wird durch den menschlichen Betrachter ausgeglichen, der in der Lage ist, die Inhalte des Webdokuments mit Hilfe von Kontextinformationen zu interpretieren und so die die semantische Lücke zu überbrücken (vgl. RICHARDSON, AMUNDSEN 2013, S. 16). Die mit Schnittstellen ausgetauschten XML- oder JSON-Nachrichten werden jedoch nicht von Menschen konsumiert und weisen für sich genommen keinerlei eigene Semantik auf. Eine Interpretation von Nachrichten in diesen Formaten ist ohne zusätzliche Informationen daher nicht möglich (vgl. ALLEN 2016).

Die Definition eines applikationsspezifischen Media Types bringt andere Nachteile mit sich. Sie erfordert eine aufwändige Registrierung bei der IANA (vgl. FREED, KLENSIN, HANSEN 2013). Damit geht nicht nur ein administrativer Aufwand einher, sondern erschwert auch die zukünftige Weiterentwicklung, da Änderungen an der Spezifikation des Media Type ebenfalls an die IANA kommuniziert werden müssen. Die Einführung eigener Standards fördert zudem Insellösungen, die sich nachteilig auf die Interoperabilität des Systems auswirken.

Ein möglicher Ausweg aus dieser Problematik sind Profile (vgl. WILDE 2013). Mit ihnen kann ein existierender Media Type um zusätzliche Semantik ergänzt werden. Aber die Spezifikation von Profilen erfolgt, wie auch bei Media Types, üblicherweise in Form von separaten, menschenlesbaren Dokumenten.

Existierende API-Beschreibungstechnologien könnten als Ausgangspunkt für die Spezifikation von Profilen dienen. Viele sind jedoch nicht mit den Prinzipien von REST vereinbar. Andere Ansätze lassen sich nur mit vordefinierten Nachrichtenformaten nutzen oder weisen Schwächen in der maschinenlesbaren Beschreibung von REST-Schnittstellen auf (siehe *3 Existierende Lösungen*).

Somit präsentiert, sich sowohl aus menschlicher Sicht als auch im Hinblick auf die maschinelle Verarbeitung, eine unbefriedigende Situation. Die Entkopplung von API und Dokumentation erschwert für beide Seiten den Zugang zur Schnittstellenbeschreibung. Für den Entwickler bedeutet dies, dass ein exploratives Vorgehen bei der Anbindung der Schnittstelle behindert wird, da er im Kontext der Nutzung nicht die Informationen erhält, die er für das Verständnis benötigt. Aus technischer Perspektive fehlt eine geeignete Lösung für die maschinenlesbare Beschreibung von REST-Schnittstellen, die den Architekturprinzipien von REST und dem Web gerecht wird. Für viele Entwicklungswerkzeuge ist eine solche aber unverzichtbar. Sie ist die Grundlage für automatisierte Tests, Generierung von Dokumentation und Code oder für die Validierung von Client-Implementierungen.

1.2 Zielsetzung

Das Ziel der vorliegenden Arbeit ist die Entwicklung eines Konzepts für die Beschreibung von REST-konformen Webschnittstellen. Im Ergebnis soll eine Lösung entstehen, welche

(1) auf den Architekturprinzipien des Webs basiert und den Anforderungen des REST-Architekturstils gerecht wird,
(2) eine einfache Integration in existierende, REST-konforme Web-APIs erlaubt,
(3) die API im Kontext ihrer Nutzung beschreibt und somit zur Laufzeit abrufbar ist,
(4) sowohl für die Nutzung durch den Menschen als auch für eine maschinelle Auswertung geeignet ist und
(5) neben einer Beschreibung der Schnittstellenmechanik auch die fachliche Semantik definiert.

Validiert wird dieses Konzept über eine prototypische Implementierung.

1.3 Abgrenzung

Trotz der eindeutigen Definition des REST-Architekturstils durch Fielding 2000 wird die Bezeichnung REST an vielen Stellen missbräuchlich für verschiedene Formen HTTP-basierter Schnittstellen genutzt. Im Kontext der vorliegenden Arbeit sind mit REST-Schnittstellen jedoch nur solche gemeint, die den Kriterien des Architekturstils entsprechen.

Die Beschreibung solcher Schnittstellen umfasst häufig nicht nur die hier betrachteten mechanischen und semantischen Aspekte, sondern auch nichtfunktionale Eigenschaften wie Verfügbarkeitsgarantien und SLAs, Zugriffsbeschränkungen, Nutzungsbedingungen oder Preisinformationen (vgl. Lane 2014). Eine zusätzliche Behandlung dieser Aspekte würde jedoch den Rahmen dieser Arbeit überschreiten. Dennoch ist denkbar, dass eine zukünftige Weiterentwicklung des hier vorgestellten Konzepts auch diese Themen berücksichtigt.

1.4 Vorgehensweise

Einleitend wurden die Motivation und die Zielsetzung dieser Masterthesis vorgestellt. Der weitere Verlauf der Arbeit gliedert sich wie folgt:

Im Rahmen von *2 Grundlegende Konzepte und Technologien* werden die theoretischen Grundlagen erläutert sowie relevante Standards und Lösungen in diesem Kontext vorgestellt. Den Ausgangspunkt bilden die wesentlichen Konzepte und Architekturprinzipien des Webs. Darauf aufbauend erfolgt eine Betrachtung der Webservices-Architektur des W3C, anhand derer das Konzept von Contracts für Schnittstellen in verteilten Systemen besprochen wird. Nachfolgend wird der Architekturstil REST vorgestellt und das Verständnis von Contracts in diesem Kontext erörtert. Schließlich wird mit RDF eine Technologie aus dem Umfeld von Linked Data behandelt, deren Einsatz für die Beschreibung von Schnittstellen im weiteren Verlauf der Arbeit von zentraler Bedeutung ist.

Im nächsten Schritt betrachtet *3 Existierende Lösungen* vier Ansätze für die Dokumentation von Schnittstellen. Mit OpenAPI, hRESTS, ALPS und Hydra wurden hierfür Projekte mit unterschiedlichen Schwerpunkten in der Herangehensweise gewählt. Jeder dieser Ansätze wird anhand der Zielsetzung dieser Arbeit bewertet, mit dem Zweck die identifizierten Vor- und Nachteile bei der Konzeption von hypercontract zu berücksichtigen.

Aufbauend auf diesen Erkenntnissen erarbeitet *4 hypercontract* ein Lösungskonzept für die Problemstellung dieser Arbeit. Hierfür wird zunächst eine

Beispielanwendung vorgestellt, deren Schnittstelle im Rahmen dieser Arbeit als Versuchsobjekt dient. Anschließend erfolgt die schrittweise Erstellung eines Profils, welches die technischen und fachlichen Aspekte der API dokumentiert. Der letzte Teil betrachtet die Integration dieses Profils in die Schnittstelle und dessen Bereitstellung für den Konsumenten.

Das erarbeitete Konzept wird in 5 *Evaluation* – analog zur Betrachtung existierender Lösungen in Kapitel 3 – anhand der eingangs formulierten Zielsetzung evaluiert.

Den Abschluss bildet 6 *Zusammenfassung und Ausblick* mit einem Überblick über die Ergebnisse der Ausarbeitung und möglicher Ansatzpunkte für die zukünftige Weiterentwicklung des Konzepts.

2 Grundlegende Konzepte und Technologien

Die Diskussion über Contracts im Web ist nicht ohne ein Verständnis für die grundlegenden Konzepte und Technologien des Webs und die Historie rund um Webservices als Vorgänger des REST-Architekturstils möglich. Erst auf diesem theoretischen Fundament ist ein Verständnis für die Eigenschaften von REST und der dahinterstehenden Motivation möglich, um schließlich zum Kern des Problems zu gelangen: Contracts für REST-konforme Schnittstellen. Mit dem Resource Description Framework wird schließlich eine Technologie vorgestellt, die im Kontext des Semantik Webs und der daraus hervorgegangenen Linked-Data-Bewegung eine zentrale Rolle spielt und aufgrund der konzeptionellen Nähe zum Web ein naheliegender Kandidat für die Beschreibung von Webschnittstellen ist.

2.1 Architekturprinzipien und Konzepte des Webs

Die im Folgenden beschriebenen Designprinzipien für die Architektur verfolgen primär zwei Ziele, die für alle Technologien des W3C gelten: *Interoperabilität* und *Evolvierbarkeit* (vgl. BERNERS-LEE 1998a).

Interoperabilität basiert auf gemeinsamen Schnittstellen, ohne ein vollständiges Verständnis für die Implementierung des Schnittstellenpartners besitzen zu müssen. Dies erlaubt bis zu einem gewissen Grad die Weiterentwicklung der Technologie ohne die Interoperabilität mit anderen Technologien zu gefährden.

Die Evolvierbarkeit kommt ins Spiel, wenn die Weiterentwicklung auch Änderungen an der Schnittstelle notwendig macht. Hier wirken zwei Kräfte gegeneinander: Strebt man nach vollständiger Interoperabilität, schränkt dies die Evolvierbarkeit der Schnittstelle stark ein. Auf der anderen Seite ist eine Gewährleistung der Interoperabilität praktisch unmöglich, wenn eine uneingeschränkte Evolvierbarkeit gegeben ist (vgl. FRYSTYK NIELSEN 1999).

Bei der Weiterentwicklung des World Wide Webs müssen diese beiden Aspekte somit stets gegeneinander abgewogen werden. Die von BERNERS-LEE 2013 beschriebenen Designprinzipien und der Aspekt der *Erweiterbarkeit* (vgl. BERNERS-LEE 1998a) zielen darauf ab, diesen Architekturzielen gerecht zu werden.

Erweiterbarkeit und das Designprinzip der *Toleranz* stehen in einem engen Verhältnis zueinander. Die Evolution einer Lösung durch Erweiterung ermöglicht die Aufrechterhaltung der Interoperabilität unter der Voraussetzung, dass der Nutzer dieser Lösung tolerant gegenüber Elementen ist, die er nicht versteht. Damit eng verwandt ist das Design Pattern des *Tolerant Readers* (vgl. DAIGNEAU 2012, 243ff.). Ein

gutes Beispiel hierfür sind klassische Webtechnologien wie HTML oder CSS, die permanent weiterentwickelt werden, ohne in großem Maßstab inoperabel zu älteren Browsern zu werden. Stößt der Client auf ein Element oder eine Eigenschaft, die er nicht versteht, wird sie ignoriert (vgl. BERNERS-LEE 2013; CARPENTER 1996).

Das Designprinzip der *Einfachheit* versucht die Notwendigkeit für eine Weiterentwicklung von vornherein zu vermeiden. Dabei ist Einfachheit nicht das gleiche wie „einfach zu verstehen“, sondern im Sinne einer simplen Lösung zu versehen: Eine Sprache die weniger Grundelemente verwendet und trotzdem eine gleichbleibende Aussagekraft besitzt ist simpler und bietet weniger Angriffsfläche für die Herausforderungen in Hinblick auf Interoperabilität und Evolvierbarkeit (vgl. BERNERS-LEE 2013).

In eine ähnliche Richtung geht das Prinzip der *Rule of Least Power*, oder übertragen auf Sprachen die *Least Powerful Language*. Demnach bietet eine weniger aussagekräftige Sprache mehr Möglichkeiten mit den Daten zu arbeiten, die in der Sprache abgebildet sind (vgl. BERNERS-LEE 2013).

Im Zusammenspiel mit anderen Technologien und Lösungen spielt darüber hinaus das Prinzip der *Modularität* eine wesentliche Rolle. Das bedeutet nicht zwingend, dass die eigene Lösung modular aufgebaut sein muss (auch wenn dies im Hinblick auf die Wartbarkeit häufig sinnvoll ist). Sie muss aber so gestaltet sein, dass sie Teil eines größeren Systems sein kann (vgl. BERNERS-LEE 2013). Klar definierte Schnittstellen zu anderen Technologien erlauben eine lose Kopplung und dadurch eine unabhängige Weiterentwicklung der beteiligten Systeme (vgl. BERNERS-LEE 2008).

Diese Modularität spiegelt sich nicht nur in der technischen Implementierung wider, sondern auch in den ihnen zugrundeliegenden Spezifikationen. Die Architektur des Webs fußt auf drei konzeptionellen Grundpfeilern: Identifikation, Interaktion und Datenformate. Die drei wichtigsten Spezifikationen des Webs beschreiben jeweils einen dieser Aspekte: URIs, HTTP und HTML. Dahinter steckt das Prinzip, orthogonale Konzepte in orthogonalen Spezifikationen abzubilden und somit unabhängig voneinander weiterentwickeln zu können (vgl. JACOBS, WALSH 2004).[1]

[1] Die Internationalisierung von URIs in Form von IRIs (vgl. DÜRST, SUIGNARD 2005), die Einführung von HTTP/2 (vgl. BELSHE, PEON, THOMSON 2015) und der HTML Living Standard (vgl. WHATWG 2019) verdeutlichen beispielhaft die Anwendung dieses Prinzips.

2.1.1 Identifikation durch URIs

Die fundamentalste Spezifikation der Webarchitektur ist die des Uniform Resource Identifiers (URI) (vgl. BERNERS-LEE 1998b). Sie ist der Nachfolger der 1994 verfassten Spezifikation für Uniform Resource Locators (URL), mit welchem Ressourcen im Internet über eine eindeutige Adresse lokalisiert werden können (vgl. BERNERS-LEE, MASINTER, MCCAHILL 1994). Mit dem URI hingegen lassen sich auch Ressourcen identifizieren, die nicht über einen Standort identifizierbar sind. Allerdings gilt es als Best Practice, dass ein URI eine Repräsentation der Ressource bereitstellt, die sie identifiziert (vgl. JACOBS, WALSH 2004).

URIs dienen als applikations- und domänenübergreifendes, einheitliches Identifikationssystem mit welchem sich nicht nur Webressourcen, sondern jegliche Konzepte global eindeutig identifizieren lassen. Ihre Omnipräsenz in fast allen standardisierten Webtechnologien fördert in großem Maße Netzwerkeffekte. Je konsequenter ein URI zur Identifizierung eines Informationselements genutzt wird, desto höher ist seine Aussagekraft (vgl. JACOBS, WALSH 2004). Auf der anderen Seite ist es aber auch möglich, dass Applikationen unabhängig voneinander dasselbe Konzept über verschiedene URIs referenzieren. Hinzu kommt, dass zwischen dem Konzept selbst und der Beschreibung des Konzepts unterschieden werden muss – beides sind Ressourcen, die über eigene URIs getrennt voneinander identifizierbar sein sollten (vgl. SAUERMANN, CYGANIAK 2008).

Der URI selbst sollte vom Nutzer als *Opaque String* betrachtet werden: eine Zeichenkette über dessen Aufbau und Bedeutung er keine Annahmen treffen darf und die selbst keine Aussagen über das referenzierte Konzept enthält (vgl. JACOBS, WALSH 2004). Im Kontext von Webschnittstellen wird gegen dieses Prinzip häufig verstoßen, indem die Konstruktion der URIs für die Ressourcen der Schnittstelle dem Client überlassen wird.

2.1.2 Interaktion via HTTP

Die Identifikation einer Ressource durch einen eindeutigen URI ist die Voraussetzung, um mit dieser Ressource interagieren. Dies geschieht durch den Austausch von Nachrichten über standardisierte Webprotokolle – beispielsweise HTTP, FTP oder SMTP (vgl. JACOBS, WALSH 2004). Häufig ist der URI hierbei auch ein *Uniform Resource Locator* (URL), informiert also über den Standort der Ressource.

Welches Protokoll zum Einsatz kommt ist abhängig vom Anwendungsfall. Der Zugriff auf Websites und Webschnittstellen erfolgt üblicherweise über das *Hypertext Transfer Protocol* (HTTP).

HTTP ist ein Applikationsprotokoll, das auf einer sehr allgemeinen Ebene die Interaktion mit Ressourcen im Web regelt. Die Semantik der ausgetauschten Nachrichten wird im Wesentlichen über die Anfragemethode und den Statuscode der Antwort kommuniziert.

Die Methode ist Teil der Nachricht, die der Client an den Server sendet. Insgesamt definiert die HTTP-Spezifikation acht Methoden. Für die Abfrage und Manipulation von Ressourcen sind insbesondere GET, POST, PUT und DELETE relevant. GET ist von den vier Methoden die einzige, die als *sicher* spezifiziert ist. Eine Anfrage mit GET darf demnach keine Veränderungen des Ressourcenzustands oder sonstige Seiteneffekte auslösen (vgl. FIELDING, RESCHKE 2014, S. 21f.). *Idempotente* Methoden garantieren, dass eine Wiederholung der Anfrage keine weiteren Änderungen bewirkt. Es ist also unerheblich, ob eine PUT- oder DELETE-Anfrage einmal oder mehrmals abgesendet wird, das Ergebnis ist dasselbe (vgl. FIELDING, RESCHKE 2014, 23f.).

Die Nachrichten, die der Server als Antwort auf eine Anfrage sendet, enthalten stets einen Statuscode, der über das Ergebnis einer Anfrage informiert. Konnte die Anfrage erfolgreich verarbeitet werden, so erhält der Client einen Statuscode aus dem 2xx-Bereich, Umleitungen auf andere Ressourcen werden mit 3xx-Codes gekennzeichnet und Fehlerfälle mit 4xx- oder 5xx-Codes, abhängig davon, ob der Fehler auf Client- oder Serverseite liegt (vgl. FIELDING, RESCHKE 2014, 47ff.).

Die korrekte Verwendung von Methoden und Statuscodes spielt eine wichtige Rolle, damit nicht nur Client und Server ein gemeinsames Verständnis über die Bedeutung der Nachricht besitzen, sondern auch zwischengeschaltete Systeme. Clientanfragen mit idempotenter Methode können beispielsweise gefahrlos wiederholt werden, wenn Netzwerkprobleme dazu führen, dass der Client keine Antwort erhalten hat. Ein Fehlercode in der Antwort eines Servers wiederum verhindert, dass diese durch einen Cache zwischengespeichert wird.

Neben Methoden und Statuscodes enthalten die zwischen Client und Server ausgetauschten Nachrichten Kopfzeilen mit Metadaten (*Request Header* bzw. *Response Header*) und optional einen Hauptteil mit dem Nachrichteninhalt (*Request Body* bzw. *Response Body*).

Über die Kopfzeilen werden verschiedene Aspekte des Nachrichtenaustauschs gesteuert. Wichtige Anwendungsfälle sind unter anderem Sicherheitsaspekte wie Autorisierung von Anfragen, die Steuerung des Nachrichten-Cachings sowie die Aushandlung des Nachrichtenformats.

```
1   GET / HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: application/json, application/xml, text/csv

4   HTTP/1.1 200 OK
5   Content-Type: application/json; charset=utf-8
```

Listing 1: Clientanfrage und Serverantwort mit Aushandlung des Media Types via Content Negotiation

Letzteres wird im Rahmen von HTTP auch als *Content Negotiation* bezeichnet. Der Client informiert den Server per `Accept`-Header in seiner Nachricht über die Nachrichtenformate, die er unterstützt. Der Server liefert die Antwort in einem der angegebenen Nachrichtenformaten und kommuniziert über den `Content-Type`-Header, für welches er sich entschieden hat (siehe *Listing 1*). Unterstützt der Server keines der gewünschten Formate, antwortet er mit dem Statuscode `406 Not Acceptable` (vgl. FIELDING, RESCHKE 2014, 18ff.).[2]

2.1.3 Datenformate und Hypermedia

Die Interaktion mit einer Webressource erfolgt durch den Austausch von Repräsentationen dieser Ressource. Das Web macht aber keine Vorgaben, welche Datenformate zu nutzen sind. Es liegt in der Verantwortung von Client und Server sich auf ein für beide Seiten verständliches Format zu einigen, welches über die Angabe des Media Types in den Kopfzeilen der Nachrichten identifiziert wird. Auf diese Weise können sich neue Datenformate unabhängig von der Ressourcenidentifikation und -interaktion etablieren, sollten neuen Anforderungen dies erforderlich machen (vgl. JACOBS, WALSH 2004).

Eine Datenformatspezifikation ist dabei nicht nur eine Beschreibung von Syntax und Aufbau, sondern steht für das gemeinsame Verständnis, wie Repräsentationsdaten interpretiert werden müssen (vgl. JACOBS, WALSH 2004). Ein XML-Parser kann beispielsweise eine XHTML-Datei verarbeiten, ist jedoch nicht in der Lage, die Semantik dieses Dokuments zu verstehen. Das `h1`-Element ist für ihn lediglich ein Auszeichnungselement für textbasierte Daten mit undefinierter Bedeutung. Ein HTML-Parser ist hingegen in der Lage, diese Daten als Überschrift erster Ordnung zu interpretieren.

Die fachliche Aussagekraft eines Datenformats entsteht somit erst über dessen Spezifikation. Wird ein Datenformat genutzt, das keine Unterstützung für die Fachdomäne der Anwendung bietet, führt dies zu einem Informationsverlust, der von den

---

[2] Analog kann auch eine Aushandlung von Zeichenkodierung, Zeichensatz oder Inhaltssprache erfolgen. Hierfür definiert die HTTP-Spezifikation die Anfrage-Header `Accept-Encoding`, `Accept-Charset` und `Accept-Language` sowie die Antwort-Header `Content-Encoding`, `Content-Charset` und `Content-Language`.

beteiligten Systemen auf andere Weise kompensiert werden muss. Im Falle einer Website ist dies in der Regel der Anwender, der HTML-Dokumente im Kontext der Fachlichkeit interpretiert.

Da mit der Ausarbeitung neuer Formate ein hoher Aufwand einhergeht, ist es in den meisten Fällen sinnvoller, vorhandene Datenformate zu verwenden oder diese für den eigenen Anwendungsfall zu erweitern. Idealerweise stellen sie hierfür Erweiterungsmechanismen bereit, so dass Erweiterungen eines Sprachstandards auch für Systeme, die den Standard nicht verstehen, erkenntlich sind (vgl. JACOBS, WALSH 2004).

HTML, das erste aller Datenformate im Web, hat darüber hinaus eine Funktion popularisiert, die aus dem Internet nicht mehr wegzudenken ist: Links. Sie sind ein definierendes Merkmal des Webs und mitverantwortlich für dessen Erfolg. (vgl. JACOBS, WALSH 2004).

Die Einbettung von Verweisen auf andere Ressourcen über URIs ist allerdings ein sprachübergreifendes Konzept mit eigener Spezifikation. Ein Link ist demnach eine typisierte Verbindung zwischen zwei Ressourcen und besteht aus einem Linkkontext, einem Link-Relationstyp und einem Linkziel (vgl. NOTTINGHAM 2017).

Der Link-Relationstyp ist optional, er spielt aber eine wichtige Rolle, wenn es darum geht, das fachliche Protokoll einer Anwendung auf Basis von HTTP abzubilden. Die Link-Spezifikation erlaubt dabei explizit die Definition eigener Link-Relationstypen in Form von URIs. Das hat den Vorteil, dass der URI auf eine Beschreibung des Typs verweisen kann. *Listing 2* zeigt beispielhaft die Auszeichnung eines HTML-Links mit dem Link-Relationstyp `https://example.org/rel/foo`.

```
1   <a href="/bar" rel="https://example.org/rel/foo">Click here</a>
```

Listing 2: Angabe eines Link-Relationstyps für einen HTML-Verweis

Über die Definition von Link-Relationstypen kann ein Media Type die Semantik von Hypermedia-Elementen und deren Eigenschaften spezifizieren. Ein Client, der das Datenformat implementiert, weiß somit, wie der Aufruf des Hypermedia-Elements erfolgen muss und was er fachlich bedeutet (vgl. NOTTINGHAM 2017).

Die Relevanz von Links in Webformaten wird deutlich, wenn man sich vor Augen führt, dass alle Ressourcen, die nicht in einem Hypermedia-fähigen Repräsentationsformat bereitgestellt werden, eine Sackgasse darstellen (vgl. JACOBS, WALSH 2004). Datenformate ohne Hypermedia-Unterstützung laufen daher dem Wesenskern des Webs zuwider:

Das World Wide Web verdankt seinen Namen der Grundidee, Informationen miteinander zu einem weltumspannenden Netz zu verknüpfen.

— TILKOV U. A. 2015, S. 71

2.2 Webservices

Neben der Nutzung des Webs durch den Menschen hat sich das Web schon frühzeitig auch als Plattform für die Machine-to-Machine-Kommunikation etabliert. Im Mittelpunkt stand dabei zunächst der Ansatz, *Remote Procedure Calls* (RPC) mit Webtechnologien abzubilden. Dafür werden die Funktionsaufrufe und ihre Antworten serialisiert und über die Protokolle des Webs zwischen den beteiligten Systemen ausgetauscht.

2.2.1 XML-RPC und SOAP

XML-RPC spezifiziert für die Serialisierung der Aufrufe und Antworten ein XML-basiertes Datenmodell, mit welchem sich Nachrichten formulieren lassen, die anschließend über HTTP transportiert werden (vgl. XML-RPC 1999). Aus der Weiterentwicklung dieser Spezifikation ist das wesentlich mächtigere Nachrichtenprotokoll SOAP entstanden. Im Gegensatz zu XML-RPC ist es durch das W3C standardisiert und nicht an ein konkretes Transportprotokoll gebunden – neben HTTP ist beispielsweise auch der Einsatz von SMTP oder JMS möglich. Konzeptionell steht nicht mehr der RPC im Mittelpunkt, sondern der Austausch von Nachrichten. Diese sind nach wie vor XML-basiert, bestehen jedoch aus einem Envelope, welcher den Nachrichten-Header mit Metainformationen enthält sowie den Nachrichten-Body, der die Nutzdaten umfasst. Der Aufbau des Bodys ist anwendungsspezifisch und kann so zum Austausch von Informationen gestaltet oder wie bisher zur Kodierung von RPCs genutzt werden (vgl. BOX U. A. 2000).

SOAP definiert somit ein von HTTP unabhängiges Nachrichtenformat. Das hat zur Folge, dass viele Elemente der HTTP-Spezifikation in SOAP keine Anwendung finden, da das Protokoll eigene Äquivalente definiert. Hierzu zählt beispielsweise die Kodierung von Metadaten über HTTP-Header oder die Verwendung von Statuscodes für die Kommunikation von Erfolgs- oder Fehlerfällen in der Bearbeitung einer Anfrage. HTTP wird somit nicht wie vorgesehen als Applikationsprotokoll genutzt, sondern lediglich als praktikable Transportschicht, die es erlaubt, bestehende Infrastruktur nutzen zu können und Unternehmensfirewalls einfacher zu überwinden.

Im Umfeld von SOAP sind eine ganze Reihe weiterer Spezifikationen entstanden, die jeweils Aspekte beschreiben, die von der SOAP-Spezifikation nicht betrachtet werden. Neben der *Web Services Description Language* (WSDL) zur Beschreibung von Webschnittstellen befassen sich diese als WS-* zusammengefassten Spezifikationen mit Aspekten wie Sicherheit, Auffindbarkeit, Adressierung und Orchestrierung von Webdiensten (vgl. W3C 2011).

2.2.2 Webservice-Architektur

Aus den konkreten Technologien rund um SOAP hat das W3C eine Referenzarchitektur für die Machine-to-Machine-Kommunikation basierend auf Webtechnologien abstrahiert. Motiviert ist die *Web Services Architecture* (WSA) durch die grundlegenden Designprinzipien des Webs – Interoperabilität und Erweiterbarkeit (siehe *2.1 Architekturprinzipien und Konzepte des Webs*, *Austin u. a. 2004*).

> *The architecture does not attempt to specify how Web services are implemented, and imposes no restriction on how Web services might be combined. The WSA describes both the minimal characteristics that are common to all Web services, and a number of characteristics that are needed by many, but not all, Web services.*
>
> *The Web services architecture is an interoperability architecture: it identifies those global elements of the global Web services network that are required in order to ensure interoperability between Web services.*
>
> — Booth u. a. 2004

Im Mittelpunkt steht dabei das Konzept der Webservices: Softwaresysteme, die über URIs identifizierbar sind, in abstrakter Form fachliche Funktionalität repräsentieren und diese über Webtechnologien bereitstellen. In der Definition des Begriffs durch die WSA ist dessen konzeptionelle Nähe zu den SOAP-Technologien deutlich zu erkennen:

> *A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.*
>
> — Booth u. a. 2004

Nicht zuletzt deshalb sind Webservices in der Wahrnehmung eng mit SOAP verbunden, allerdings wurde die Bezeichnung schon immer auch für andere webbasierte Dienste genutzt (vgl. PAUTASSO, ZIMMERMANN, LEYMANN 2008; PRESCOD 2002). Die explizite Erwähnung von WSDL zur Beschreibung von Webservices sowie die Festlegung auf SOAP-Messages als Nachrichtenformat und XML als Serialisierungssprache kann man daher als Widerspruch zur eingangs formulierten Absichtserklärung deuten, keine konkreten Vorgaben zur Implementierung von Webservices formulieren zu wollen. Die von der WSA beschriebenen Konzepte sind tatsächlich weitgehend technologieagnostisch und knüpfen an vielen Stellen an die Konzepte der Web-Architektur an (vgl. JACOBS, WALSH 2004).

Für eine Betrachtung von Machine-to-Machine-Kommunikation im Einklang mit den Prinzipien des Webs stellt die WSA daher eine wichtige konzeptionelle Grundlage dar, auch wenn SOAP und die damit verbundenen Technologien mit der Zeit an Bedeutung verloren haben.

2.2.3 Architekturmodelle und Konzepte

Die *Web Services Architecture* beschreibt vier Architekturmodelle, die jeweils unterschiedliche Konzepte einer Webservice-basierten Architektur in den Mittelpunkt stellen (vgl. BOOTH U. A. 2004, siehe *Abbildung 1*).

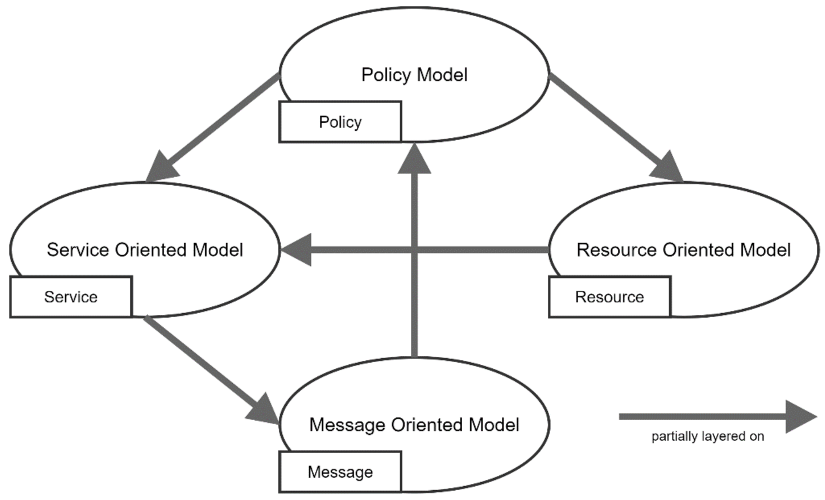


Abbildung 1: Architekturmodelle der Web Services Architecture nach BOOTH U. A. 2004

Allen Modellen gemein ist die Idee, dass Computerprogramme – so genannte *Agents* – im Auftrag von Personen oder Organisationen (*Entities*) handeln und durch den Austausch von Nachrichten (*Messages*) miteinander interagieren. Im Wesentlichen wird dieses Zusammenspiel vom *Message Oriented Model* beschrieben.

Aufgrund der üblicherweise nachrichtenbasierten Kommunikation im Web bildet dieses Modell zugleich auch die Grundlage für die anderen Modelle.

Im Mittelpunkt des *Service Oriented Model* stehen die Dienste (*Service*), die eine Person oder Organisation, die *Provider Entity*, über das Web anbieten will. Realisiert werden diese Services über *Provider Agents*, die die fachliche Funktionalität implementieren. Damit ein Service durch andere genutzt werden kann, wird er mit Hilfe einer maschinenlesbaren *Service Description* dokumentiert. Sie beschreibt die Schnittstelle und die darüber ausgetauschten Nachrichten, kann darüber hinaus aber auch eine Beschreibung der fachlichen Semantik enthalten (siehe *2.2.4 Contracts in Webservices*). Anschließend kann eine andere Person oder Organisation, die *Requester Entity*, einen zur *Service Description* kompatiblen *Requester Agent* implementieren und so den Service in Anspruch nehmen (vgl. BOOTH U. A. 2004).

Ein Service wird über einen URI identifiziert und ist somit im Sinne von BERNERS-LEE, FIELDING, MASINTER 2005 eine adressierbare Ressource, die die Dienste einer Person oder Organisation repräsentiert. Das *Resource Oriented Model* betrachtet Webservices aus Sicht dieser eindeutig identifizierbaren Ressourcen. Der Schwerpunkt liegt dabei auf der Frage der *Ownership* durch eine Person oder Organisation: Die Entity, in deren Besitz sich eine Ressource befindet, hat die Kontrolle über die Ressource und kann Richtlinien (*Policy*) für die Nutzung der Ressource durchzusetzen. Auf geltende Policys kann über die *Resource Description* hingewiesen werden, die auch die semantische Bedeutung der Ressource definiert (vgl. BOOTH U. A. 2004).

Policys sind unter anderem in Hinblick auf Querschnittsaspekte wie Sicherheit und Quality of Service relevant und stehen im Fokus des letzten Architekturmodells, dem *Policy Oriented Model*.

Schon bei dieser oberflächlichen Betrachtung der Architekturmodelle wird deutlich, in welchem Kontext die *Web Services Architecture* entstanden ist. Insbesondere im *Service Oriented Model* und im *Policy Oriented Model* ist der Einfluss von SOAP und den WS-*-Spezifikationen nicht zu übersehen. Sie knüpft andererseits aber auch an Kernprinzipien der Webarchitektur an, indem sie mit dem *Message Oriented Model* das wichtigste Kommunikationsprotokoll des Webs abbildet und im *Resource Oriented Model* zentrale Konzepte des Webs aufgreift. An dieser Stelle setzt der Architekturstil REST an. Im Gegensatz zu SOAP stehen nicht Services im Mittelpunkt, sondern Ressourcen. Das ändert jedoch nichts an der grundlegenden Tatsache, dass Agents im Auftrag der dahinterstehenden Personen und Organisationen

Nachrichten austauschen und auf diese Weise fachliche Funktionalität anbieten und konsumieren.

2.2.4 Contracts in Webservices

Unabhängig davon, ob man Webservices aus der Perspektive von Services oder Ressourcen betrachtet, müssen einige Voraussetzungen gegeben sein, damit Provider- und Requester-Seite Daten über Nachrichten miteinander austauschen können. Beide Parteien müssen sich einig sein

(1) über die Mechanik des Nachrichtenaustauschs,
(2) über die fachliche Bedeutung der ausgetauschten Daten
(3) und über fachliche Bedeutung des Datenaustauschs: welcher Zweck wird mir ihm verfolgt?

(vgl. BOOTH U. A. 2004)

Die Mechanik des Nachrichtenaustauschs umfasst den Nachrichtenaufbau, die verwendeten Datenformate und das Protokoll, über welches die Nachrichten transportiert werden. Die Mechanik eines SOAP-basierten Webservices kann beispielsweise derart beschrieben werden, dass Nachrichten bestehend aus einem SOAP-Envelope mit Header und Body als XML serialisiert über HTTP transportiert werden. Diese Informationen lassen sich in der Service Description maschinenlesbar festhalten. Im Falle von SOAP geschieht dies mit Hilfe der WSDL (vgl. BOOTH U. A. 2004).

Die fachliche Bedeutung der Daten sowie der Zweck ihres Austauschs ergeben die Semantik des Webservices. Ohne ein gemeinsames Verständnis für die Semantik können Daten falsch interpretiert werden. Beispielsweise besitzen die Lieferadresse einer Bestellung und ein Adresseintrag in einem Benutzerprofil möglicherweise dieselben Datenfelder, stehen aber dennoch für unterschiedliche Konzepte. Ebenso muss die Bedeutung des Nachrichtenaustauschs beiden Seiten unmissverständlich bekannt sein, damit sie sich über das Ziel und die Konsequenzen ihrer Interaktion im Klaren sind: Die Fehlinterpretation einer Reservierung als Bestellung hat in einem E-Commerce-Szenario reale Konsequenzen.

Eine maschinenlesbare Beschreibung der semantischen Aspekte eines Webservices ist im Vergleich zur Mechanik eine wesentlich komplexere Herausforderung und die Sinnhaftigkeit dieses Vorhabens wird vereinzelt auch in Frage gestellt (vgl. MARTIN U. A. 2004; KLUSCH 2008, 41-99). Eine Beschreibung einzelner semantischer Aspekte in maschinenlesbarer Form ist jedoch durchaus praktikabel und sinnvoll. Die eindeutige Identifikation von Konzepten innerhalb einer fachlichen

Domäne ist beispielsweise ein wesentliches Element der Softwaregestaltung nach Domain-Driven Design (vgl. EVANS 2011, S. 24).[3]

Alle Aspekte, die sich nicht in maschinenlesbarer Form festhalten lassen, müssen auf zwischenmenschlicher Ebene über verschiedene Kanäle transportiert werden. Dies kann informell über mündliche Vereinbarungen oder Konventionen kommuniziert geschehen, oder formell durch Dokumentation oder gar rechtsverbindliche Vereinbarungen (vgl. ERL 2006, S. 137–138; BOOTH U. A. 2004).

Die Einigung über die Mechanik und Semantik eines Webservices stellt den *Contract* dar, den die beteiligten Parteien eingehen. Der Contract ist somit nicht zwangsläufig ein konkretes Dokument, sondern repräsentiert als abstraktes Konzept die Einigung von Requester und Provider Entity auf ein gemeinsames Verständnis für die Mechanik und Semantik eines Webservices (vgl. BOOTH U. A. 2004).[4] Ein expliziter Contract schafft auf beiden Seiten Sicherheit und Klarheit über die Voraussetzungen für die Nutzung des Services. Er hält die technischen wie fachlichen Systemgrenzen fest und bildet eine Abstraktion, die über die konkrete Implementierung hinaus gültig ist. Beide Seiten sind dadurch lose miteinander gekoppelt und können sich unabhängig voneinander weiterentwickeln, sofern sie weiterhin die über den Contract vereinbarten Bedingungen einhalten (vgl. NEWCOMER, LOMOW 2005, S. 64).

Findet zwischen Requester und Provider weder formell noch informell eine Absprache über die Nutzung des Webservices statt, entsteht im ungünstigsten Fall ein impliziter Contract, der sich aus der konkreten Nutzung durch den Requester und dessen Interpretation der Schnittstelle ergibt. Neben der Gefahr von Fehlinterpretationen hat dies auch zur Folge, dass die Evolvierbarkeit des Webservices stark eingeschränkt wird, da der Provider mit *jeder* Änderung des Webservices einen potenziellen Bruch des Contracts riskiert.

Eine ähnliche Situation ist gegeben, wenn die Schnittstellenbeschreibung und Implementierung eng miteinander gekoppelt sind. Ein Beispiel hierfür sind SOAP-Webservices, die mit der WSDL beschrieben werden:

---

[3] *2.4 RDF und Linked Data* beschreibt, wie dies mit Hilfe des Resource Description Frameworks möglich ist.

[4] Weitergehende Definitionen von Contracts umfassen neben den technischen und semantischen Eigenschaften auch eine Beschreibung der nichtfunktionalen Attribute, beispielsweise Service Level Agreements, Nutzungsbedingungen oder Lizenzinformationen. Diese Aspekte werden im Rahmen der vorliegenden Arbeit jedoch nicht betrachtet.

> *While WSDL pays lip service to SOAP's message-oriented processing model, in fact it is mostly used as nothing more than a verbose object interface definition language (IDL), which forces an unsuitable RPC-like model of parameters, return values, and exceptions onto Web Services.*
>
> — WEBBER, PARASTATIDIS, ROBINSON 2010, S. 381

Das von der WSDL geförderte methodenorientierte Abstraktionsmodell eignet sich hervorragend als Abstraktionsschicht für die Programmiersprachen, mit denen Webservices implementiert werden. Es läuft jedoch der Architektur des Webs zuwider und führt häufig zu einer engen Kopplung der WSDL-basierten Schnittstellenbeschreibung und der dahinterliegenden Implementierung. Auch hier kann eine Änderung der Webservice-internen Logik schnell einen Bruch des Contracts zur Folge haben (vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, S. 382).

Die durch die WSDL erzeugte Komplexität ist einer der Faktoren, die dazu geführt haben, dass sich der Architekturstil REST als Alternative zum Ökosystem rund um SOAP etabliert und dieses in vielen Bereichen verdrängt hat.

2.3 REST als Architekturstil

Representational State Transfer (REST) ist ein von FIELDING 2000 beschriebener Architekturstil für verteilte Systeme. Es ist somit kein Standard und keine konkrete Technologie, sondern eine Reihe von Leitsätzen und bewährten Praktiken, die auf die Gestaltung von skalierbaren, langlebigen und evolvierbaren Applikationen abzielen:

> *REST is software design on the scale of decades: every detail is intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency.*
>
> — FIELDING 2008

Nicht zufällig klingen hier ähnliche Architekturziele durch, wie sie auch für das Web gelten. REST ist eine Abstraktion der Architektur des Webs mit dem Ziel, sie generell auf verteilte Systeme anwendbar zu machen. REST ist daher unabhängig von konkreten Standards wie URIs und HTTP zu sehen und könnte theoretisch auch im Kontext anderer Technologien angewandt werden. In der Praxis bleibt das Web jedoch die einzig relevante Implementierung (vgl. TILKOV U. A. 2015, S. 10). Die Popularität des REST-Architekturstils im Zusammenhang mit Web-APIs ist der Tatsache geschuldet, dass er einen Leitfaden bietet, um verteilte Systeme basierend

auf der Architektur des Webs umzusetzen (vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, 1f.).

Die Idee, Applikationen über das Web miteinander kommunizieren zu lassen, ist nicht erst mit REST aufgekommen. Als Transportprotokoll kam HTTP schon bei XML-RPC und SOAP zum Einsatz (siehe *2.2.1 XML-RPC und SOAP*). In REST-konformen Applikationen ist das Web jedoch nicht allein die Transportschicht, sondern die architektonische Basis. URIs werden genutzt, um Ressourcen eindeutig zu identifizieren, HTTP dient als Applikationsprotokoll zur Interaktion mit diesen Ressourcen und der Nachrichtenaustausch erfolgt unter Verwendung Hypermedia-fähiger Datenformate.

REST-Applikation nutzen das Web im Sinne seiner Architektur (vgl. TILKOV U. A. 2015, S. 10).[5] Sie weisen daher im Hinblick auf Interoperabilität und Evolvierbarkeit ähnliche Eigenschaften wie das Web auf und gelten zudem als hoch skalierbar und einfacher zu verstehen (vgl. WILDE, PAUTASSO 2011, S. 2; TILKOV 2015, S. 138).

2.3.1 Prinzipien

Den Kern von REST bilden eine Reihe von Prinzipien, die FIELDING 2000 als *Contraints* beschreibt.

Das erste Prinzip, *Client-Server*, ist für Webanwendungen gängige Praxis. Eine Serverkomponente bietet eine Funktionalität an, die von der Clientkomponente genutzt wird. Die Initiative geht dabei stets vom Client aus. Die Motivation hinter diesem REST-Constraint ist die Separation of Concerns und damit einhergehend eine geringere Abhängigkeit zwischen den beteiligten Systemen. Dies fördert wiederum die Evolvierbarkeit der einzelnen Komponenten (vgl. FIELDING 2000, S. 96).

Die Kommunikation zwischen Client und Server erfolgt *stateless*. Die ausgetauschten Nachrichten müssen alle Informationen enthalten oder referenzieren, die für ihre Verarbeitung notwendig sind (vgl. FIELDING 2000, 96f.). Die Rede ist daher auch von selbstbeschreibenden Nachrichten (siehe *2.3.2.3 Selbstbeschreibende Nachrichten*). Für den Server bedeutet dies, dass er alle Anfragen des Clients unabhängig voneinander betrachten und verarbeiten kann (vgl. RICHARDSON, AMUNDSEN 2013, S. 360). Die Zustandslosigkeit der Kommunikation in REST ist ein wichtiger Grund für die positiven Skalierungseigenschaften des Architekturstils.

[5] TILKOV 2015 vergleicht dies mit einer relationalen Datenbank, die lediglich als Key-Value-Store genutzt wird. Es ist technisch möglich, ignoriert aber die zugrundeliegende Architektur und deren Vorteile (vgl. TILKOV 2015, S. 134).

Ein weiteres Prinzip, das sich positiv auf die Skalierbarkeit auswirkt, ist die Anforderung, dass Nachrichten des Servers darüber Auskunft geben müssen, ob sie *cacheable* sind. In der Praxis werden hierfür die Cache-Header von HTTP eingesetzt (vgl. FIELDING 2000, 97f.). So informiert der Server nicht nur den Client, sondern auch zwischengeschaltete Systeme darüber, ob und wie lange Antworten zwischengespeichert werden dürfen. Es liegt demnach in der Verantwortung des Servers festzulegen, in welcher Form ein Caching stattfindet. Hier unterscheidet sich das Web von vielen anderen Technologien, die die Cache-Kontrolle den Schnittstellennutzern überlassen (vgl. STURGEON 2017).

Neben Caching werden auch Aspekte wie Sicherheit oder Load-Balancing häufig über zwischengeschaltete Systeme abgebildet. Diese Architektur aus separaten, aufeinander aufbauenden Schichten beschreibt das REST-Prinzip *Layered System*. Jede Schicht kann dabei nur die direkt unter ihr liegende Schicht sehen. Dadurch wird die Gesamtkomplexität des Systems reduziert und es ergeben sich verschiedene Möglichkeit für die Kapselung von Altsystemen (vgl. FIELDING 2000, 100f.).

Das Prinzip *Code-on-Demand* beschreibt die Möglichkeit, dass der Client Features bei Bedarf vom Server nachlädt, anstatt sie selbst zu implementieren. Dieser Constraint ist optional und spielt in der Praxis selten eine Rolle (vgl. FIELDING 2000, S. 102).

Das Prinzip, welches das Bild von REST am meisten prägt, ist das *Uniform Interface*. Es stellt das wesentliche Merkmal dar, welches REST von anderen netzwerkbasierten Architekturstilen abhebt.

2.3.2 Uniform Interface

Die Funktionalität einer REST-basierten Anwendung wird über Ressourcen abgebildet, die eine einheitliche Schnittstelle anbieten. Durch diese generische Schnittstelle wird die Gesamtarchitektur vereinfacht und die Sichtbarkeit der Interaktion mit Ressourcen gefördert, da zwischengeschaltete Applikationen die Kommunikation auf einer allgemeinen Ebene verstehen, ohne ein Verständnis für die konkrete Fachlichkeit der Nachricht haben zu müssen. Die Einhaltung dieses Prinzips fördert darüber hinaus die lose Kopplung der Systemkomponenten und erleichtert dadurch deren Weiterentwicklung (vgl. FIELDING 2000, 99f.).

Das Uniform Interface stellt vier Anforderungen an die Schnittstelle, die eine Applikation veröffentlicht:

(1) Ressourcen müssen eindeutig identifizierbar sein,
(2) die Interaktion mit Ressourcen erfolgt über den Austausch von Repräsentationen,
(3) die ausgetauschten Nachrichten sind selbstbeschreibend und
(4) der Applikationszustand wird über Hypermedia gesteuert.

(vgl. FIELDING 2000, S. 100).

Die Parallelen zu den Grundpfeilern der Webarchitektur sind klar zu erkennen (siehe *2.1 Architekturprinzipien und Konzepte des Webs*). Im Folgenden wird erläutert, wie REST-Applikationen durch den Einsatz von URIs, HTTP und Hypermedia-Formaten diesen Anforderungen entsprechen.

2.3.2.1 Identifizierbare Ressourcen

Die zentralen Bausteine von REST-APIs sind identifizierbare Ressourcen. Allgemein formuliert ist eine Ressource jedes Konzept, das über einen Namen identifiziert werden kann (vgl. FIELDING 2000, S. 106). Häufig sind dies die fachlichen Kernkonzepte, die eine Anwendung nach außen bereitstellt. Ihr Schnitt richtet sich nach den Anforderungen, die die Konsumenten an die API stellen. Es ist daher wichtig zwischen Ressourcenmodell der Webschnittstelle und internem Entitätenmodell der Anwendung zu unterscheiden. Der Ressourcenschnitt ist Teil des Contracts mit den Nutzern der Schnittstelle, lässt aber keine Rückschlüsse auf die Implementierung der Schnittstelle zu. Es muss möglich sein, die Implementierung zu ändern, ohne dass dies Auswirkungen auf die Schnittstelle hat (vgl. ROBINSON 2011, S. 68).

Die Identifikation von Ressourcen erfolgt über URIs und ist daher global eindeutig. Im Kontext von HTTP-basierten Applikationen informiert sie in Form des URLs zugleich über den Standort und liefert dem Client somit alle notwendigen Informationen, um auf die Ressource zuzugreifen.

Die Werte einer Ressource stellen den Ressourcenzustand dar. Dieser kann statisch sein oder sich über die Zeit verändern. Statisch bleibt jedoch in jedem Fall die semantische Bedeutung der Ressource. Der Zustand ist, wie die Ressource selbst, zudem unabhängig von der Repräsentation (vgl. RICHARDSON, AMUNDSEN 2013, S. 359–360; FIELDING 2000, S. 107).

2.3.2.2 Abfrage und Manipulation mittels Repräsentationen

Jede Ressource eines Servers stellt für die Abfrage und Manipulation ihres Zustands dieselben Methoden bereit. Diese bilden den Kern des Uniform Interface von REST-Applikationen. Sie beschreiben auf einer sehr allgemeinen Ebene, was mit dem Zustand der Ressource geschehen soll. In webbasierten REST-APIs werden hierfür üblicherweise die Methoden von HTTP genutzt (vgl. ROBINSON 2011, S. 64). Im Gegensatz zu RPC-basierten Schnittstellen werden somit alle fachlichen Vorgänge auf Basis dieser Verben abgebildet. Dadurch findet eine Verschiebung der Semantik statt hin zu den Ressourcen und ihren Beziehungen untereinander.

Die Ressource stellt einen Kommunikationsendpunkt dar, mit dem der Client auf Basis der angebotenen Methoden interagieren kann. Die Abfrage einer Ressource via GET liefert nicht die Ressource selbst, sondern eine Repräsentation des aktuellen Zustands der Ressource. Ebenso sendet der Client beim Erstellen von neuen Ressourcen oder Manipulieren von vorhandenen eine Repräsentation des gewünschten Ressourcenzustands. Die Verwaltung des Ressourcenzustands liegt somit in der Verantwortung des Servers. Der Client arbeitet mit Repräsentationen, die den Zustand der Ressource zu einem bestimmten Zeitpunkt abbilden (vgl. FIELDING 2000, S. 108f.).

Das Format dieser Repräsentationen wird via Content Negotiation ausgehandelt. Die Ressource und ihre URI sind daher unabhängig von der Repräsentation. Der Zustand einer Ressource kann über dieselbe URI als XML-Dokument, HTML-Seite oder Grafik abgefragt werden, sofern sich die beteiligten Seiten auf ein gemeinsames Format einigen können (vgl. FIELDING 2000, S. 107).

2.3.2.3 Selbstbeschreibende Nachrichten

Die Nachrichten, die Client und Server untereinander austauschen, müssen selbstbeschreibend sein. Sie enthalten oder referenzieren alle Informationen, die für ihr Verständnis erforderlich sind (vgl. TILKOV 2015, S. 135). Dies ist notwendig, um eine zustandslose Kommunikation zwischen Client und Server zu ermöglichen.

Die Angabe eines Media Types zur Deklaration des Nachrichtenformats ist ein wichtiger Schritt in Richtung selbstbeschreibender Nachrichten, da über dessen Spezifikation sowohl die Syntax und der Aufbau als auch die Semantik der Nachricht kommuniziert werden kann.

Generische Media Types wie XML oder JSON definieren hingegen lediglich Aufbau und Syntax der Nachrichten, nicht jedoch die fachliche Semantik. Ohne zusätzliche Kontextinformationen ist eine Interpretation des Nachrichteninhalts nicht möglich.

Im Sinne von REST gelten diese Nachrichten daher nicht als selbstbeschreibend. *2.3.3 Contracts in REST* wird auf diese Problematik näher eingehen.

2.3.2.4 Hypermedia as the Engine of Application State

Dank selbstbeschreibender Nachrichten kann der Server Clientanfragen isoliert betrachten und verarbeiten. Aus Serverperspektive existiert kein Zusammenhang zwischen einzelnen Anfragen. Daher besteht auch keine Notwendigkeit, Informationen über den Zustand des Clients zu speichern – beispielsweise in Form einer Session (vgl. WILDE, PAUTASSO 2011, S. 3–4).

Aus Sicht des Clients bauen die einzelnen Anfragen jedoch aufeinander auf. Einige Anfragen sind erst möglich, wenn zuvor andere Anfragen ausgeführt wurden – sei es, um Informationen zu sammeln oder um Ressourcenzustände zu manipulieren. Jede Interaktion mit dem Server stellt einen Schritt in dem fachlichen Ablauf dar, den der Client gemeinsam mit der API durchläuft. Die Position des Clients innerhalb dieses Ablaufs und die dabei von ihm aggregierten Informationen werden als Applikationszustand (*Application State*) bezeichnet (vgl. RICHARDSON, AMUNDSEN 2013, S. 357). Da jede weitere Anfrage stets einen neuen Applikationszustand und potenziell auch einen neuen Ressourcenzustand zur Folge hat, spricht man beim Aufruf einer URL auch von einem *Zustandsübergang* (vgl. ebd., S. 11).

Aufgrund des Client-Server-Prinzips liegt es in der Verantwortung des Clients, diese Zustandsübergänge zu initiieren und die Integrität einer Folge von Interaktionen zu gewährleisten (vgl. ROBINSON 2011, S. 65). Durch die Nutzung eines hypermediafähigen Nachrichtenformats kann der Server den Client jedoch über gültige Zustandsübergänge informieren und so Einfluss auf dessen Verhalten nehmen. Hierfür bettet er entsprechende Links in die Repräsentationen ein, die er an den Client übermittelt. Unzulässige Zustandsübergange werden vom Server von vornherein nicht angeboten, indem er das entsprechende Hypermedia-Element aus der Antwort weglässt. Auf diese Weise gelingt es, dem Client-Server-Prinzip gerecht zu werden, den Applikationszustand auf Client-Seite zu halten und dennoch serverseitig Einfluss auf die Zustandsübergänge in der Applikation auszuüben (vgl. RICHARDSON, AMUNDSEN 2013, S. 13). Diese Steuerung des Applikationszustands über Hypermedia bezeichnet FIELDING 2000 als *Hypermedia as the Engine of Application State* (vgl. ebd., S. 82).

Im Ergebnis entsteht so eine Web-API, die der Funktionsweise einer Website ähnelt. Dem Konsumenten muss nur die Einstiegs-URL bekannt sein. Anschließend entscheidet er anhand der vom Server angeboten Links über seine nächsten Schritte. Damit er diese Entscheidung treffen kann, ist es wichtig, dass die

Hypermedia-Elemente mit Link-Relationstypen ausgezeichnet sind. Sie identifizieren die Semantik eines Links und beschreiben in welchem Verhältnis der aktuelle Kontext zur verlinkten Ressource steht. Sie können aber auch signalisieren, dass die Ressource bestimmte Attribute oder Verhaltensweisen aufweist (vgl. NOTTINGHAM 2017). Dazu gehört beispielsweise, mit welcher Methode und welchen Anfragedaten sie aufgerufen werden kann.

2.3.3 Contracts in REST

Die wesentlichen Merkmale von REST stellen etablierte Lösungen zur Schnittstellenbeschreibung vor eine Herausforderung.

Populäre Technologien wie WSDL oder OpenAPI dokumentieren bereits zur Entwurfszeit alle URLs und deren HTTP-Methoden. Dadurch wird das Konzept von *Hypermedia as the Engine of Application State* untergraben, welches die Steuerung des Clients durch Links zur Laufzeit vorsieht. Die enge Kopplung, die daraus entsteht, wird durch Einschränkungen bei den Datenformaten noch weiter verstärkt. Während WSDL nur in Kombination mit SOAP und dessen Nachrichtenformat sinnvoll genutzt werden kann, ist OpenAPI eng an JSON (oder seltener XML) als Datenformat gebunden (siehe *3.1 OpenAPI*). Eine Aushandlung des Datenformats zur Anfragezeit ist dadurch nicht mehr möglich.

Wie entsteht also der Contract zwischen Anbieter und Konsumenten einer REST-Schnittstelle, wenn nicht alle Aspekte bereits zur Entwurfszeit in einer Schnittstellenbeschreibung festgehalten werden können?

2.3.3.1 Media Types und Protokolle

Den Kern von Contracts für REST-APIs bilden Media Types (siehe *Abbildung 2*). Die Einigung zwischen Anbieter und Konsument einer Schnittstelle auf einen Media Type stellt sicher, dass beide Seiten die Nachrichten verarbeiten können, die zwischen ihnen ausgetauscht werden (siehe *2.1.2 Interaktion via HTTP*).

Der Media Type selbst ist nicht mehr als eine Zeichenkette, die die Art eines Nachrichteninhalts identifiziert (vgl. FREED, KLENSIN, HANSEN 2013, S. 2). Erst über dessen Spezifikation erfährt der Nutzer, wie Nachrichten formuliert und interpretiert werden müssen, deren Inhaltstyp mit dem Media Type ausgezeichnet ist. Für die Art der Spezifikation gibt es dabei keine Vorgabe. Üblicherweise erfolgt sie als für den Menschen aufbereitete Dokumentation, einen etablierten maschinenlesbaren Standard gibt es nicht (vgl. ebd., S. 14; ALLEN 2016).

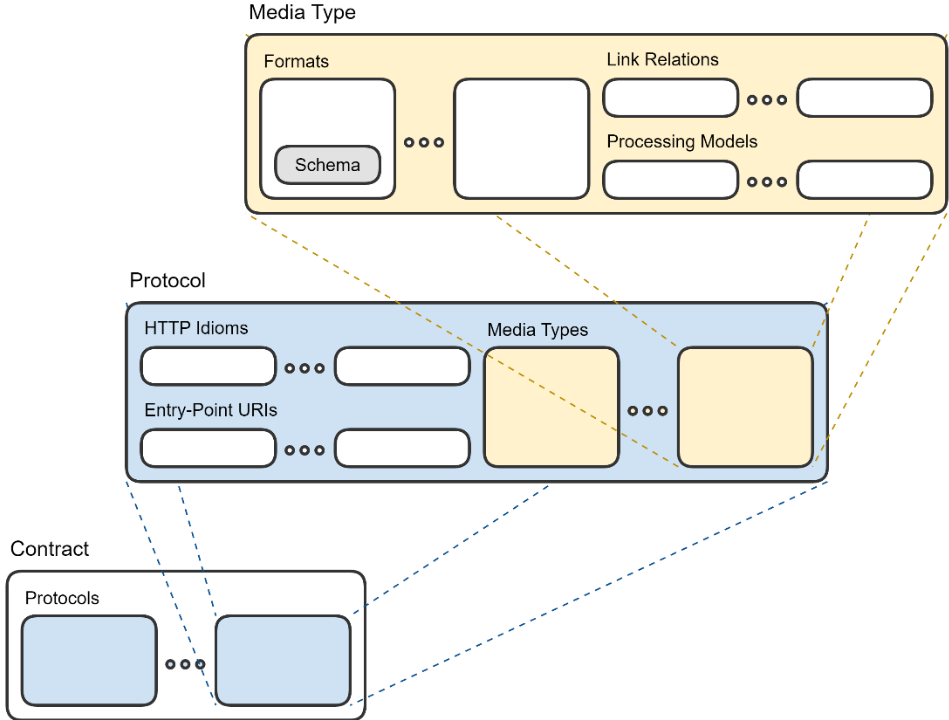


Abbildung 2: Contracts als Komposition aus Media Types und Protokollen nach WEBBER, PARASTATIDIS, ROBINSON *2010, S. 109*

Die Spezifikation eines Media Types kann neben der reinen Syntax des Datenformats auch eine semantische Beschreibung der Auszeichnungselemente, Link-Relationstypen und Verarbeitungsmodelle umfassen (vgl. RICHARDSON, AMUNDSEN 2013, S. 358; vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, S. 109). So lassen sich über den Media Type alle Informationen dokumentieren, die der Konsument einer REST-Schnittstelle benötigt, um einen kompatiblen Client zu implementieren. Alle weiteren Informationen werden erst zu einem späteren Zeitpunkt benötigt.

Die Einstiegs-URL der REST-API muss dem Client spätestens zum Startpunkt der Interaktion bekannt sein. Sie kann in der Implementierung fest hinterlegt sein oder zur Laufzeit injiziert werden. Nach Abruf der Einstiegsressource wird der Client anschließend über Links in den Ressourcenrepräsentationen über die URIs weiterer Ressourcen informiert.

Der letzte Baustein, den der Client für die Nutzung der Schnittstelle benötigt, ist die Protokollsemantik. Sie beschreibt, wie fachliche Zustandsübergänge in HTTP abgebildet werden (vgl. RICHARDSON, AMUNDSEN 2013, S. 138). Beispielsweise könnte die Protokollsemantik für die Bestellung eines Warenkorbs so aussehen, dass die Warenkorbdaten zusammen mit Adress- und Zahlungsinformationen in einem JSON-basierten Format kodiert via POST-Anfrage an den Server gesendet werden. Protokollsemantik kann entweder direkt in den Repräsentationen

transportiert werden oder wurde im Rahmen der Spezifikation von Link-Relationstypen oder Hypermedia-Elementen über den Media Type dokumentiert.[6]

Die Kombination aus eingesetzten Media Types, Einstiegs-URLs und Protokollsemantik ergeben Protokolle (siehe *Abbildung 2*), die die fachlichen Abläufe beschreiben, welche vom REST-Webservice implementiert werden (vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, S. 108).

2.3.3.2 Contracts im Web

Der Contract zwischen Requester und Provider kommt im Wesentlichen durch die Aushandlung des Media Types im Rahmen der Content Negotiation zustande. Der Media Type steht für das gemeinsame Verständnis von Mechanik und Semantik der Schnittstelle (vgl. WEBBER, PARASTATIDIS, ROBINSON 2010, S. 103; RICHARDSON, AMUNDSEN 2013, S. 358). Auch wenn dem Client zu diesem Zeitpunkt noch nicht alle Informationen zur Verfügung stehen ist dank dieser Einigung sichergestellt, dass alle Voraussetzungen gegeben sind, damit der Client die von der REST-Schnittstelle implementierten Protokolle ausführen kann. Der vollständige Contract manifestiert sich somit erst zur Laufzeit im Zusammenspiel von Media Types und Protokollen (siehe *Abbildung 2*).

Schnittstellenbeschreibungen, deren Ziel es ist, die Interaktion mit einer Schnittstelle bereits im Vorfeld vollständig zu beschreiben, lassen sich daher nur schwer mit den Prinzipien von REST und der Architektur des Webs vereinbaren (vgl. WILDE, PAUTASSO 2011, S. 6).

> *The Web breaks away from the traditional way of thinking about upfront agreement on all aspects of interaction for a distributed application. Instead, the Web is a platform of well-defined building blocks from which distributed applications can be composed. Hypermedia can act as instant and strong composition glue.*
>
> — WEBBER, PARASTATIDIS, ROBINSON 2010, S. 108

[6] Die HTML-Spezifikation bietet für beide Vorgehensweisen Beispiele. Sie definiert, dass verlinkte Ressourcen unabhängig vom verwendeten HTML-Element, sofern nicht anders angegeben, mit der `GET`-Methode aufgerufen werden. Dieses Standardverhalten lässt sich beispielsweise im Falle des `form`-Elements mit dem `method`-Attribut überschreiben. Über das `enctype`-Attribut desselben Elements kann der Server dem Client zudem mitteilen, in welchem Datenformat er die Anfrage erwartet. Ein HTML-kompatibler Client muss somit auch die laut HTML-Spezifikation für dieses Attribut gültigen Media Types `application/x-www-form-urlencoded`, `multipart/form-data` und `text/plain` unterstützen (vgl. WHATWG 2019).

Das Zustandekommen des Contracts aus verschiedenen Einzelteilen spiegelt das Prinzip orthogonaler Spezifikationen wider und fördert die Modularität und Erweiterbarkeit der Lösung (siehe *2.1 Architekturprinzipien und Konzepte des Webs*).

Gleichzeitig wird der Client in die Pflicht genommen. Von ihm wird eine flexible und tolerante Implementierung erwartet. Denn erst im Rahmen der weiteren Interaktion entscheidet sich, welche Ressourcen und Zustandsübergänge tatsächlich verfügbar sind. Es ist außerdem möglich, dass sich die fachlichen Modelle der Schnittstelle seit der Implementierung des Clients weiterentwickelt haben und die Repräsentationen (in abwärtskompatibler Weise) zusätzliche Auszeichnungselemente oder Link-Relationstypen enthalten, die dem Konsumenten unbekannt sind. Der Client muss auf derartige Veränderungen vorbereitet sein, damit die Evolvierbarkeit der API gewährleistet bleibt. Im Web bedeutet dies nicht, dass der Contract zwischen Requester und Provider seine Gültigkeit verloren hätte. Stattdessen wird vom Client erwartet, dass er diese Elemente ignoriert. Er darf sich auch nicht darauf verlassen, dass bestimmte Links stets Teil einer Ressourcenrepräsentation sind. Geänderte Geschäftsregeln, Berechtigungen oder Systemverfügbarkeiten können dazu führen, dass ein Link dauerhaft oder temporär nicht angeboten wird.

Auf den ersten Blick erscheint diese Herangehensweise an Contracts im Web kontraproduktiv. Immerhin ist das Ziel eines Vertrags, alle Unwägbarkeiten bereits im Vorfeld auszuschließen und klare Verhältnisse zu schaffen. Betrachtet man jedoch die Evolution des Webs, werden die Stärken dieses Vorgehens deutlich. Trotz kontinuierlicher Weiterentwicklung des HTML-Standards ist dessen Media Type nach wie vor `text/html`. Der Contract, den ein Browser 1996 mit dem Server einer Website eingegangen ist, hat heute noch Bestand.

Abbildung 3 verdeutlicht dies anhand der Spezifikation von HTML 2.0 (vgl. Berners-Lee, Connolly 1995) und dem Working Draft zu HTML 5.3 (vgl. Aas u. a. 2018). Der Internet Explorer 3 (oben) ist in der Lage, die über zwanzig Jahre später veröffentlichte HTML-5.3-Spezifikation anzuzeigen,[7] während Edge 79 (unten) keine Probleme mit der Darstellung der wesentlich älteren HTML-2.0-Spezifikation hat.

[7] Die für das W3C-Logo notwendige Unterstützung von SVG-Grafiken bekam der Internet Explorer erst fünfzehn Jahre später.

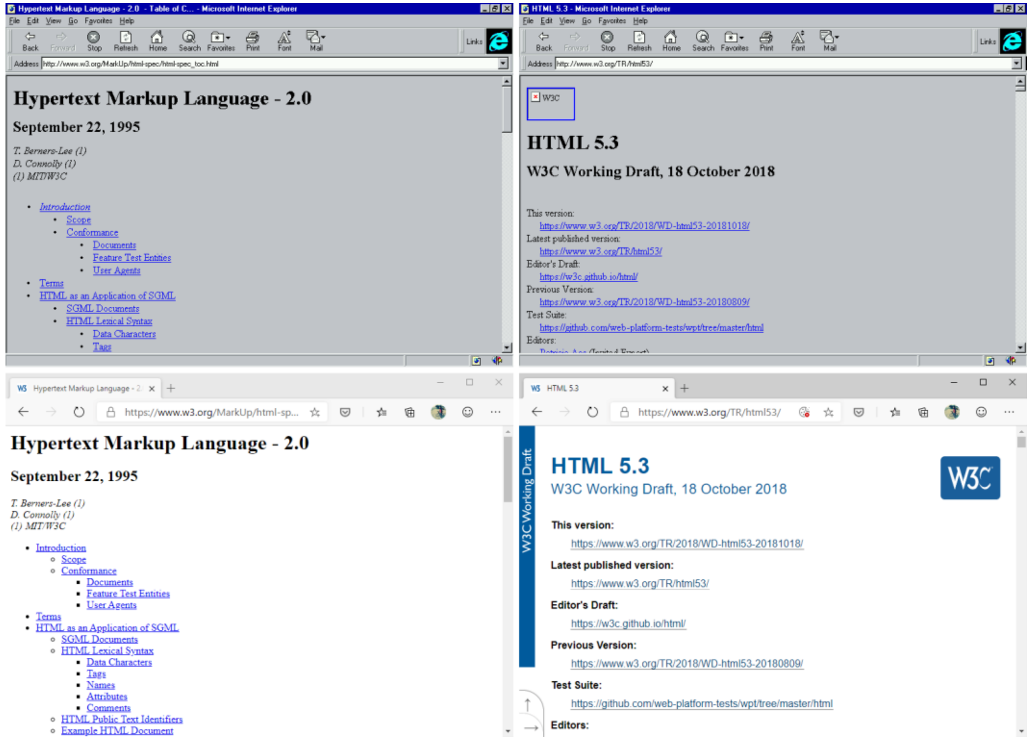


Abbildung 3: Die Spezifikationen von HTML 2.0 und 5.3 im Internet Explorer 3 und Edge 79

Nicht jede Schnittstelle im Web muss eine jahrzehntelange Abwärtskompatibilität gewährleisten – nur wenige Schnittstellen existieren überhaupt so lange. Je wichtiger jedoch Evolvierbarkeit und Interoperabilität in der Abwägung der Qualitätsmerkmale einer Systemarchitektur sind, desto mehr Gründe sprechen für die Gestaltung von Contracts im Sinne der Webarchitektur. Die Nutzung etablierter Standards kann zudem den damit verbundenen Spezifikations- und Implementationsaufwand verringern.

2.3.3.3 Standardisierte und benutzerdefinierte Media Types

Die Wahl geeigneter Media Types für eine Schnittstelle ist eng mit deren fachlicher Domäne verknüpft. Weit verbreitete Standards wie XML oder JSON bieten die größtmögliche Interoperabilität für den Datenaustausch, vermitteln für sich genommen aber keinerlei Semantik. Der Empfänger einer Nachricht, die mit dem Media Type `application/json` ausgezeichnet ist, kann somit lediglich Auszeichnungselemente von Werten unterschieden, ist jedoch nicht in der Lage die kodierten Daten ohne zusätzliche Informationen zu interpretieren (vgl. ALLEN 2016). Für das REST-Prinzip der selbstbeschreibenden Nachricht reicht dies nicht aus (siehe *2.3.2.3 Selbstbeschreibende Nachrichten*).

Existiert für die fachliche Semantik des Anwendungsfalls kein geeigneter Standard, besteht die Möglichkeit, einen eigenen Media Type zu definieren. Der einfachste Weg ist die Nutzung des `x.`-Präfixes. Media Types mit diesem Subtype erfordern

keine Registrierung bei der IANA und sind daher mit keinerlei Aufwand oder Verpflichtungen verbunden. Ein solcher Media Type könnte beispielsweise `application/x.hypershop` lauten. Das `x.`-Präfix darf jedoch nur in lokalen, geschlossenen Umgebungen verwendet werden (vgl. FREED, KLENSIN, HANSEN 2013, S. 7).

Applikationsspezifische Media Types für öffentliche Schnittstellen müssen bei der IANA registriert werden. Da dies eine umfassende Spezifikation erfordert, ist es häufig sinnvoll (und vom W3C empfohlen), ein etabliertes Format als Ausgangspunkt zu nutzen und auf dieser Basis die fachlichen Aspekte zu spezifizieren (vgl. JACOBS, WALSH 2004; ALLEN 2016). Nichtsdestotrotz ist die Registrierung eines eigenen Media Types ein aufwändiges Unterfangen und jede Änderung der Spezifikation erfordert ein erneutes Durchlaufen dieses Prozesses (vgl. FREED, KLENSIN, HANSEN 2013, 7ff.).

Eine mögliche Alternative sind die im folgenden Abschnitt beschriebenen Profile.

2.3.3.4 Profile

Profile ermöglichen es, eine Nachricht um zusätzliche Semantik zu ergänzen, die vom Media Type nicht spezifiziert ist.

> *[...] a profile is the appropriate mechanism to signal that the original semantics and processing model of the media type still apply, but that an additional processing model can be used to extract additional semantics.*
>
> — WILDE 2013, S. 4

Das Profil verändert dabei nicht die Semantik einer Nachricht, sondern informiert über Semantik (Einschränkungen, Konventionen, Erweiterungen), die über den Media Type hinausgeht. Der Media Type für sich genommen verliert somit nicht seine Gültigkeit. Ein Client, der ein Profil nicht kennt, kann die Nachricht noch immer auf Basis des Media Types interpretieren, allerdings wird er nicht alle Aspekte ihres Inhalts verstehen (vgl. WILDE 2013, S. 1; RICHARDSON, AMUNDSEN 2013, S. 359).

Betrachtet man erneut das Beispiel eines Online-Shops, dessen fachliche Semantik auf die dokumentenzentrierte Semantik von HTML abgebildet wird, ermöglicht die Angabe eines Profils in Ergänzung zum Media Type `text/html` die Definition der fachlichen Semantik des Online-Shops. Für die Konsumenten, die dieses Profil unterstützen, wird damit verständlich, welche Elemente auf der Seite beispielsweise ein Produkt beschreiben. Für alle anderen Konsumenten bleibt die Semantik von

HTML erhalten, die beispielsweise den Namen des Produkts als Überschrift erster Ordnung auszeichnet.

Durch den Einsatz von Profilen können Nachrichten so im Sinne von REST selbstbeschreibend formuliert werden, ohne dass hierfür ein applikationsspezifischer Media Type notwendig ist. Während der Media Type Syntax, Datenformat und Hypermedia-Elemente definiert, steuert das Profil alle applikationsspezifischen Informationen bei. Diese umfassen einerseits fachliche Semantik wie die Bedeutung von Auszeichnungselementen, Feldnamen und Link-Relationstypen. Andererseits kann das Profil aber auch genutzt werden, um Protokollsemantik zu vermitteln, sollte dies über den eingesetzten Media Type nicht möglich sein. In diesem Fall definiert das Profil über den Link-Relationstyp nicht nur die Bedeutung eines Links und die Konsequenzen von dessen Aufruf, sondern auch wie der Aufruf technisch zu erfolgen hat: HTTP-Methode, Aufbau des Request-Bodys und die zu erwartende Antwort (vgl. RICHARDSON, AMUNDSEN 2013, S. 138).

Im Sinne orthogonaler Spezifikationen können Profile und Media Types unabhängig voneinander spezifiziert werden. Übertragen auf die Konzepte von REST bedeutet dies, dass Profile Aussagen über Ressourcen unabhängig von ihrer konkreten Repräsentation treffen können. Wird ein neuer Media Type für eine Ressource eingeführt, erfordert dies somit nicht zwingend eine Anpassung des Profils (vgl. WILDE 2013, S. 3).

Ebenso ist es möglich, im Sinne des Interface-Segregation-Prinzips (vgl. MARTIN 1996) abhängig vom Anwendungsfall verschiedene Profile für eine Ressource zu definieren. Der Konsument ist so nicht gezwungen, ein umfassendes Profil zu implementieren, sondern kann aus einer Reihe von Profilen selektiv die implementieren, die für seine Anforderungen relevant sind (vgl. WILDE 2013, S. 4).

Die Verknüpfung einer Nachricht mit einem Profil erfolgt über einen Link, der mit dem Link-Relationstyp `profile` ausgezeichnet ist. *Listing 3* zeigt, wie der `Link`-Header genutzt werden kann, um in den Kopfzeilen der Nachricht und damit unabhängig vom verwendeten Media Type auf ein Profil zu verweisen.

```
2  HTTP/1.1 200 OK
3  Content-Type: application/json; charset=utf-8
4  Link: <https://example.hypercontract.org/profile>; rel="profile"
```

Listing 3: Verweis auf das Profil über einen `Link`-Header

Alternativ können Media Types um einen `profile`-Parameter ergänzt werden, der auf das Profil verweist (siehe *Listing 19*). Dieses Vorgehen ist jedoch nicht universell

einsetzbar, da dieser Parameter von der Spezifikation des jeweiligen Media Types definiert sein muss.[8]

```
1  HTTP/1.1 200 OK
2  Content-Type: application/hal+json;profile="
   https://example.hypercontract.org/profile"; charset=utf-8
```

Listing 4: Verweis auf das Profil über den `profile`-Parameter des JSON-HAL Media Types

Die dritte Variante ist die Einbettung in den Nachrichteninhalt über spezielle Hypermedia-Controls. Somit ist auch dieser Ansatz abhängig vom verwendeten Media Type. In HTML können Profile beispielsweise im Rahmen von Microdata-Attributen genutzt werden (vgl. NEVILE, BRICKLEY 2018).

Die Beispiele zeigen, wie Nachrichten auf Profile verweisen können, die ihren Inhalt beschreiben. Für die Aushandlung eines Contracts zwischen Anbieter und Konsument einer REST-Schnittstelle muss das Profil aber bereits im Rahmen der Content Negotiation berücksichtigt werden und kann somit nicht Teil des Nachrichteninhalts sein. Auch `Link`-Header werden bei der Inhaltsaushandlung nicht berücksichtigt. Nach aktuellem Stand der Spezifikationen wäre somit allenfalls der Einsatz des `profile`-Parameters am Media Type möglich, was die Auswahl nutzbarer Media Types stark einschränkt. SVENSSON, VERBORGH 2019 schlagen daher analog zu den Headern `Accept`, `Accept-Encoding` und `Accept-Language` zur Content Negotiation auf Basis von Media Type, Zeichenkodierung und Sprache den `Accept-Profile`-Header vor.[9] Mit diesem kann ein Client eine Repräsentation konform zu einem oder mehreren Profilen anfragen. Unterstützt der Server das Profil, enthält die Antwort einen entsprechenden `Content-Profile`-Header (siehe *Listing 5*). Ist dem Server das Profil nicht bekannt, antwortet er mit dem Statuscode `406 Not Acceptable` (vgl. ebd.).

```
1  GET / HTTP/1.1
2  Host: example.hypercontract.org
3  Accept: application/json
4  Accept-Profile: <https://example.hypercontract.org/profile>

5  HTTP/1.1 200 OK
6  Content-Type: application/json; charset=utf-8
7  Content-Profile: <https://example.hypercontract.org/profile>
```

Listing 5: Berücksichtigung des Profils bei der Content Negotiation über den `Accept-Profile`-Header

[8] Beispiele für Formate, die den Parameter unterstützen sind JSON-HAL, JSON-LD oder XHTML. In den Spezifikationen von JSON und HTML fehlt er hingegen (vgl. RICHARDSON, AMUNDSEN 2013, S. 136).

[9] Die Spezifikation der so genannten *Content Negotiation by Profile* ist zum Entstehungszeitpunkt dieser Arbeit noch ein Working Draft des W3C und daher kein offizieller Standard.

Identifiziert wird das Profil wie in den Beispielen zu erkennen über einen URI. Wie beim Einsatz von URIs für XML-Namensräume muss der URI eines Profils nicht dereferenzierbar sein (vgl. WILDE 2013, S. 4). Nichtsdestotrotz eröffnet sich dadurch die Möglichkeit, die Spezifikation des Profils als dessen Repräsentation über diesen URI bereitzustellen. Im Gegensatz zu Media Types kann die Angabe eines Profils demnach dazu genutzt werden, im Kontext einer Ressourcenrepräsentation auf die dazu passende Beschreibung zu verweisen.

Für die Spezifikation von Profilen gibt es jedoch wie auch im Falle des Media Types keine standardisierte, maschinenlesbare Beschreibungssprache (vgl. RICHARDSON, AMUNDSEN 2013, S. 155–156). Mit dem Resource Description Framework (RDF) hat das W3C allerdings eine Technologie entwickelt, mit welcher sich beliebige Informationen strukturiert und maschinenlesbar abbilden lassen, und dabei an existierende Konzepte des Webs anknüpft.

2.4 RDF und Linked Data

Das Resource Description Framework (RDF) gehört zum Technologiestack des Semantic Webs, einer vom W3C beschriebenen Erweiterung des traditionellen, dokumentenzentrierten Webs hin zu einem Web der Daten.

> *The Semantic Web is an attempt, largely, to map large quantities of existing data onto a common language so that the data can be analyzed in ways never dreamed of by its creators.*
>
> — BERNERS-LEE 2013

Hinter dem Semantic Web steht die Vision, dass das Web nicht nur von Menschen, sondern auch von Maschinen konsumiert werden kann. Strukturierte Daten sollen autonomen Agenten ermöglichen, eigenständig Schlussfolgerungen zu ziehen und Entscheidungen treffen zu können (vgl. BERNERS-LEE, HENDLER, LASSILA 2001).

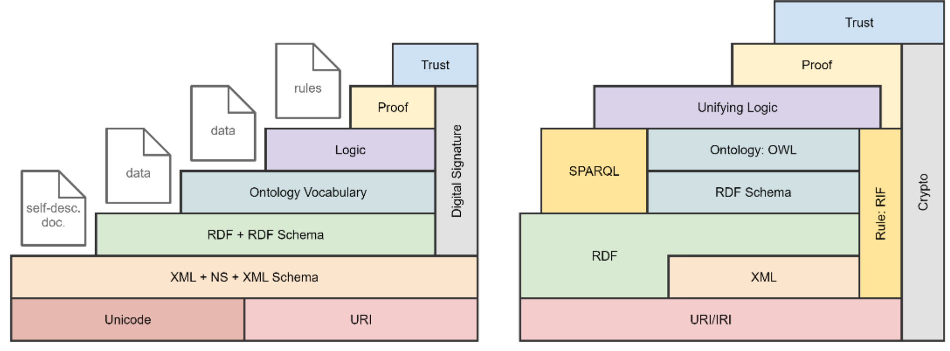


Abbildung 4: Der ursprüngliche Semantic Web Stack nach BERNERS-LEE 2000 und der erweiterte nach BRATT 2007

Der von BERNERS-LEE 2000 beschriebene Semantic Web Stack (siehe *Abbildung 4*) fußt auf etablierten Standards des Webs: Unicode als Zeichensatz, URIs als global eindeutige Identifier und XML als universelles Datenformat. Auf dieser Basis definiert RDF ein einfaches Datenmodell, in welchem Tripel bestehend aus Subjekt, Prädikat und Objekt, genutzt werden, um Aussagen über Ressourcen zu formulieren. Mit RDF Schema, ergänzt durch OWL (vgl. BRATT 2007), lassen sich wiederverwendbare Vokabulare und Taxonomien definieren, die für die Formulierung von RDF-Statements genutzt werden können. Mit SPARQL kommt 2008 noch eine RDF-basierte Abfragesprache hinzu. Zusammengenommen bilden diese Technologien die Voraussetzung für wesentliche Anforderungen an künstliche Intelligenz: Logik als Grundlage für automatisiertes Reasoning, Nachweisbarkeit und die Realisierung von Vertrauensbeziehungen.

Angesichts dieser Aussichten ist es nicht überraschend, dass die Forschungsbemühungen anfangs einen starken Fokus auf den oberen Teil des Semantic Web Stacks legten (vgl. PAGE, ROURE, MARTINEZ 2011, S. 22). Diese Vision ist jedoch bislang nicht Realität geworden und aktuelle Entwicklungen im Bereich von Machine Learning und AI finden weitgehend abseits der damals konzipierten Technologien statt. Bereits SHADBOLT, BERNERS-LEE, HALL 2006 wiesen darauf hin, dass neben allen Bestrebungen in Hinblick auf künstliche Intelligenz die grundlegenden Ideen des Semantic Webs nie realisiert wurden. Dies führte zur Veröffentlichung der vier Prinzipien von *Linked Data*:

(1) Use URIs as names for things
(2) Use HTTP URIs so that people can look up those names.
(3) When someone looks up a URI, provide useful information, using the standards (RDF, SPARQL)
(4) Include links to other URIs. so that they can discover more things.

(BERNERS-LEE 2006)

Während URIs im Kontext des Semantic Webs hauptsächlich als Identifier betrachtet wurden, rückt Linked Data ihre Funktion als Retrieval-Mechanismus wieder in den Mittelpunkt. Analog zum „traditionellen" Web sind Links ein wesentlicher Aspekt des *Web of Linked Data*:

The Semantic Web isn't just about putting data on the web. It is about making links, so that a person or machine can explore the web of data. With linked data, when you have some of it, you can find other, related, data.

— Berners-Lee 2006

Die (typisierte) Verlinkung von Ressourcen ist nicht nur ein zentrales Element von Webseiten, sondern mit HATEOAS auch ein Kernmerkmal von REST. Eine Betrachtung, inwiefern RDF und Linked Data für den Einsatz im Rahmen einer REST-basierten Architektur relevant sein können, liegt daher nahe.

2.4.1 Ressourcenbeschreibung mit RDF

RDF wurde ursprünglich als Metadatenmodell für die Beschreibung von Ressourcen im Web konzipiert (vgl. Lassila, Swick 1999). Im Zuge des Semantic Webs wurde es jedoch zum grundlegenden Modell für die Beschreibung von beliebigen identifizierbaren Konzepten weitergedacht (vgl. Berners-Lee, Hendler, Lassila 2001).

URIs sind hierbei von zentraler Bedeutung. Sie erlauben es, Konzepte auch über die Grenzen von Datenquellen hinaus eindeutig zu identifizieren. Im Kontext von RDF werden sie deshalb genutzt, um eindeutige, maschinenlesbare Aussagen über diese Ressourcen zu formulieren.

Aussagen werden in RDF als Tripel formuliert. Da RDF-Statements in ihrem Aufbau Hauptsätzen gleichen, werden die Bestandteile als Subjekt, Prädikat und Objekt bezeichnet. Subjekt und Prädikat werden über URIs identifiziert, als Objekt sind darüber hinaus auch Literale erlaubt (vgl. Manola, Miller, McBride 2014).

```
1   <http://example.org/Indiana_Jones> <http://example.org/portrayer>
    <http://example.org/Harrison_Ford> .
2   <http://example.org/Harrison_Ford> <http://example.org/birthPlace>
    <http://example.org/Chicago> .
3   <http://example.org/Harrison_Ford> <http://example.org/birthDate>
    "1942-07-13"^^<http://www.w3.org/2000/01/rdf-schema#Date> .
```

Listing 6: Beispiele für RDF-Statements

In den RDF-Statements in *Listing 6* identifiziert der URI `http://example.org/Indiana_Jones` die fiktive Figur *Indiana Jones* und

`http://example.org/Harrison_Ford` den Schauspieler, der sie verkörpert. Über beide Ressourcen werden Aussagen getroffen:

(1) Indiana Jones wird portraitiert von Harrison Ford.
(2) Harrison Ford ist geboren in Chicago.
(3) Harrison Ford ist geboren am 13. Juli 1942.

Für eine übersichtlichere Darstellung werden die wiederkehrenden Teile der URIs in der Regel durch Präfixe ersetzt. In *Listing 7* steht `ex` für das redundante `http://example.org/` und `xsd` verkürzt die sperrige Angabe des XML-Schema-Datentyps.[10]

```
1  @prefix ex: <http://example.org/> .
2  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3
4  ex:Indiana_Jones ex:portrayer ex:Harrison_Ford .
5  ex:Harrison_Ford ex:birthPlace ex:Chicago .
6  ex:Harrison_Ford ex:birthDate "1942-07-13"^^xsd:date .
```

Listing 7: Verwendung von Namensraumpräfixen zur kompakteren Darstellung von URIs in RDF-Tripeln

Die Prädikate, die die Beziehung zwischen Ressourcen im Subjekt und den Ressourcen oder Literalen im Objekt charakterisieren, sind eine Sonderform der Ressource und werden als Eigenschaft bezeichnet. Die Tatsache, dass es sich bei ihnen auch um Ressourcen handelt, erlaubt die Formulierung von Aussagen über Eigenschaften.

```
1  @prefix ex: <http://example.org/> .
2  @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>
3
4  ex:birthDate rdfs:label "Geburtsdatum"@de
5  ex:birthDate rdfs:label "birth date"@en
6  ex:birthDate rdfs:comment "The date something or someone was
   born."@en
```

Listing 8: RDF-Statements über eine Eigenschaft

In *Listing 8* erhält die `ex:birthDate`-Eigenschaft mit Hilfe von `rdfs:label` und `rdfs:comment` natürlichsprachige Bezeichner und eine kurze Definition ihrer Bedeutung. Das Beispiel zeigt auch, wie Zeichenketten mit einer Sprachangabe versehen werden können.

[10] Eine vollständige Liste der in dieser Arbeit verwendeten Namensraumpräfixe befindet sich in Anhang *B Namensraumpräfixe*.

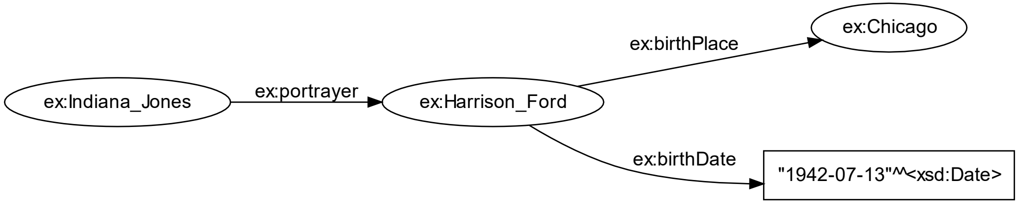


Abbildung 5: Darstellung eines RDF-Graphen

Abbildung 5 visualisiert die Tripel aus *Listing 6* in einer für RDF üblichen Notation. Ressourcen werden hierbei als Ellipsen dargestellt, Eigenschaften als Pfeile und Literale nehmen die Form von Rechtecken an. Die Darstellung verdeutlicht, dass das Datenmodell von RDF im Wesentlichen ein kantenbeschrifteter, gerichteter Graph ist.

2.4.2 Serialisierung von RDF-Daten

RDF ist unabhängig von einer konkreten Serialisierung spezifiziert. Für die textuelle Repräsentation haben sich daher je nach Anwendungszweck verschiedene Formate etabliert. Die in diesem Abschnitt bislang gezeigten Beispiele verwenden die kompakte Turtle-Syntax (vgl. Beckett u. a. 2014). Weitere Beispiele sind die mit Turtle verwandten Serialisierungen N3 und N-Triples oder das XML-basierte RDF/XML (vgl. Berners-Lee, Connolly 2011; Carothers, Seaborne 2014; Gandon, Schreiber 2014).

Ein weiteres Format ist JSON-LD, welches RDF-Graphen als JSON-Dokument kodiert. Der wesentliche Vorteil gegenüber anderen Serialisierungsarten ist die Interoperabilität mit bestehenden, JSON-kompatiblen Anwendungen. Mit Hilfe eines JSON-LD-Kontextes können die Eigenschaften eines regulären JSON-Dokuments in URIs übersetzt werden und erlauben so die Anreicherung bestehender JSON-Daten um semantische Informationen (vgl. Sporny, Kellogg, Lanthaler 2014).

```
1  {
2    "name": "Indiana Jones",
3    "portrayer": {
4      "name": "Harrison Ford",
5      "birthPlace": "Chicago",
6      "birthDate": "1942-07-13"
7    }
8  }
```

Listing 9: Abbildung von Daten als JSON-Struktur

Listing 9 drückt das zuvor beschriebene Beispiel als einfache JSON-Struktur aus und ergänzt Bezeichnungen in Form von `name`-Eigenschaften.

```
1  {
2    "@context": {
3      "@base": "http://example.org/",
4      "ex": "http://example.org/",
5      "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
6      "xsd": "http://www.w3.org/2001/XMLSchema#",
7      "name": "rdfs:label",
8      "portrayer": {
9        "@id": "ex:portrayer",
10       "@type": "@id"
11     },
12     "birthPlace": {
13       "@id": "ex:birthPlace",
14       "@type": "@id"
15     },
16     "birthDate": {
17       "@id": "ex:birthDate",
18       "@type": "xsd:date"
19     }
20   },
21   "@id": "ex:Indiana_Jones",
22   "name": "Indiana Jones",
23   "portrayer": {
24     "@id": "ex:Harrison_Ford",
25     "name": "Harrison Ford",
26     "birthPlace": "Chicago",
27     "birthDate": "1942-07-13"
28   }
29 }
```

Listing 10: Abbildung eines RDF-Graphen im JSON-LD-Format

Listing 10 ergänzt diese Struktur um JSON-LD-spezifische Eigenschaften, ohne die ursprünglichen Daten (im Beispiel hervorgehoben) zu verändern. Über `@id` erhalten die Objekte eine Identität in Form einer URI. Der in `@context` definierte JSON-LD-Kontext enthält alle Informationen, um die ursprüngliche JSON-Struktur in einen RDF-Graphen zu überführen. `@base` erweitert den `birthPlace`-Wert `"Chicago"` zu einer URI. Die Eigenschaften `name`, `portrayer`, `birthPlace` und `birthDate` werden über `@id` in RDF-Eigenschaften übersetzt und mit `@type` typisiert. Der `@id`-Wert für `@type` signalisiert, dass die Eigenschaft eine andere Ressource verweist, alle anderen Werte definieren einen literalen Eigenschaftswert.

Der JSON-LD-Kontext muss nicht zwangsläufig Teil des JSON-Dokuments sein, sondern kann auch über die einen Link-Header mit dem Nachrichteninhalt verknüpft werden (vgl. SPORNY, KELLOGG, LANTHALER 2014).

RDF verhält sich daher nicht nur agnostisch gegenüber der Wissensdomäne, in welcher es eingesetzt wird, sondern kann über die Wahl einer passenden Serialisierung auch im Umfeld von nicht-RDF-kompatiblen Systemen genutzt werden. RDF bietet somit ein universelles, sprachunabhängiges Datenmodell.

2.4.3 Vokabulare mit RDFS und OWL

Damit Anbieter und Konsument von RDF-Daten ein gemeinsames Verständnis für die Bedeutung der Daten besitzen, müssen sie die gleiche Sprache sprechen. RDF-Statements sollten daher mit Hilfe von Vokabularen formuliert werden, die von beiden Seiten verstanden werden.

Eine Grundlage bieten populäre Vokabulare wie FOAF zur Beschreibung von Personen und ihren Beziehungen, das Dublin Core Schema zur Angabe von Metadaten für digitalen Ressourcen oder das durch von Suchmaschinenbetreibern getriebene schema.org-Projekt (vgl. BRICKLEY, MILLER 2014; DCMI 2012; SCHEMA.ORG). Darüber hinaus existieren domänenspezifische Vokabulare, deren Ziel es ist, eine Wissensdomäne möglichst umfassend abzubilden.[11]

Sollten existierende Vokabulare nicht dafür geeignet sein, die Sachverhalte einer bestimmten Wissensdomäne adäquat abzubilden, können mit Hilfe von RDF Schema (RDFS) und der Ontology Web Language (OWL) neue Vokabulare in RDF definiert werden. RDFS dient der Beschreibung von Datenstrukturen in Form von Klassen und Eigenschaften (vgl. BRICKLEY, GUHA 2014). OWL kann darauf aufbauend die formale Semantik dieses Datenschemas festlegen (vgl. MOTIK, PATEL-SCHNEIDER, PARSIA 2012). Mit OWL erstellte komplexe und oftmals auch streng formale Vokabulare werden häufig als Ontologie bezeichnet. Es existiert jedoch keine klare Definition, was Vokabulare von Ontologien unterscheidet (vgl. W3C 2019).[12]

Abbildung 6 definiert `ex:Actor` und `ex:FictionalCharacter` als Unterklassen von `ex:Person`. Zusätzlich werden die Eigenschaften `ex:birthPlace`, `ex:birthDate` und `ex:portrayer` beschrieben. Mit `rdfs:domain` werden sie den Klassen `ex:Person` beziehungsweise `ex:FictionalCharacter` zugeordnet. Der Wertebereich der Eigenschaften wird über `rdfs:range` angegeben. Die Definition von `ex:Actor` und `ex:FictionalCharacter` als Unterklassen von `ex:Person` bedeutet, dass sie alle Eigenschaften von `ex:Person` erben.

---

[11] In dieser Arbeit wird größtenteils bewusst auf die Verwendung existierender Vokabulare verzichtet, um nicht zu viele Vorkenntnisse vom Leser zu erwarten. Für die zukünftige Weiterentwicklung des Projekts ist die Evaluierung verwandter Vokabulare hinsichtlich einer möglichen Wiederverwendbarkeit jedoch ein sinnvoller nächster Schritt.

[12] Für automatisiertes Reasoning, wie es in der ursprünglich formulierten Vision des Semantic Webs vorgesehen war, ist eine umfassende Beschreibung der formalen Semantik unerlässlich. Im Rahmen dieser Arbeit ist dieser Aspekt jedoch zu vernachlässigen, weshalb nur einige grundlegende Elemente von OWL zur Anwendung kommen. Im Folgenden wird daher ausschließlich der Begriff „Vokabular“ verwendet.

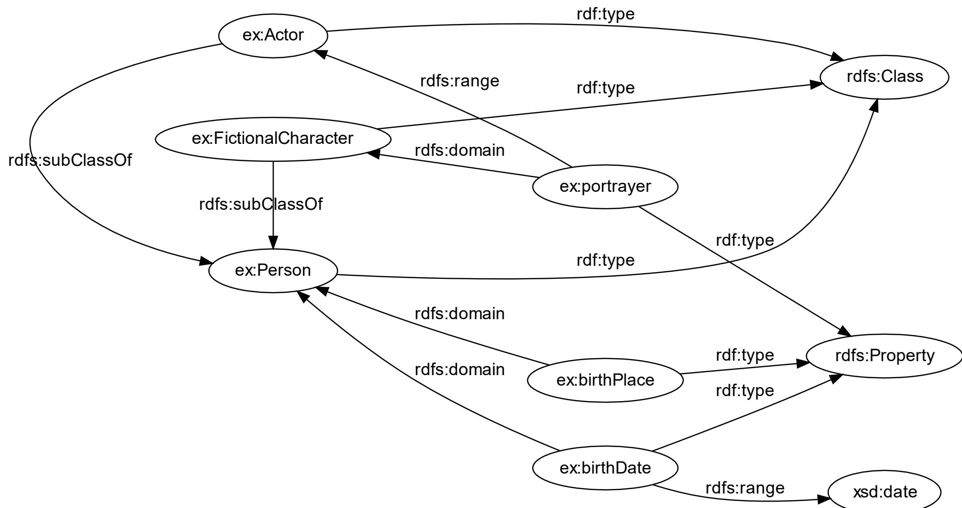


Abbildung 6: Darstellung eines RDFS-basierten Vokabulars

Eine menschenverständliche Beschreibung der Klassen und Eigenschaften kann wie im vorangegangenen Abschnitt gezeigt über die Eigenschaften `rdfs:label` und `rdfs:comment` erfolgen.

Die Verknüpfung von RDF-Daten und Vokabular erfolgt dabei nicht nur über die Verwendung von Eigenschaften. Mit `rdf:type` lassen sich die Konzepte in den Daten auch den passenden Klassen zuordnen (vgl. MANOLA, MILLER, MCBRIDE 2014). *Abbildung 7* verdeutlicht dabei auch, wie über Reasoning geschlussfolgert werden könnte, dass sowohl Harrison Ford als auch Indiana Jones Personen sind.

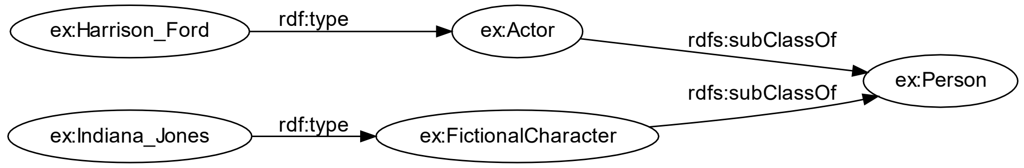


Abbildung 7: Zuordnung von Ressourcen zu Klassen über `rdf:type`

RDFS-Klassen weisen im Vergleich zu ihren Pendants aus objektorientierten Programmiersprachen zwei wesentliche Unterschiede auf. Zum einen werden Klassen und Eigenschaften in RDFS unabhängig voneinander definiert. Es ist daher möglich, dass mehrere disparate Klassen sich eine Eigenschaft teilen. Darüber hinaus trifft RDFS keine Aussage über die Kardinalität von Eigenschaften (vgl. BRICKLEY, GUHA 2014). Einer `ex:Person` im Sinne des in *Abbildung 6* beschriebenen Vokabulars könnte man daher über `ex:birthPlace` zwei verschiedene Geburtsorte zuordnen, auch wenn dies aus semantischer Sicht vermutlich nicht sinnvoll wäre.

Einschränkungen dieser Art sind mit Hilfe von OWL möglich, indem OWL-spezifische Unterklassen von `rdfs:Property` zur Typisierung der Eigenschaften genutzt werden. In *Abbildung 8* werden die Eigenschaften `ex:birthPlace` und

`ex:birthDate` als `owl:FunctionalProperty` typisiert, womit diese pro Klasseninstanz nur einen Wert annehmen können. Darüber hinaus erfolgt eine Einschränkung des Wertebereichs der Eigenschaften über `owl:ObjectProperty` und `owl:DatatypeProperty`. So ist auch ohne Berücksichtigung von `rdfs:range` klar definiert, dass `ex:birthPlace` stets eine Ressource referenziert, während `ex:birthDate` auf einen literalen Wert verweist. Da beide Property-Varianten Unterklassen der von `rdfs:Property` sind, bleibt die bisherige die Typisierung der Eigenschaften als `rdfs:Property` bestehen (vgl. MOTIK, PATEL-SCHNEIDER, PARSIA 2012).

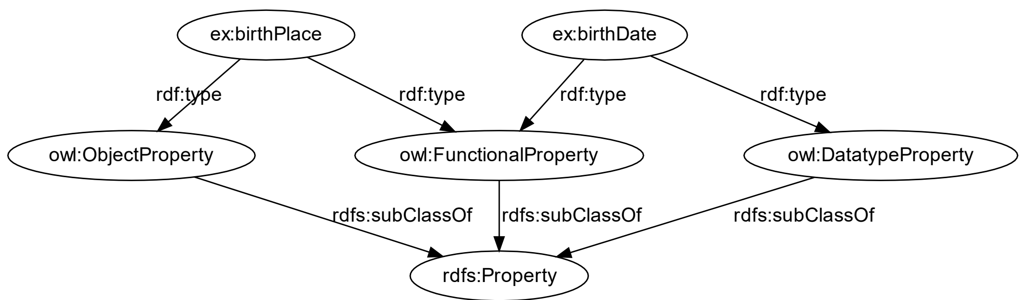


Abbildung 8: Definition von Eigenschaften mit OWL

Die Möglichkeiten von OWL gehen weit über die hier beschriebenen Beispiele hinaus. Für das Verständnis der vorliegenden Arbeit sind die vorgestellten Konzepte jedoch ausreichend. Wichtig ist die Erkenntnis, dass RDF-Daten für sich genommen ebenso wenig semantische Aussagekraft besitzen wie JSON- oder XML-Dokumente ohne zusätzlichen Kontext. Ein semantisches Verständnis für die in RDF abgebildeten Informationen entsteht erst durch die Nutzung von Vokabularen, die von allen Beteiligten verstanden werden. Sie können daher auch als Contract zwischen Anbieter und Konsument von RDF-Daten betrachtet werden (vgl. SEGARAN, TAYLOR, EVANS 2009, 128f.).

2.4.4 REST und Linked Data

Das Semantic Web lebte lange Zeit in einer Parallelwelt zum klassischen Web. Erst mit Linked Data fand eine Rückbesinnung auf das konzeptionelle Fundament statt, das auch die Basis für REST bildet (vgl. RICHARDSON, AMUNDSEN 2013, S. 272). Die Gemeinsamkeiten zeigen sich bei der Betrachtung der vier Prinzipien von Linked Data (siehe *2.4 RDF und Linked Data*).

Use URIs as names for things: Übertragen auf REST entspricht dies dem Prinzip identifizierbarer Ressourcen (siehe *2.3.2.1 Identifizierbare Ressourcen*). Da REST fast ausschließlich im Umfeld von HTTP-basierten Applikationen zum Einsatz kommt,

werden diese über URLs identifiziert, was wiederum dem zweiten Linked-Data-Prinzip Rechnung trägt.

Use HTTP URIs so that people can look up those names: Aus Sicht von RDF ist ein URI zunächst einmal nur ein Identifier. Es gibt keine Garantie, dass sich dahinter auch eine abfragbare Repräsentation der Ressource verbirgt. Dies entspricht der Herangehensweise, die RICHARDSON, AMUNDSEN 2013 als *Description Strategy* bezeichnen. Eine oder mehrere Ressourcen werden hierbei durch die Repräsentation einer anderen Ressource beschrieben. Konzept und Beschreibung des Konzepts werden somit voneinander getrennt und sind individuell identifizierbar (vgl. SAUERMANN, CYGANIAK 2008). Einer der Vorteile dieses Vorgehens ist die Möglichkeit, auch solche Ressourcen beschreiben zu können, die außerhalb des eigenen Zugriffs liegen oder deren URI-Schema keine Abfrage über das Web erlaubt (vgl. RICHARDSON, AMUNDSEN 2013, S. 263ff.). Angesichts dieser Entkopplung von Ressourcen und ihren Beschreibungen stellt sich jedoch die Frage, über welchen Mechanismus die Beschreibung einer Ressource ermittelt werden kann. Die Forderung nach dereferenzierbaren HTTP-URIs bedeutet daher eine Neuausrichtung des Semantic Webs hin zur von REST präferierten *Representation Strategy*, die sich auch im nächsten Prinzip widerspiegelt (vgl. ebd., S. 272f.).

When someone looks up a URI, provide useful information, using the standards: Die Abfrage einer Ressource soll eine Repräsentation mit dem Zustand dieser Ressource liefern (siehe *2.3.2.2 Abfrage und Manipulation mittels Repräsentationen*). BERNERS-LEE 2006 nennt hier zwar explizit RDF und SPARQL, wichtig ist jedoch vielmehr, dass etablierte Standards verwendet werden und Clients via Content Negotiation die passende Repräsentation aushandeln können (vgl. BERNERS-LEE 2009).

Include links to other URIs. so that they can discover more things: Nicht zuletzt ist das Konzept von Links zentrales Element beider Ansätze. Der Verweis auf verwandte und weiterführende Ressourcen unterscheidet nicht nur REST von anderen Schnittstellenarchitekturen (vgl. FIELDING 2008), sondern auch Linked Data von den ursprünglichen Bemühungen rund um das Semantic Web (vgl. PAGE, ROURE, MARTINEZ 2011, S. 22–23).

Angesichts dieser konzeptionellen Nähe von REST und Linked Data lohnt sich eine Betrachtung, inwiefern der Einsatz semantischer Technologien für REST-konforme Applikationen sinnvoll sein kann. Ein naheliegender Ansatz ist die Nutzung RDF-basierter Repräsentationsformate (vgl. TILKOV U. A. 2015, 107ff.; WEBBER, PARASTATIDIS, ROBINSON 2010, 358ff.; LANTHALER 2014). Mit ihnen wären keine

umfangreichen Media-Type-Spezifikationen mehr erforderlich, da die verwendeten RDF-Vokabulare diese Aufgabe zum Teil übernehmen.

Ein anderes Anwendungsgebiet für semantische Technologien ist die Beschreibung der Schnittstelle (vgl. RICHARDSON, AMUNDSEN 2013, S. 272). Sowohl Media Types als auch Profile lassen offen, in welcher Form sie spezifiziert werden sollen. RDF Schema und OWL können hier einen Teil der Lösung darstellen.

3 Existierende Lösungen

Die Eigenschaften REST-konformer Applikationen stellen gängige Lösungen zur Beschreibung von Web-APIs vor eine Herausforderung. Dennoch existiert eine Vielzahl an Projekten, die sich dieser Aufgabe widmen. Mit OpenAPI, hRESTS, ALPS und Hydra werden im Folgenden vier verschiedene Ansätze vorgestellt, die diesen Herausforderungen auf unterschiedliche Weise begegnen. Die Stärken und Schwächen der vorgestellten Technologien werden anhand von Kriterien herausgearbeitet, die auf der Zielsetzung dieser Arbeit basieren:

(1) *Vereinbarkeit mit Webarchitektur und REST*,
(2) *Integration in REST-konforme Web-APIs*,
(3) *Verfügbarkeit im Kontext der Nutzung*,
(4) *Aufbereitung für Mensch und Maschine*,
(5) *Beschreibung der Schnittstellenmechanik* und
(6) *Beschreibung der fachlichen Semantik*

(siehe *1.2 Zielsetzung*).

Zusammen mit der theoretischen Betrachtung im vorherigen Kapitel bilden diese Erfahrungen aus der Praxis die Grundlage für Erarbeitung eines Konzepts zur Beschreibung von REST-basierten Webschnittstellen, das der formulierten Zielsetzung gerecht wird.

3.1 OpenAPI

Die populärste Lösung zur Beschreibung von Web-APIs ist derzeit die *OpenAPI Specification* (vgl. Miller u. a. 2018). Zentrales Element ist eine in JSON oder YAML verfasste Schnittstellendefinition. Ausgehend von URL-Templates, die die exponierten URLs einer Web-API beschreiben, werden die erlaubten HTTP-Operationen auf diesen Endpunkten dokumentiert. Dies umfasst unter anderem erlaubte Abfrageparameter, Aufbau des Request Bodys und die zu erwartenden HTTP-Statuscodes und Response Bodys.

Listing 11 zeigt beispielhaft, wie die Beschreibung einer Web-API aussehen könnte, die unter `https://example.hypercontract.org/products` eine Liste von Produkten bereitstellt, welche über POST-Anfragen an `/shoppingCart/items` dem Warenkorb hinzugefügt werden können.

```
1  openapi: "3.0.0"
2  info:
3    version: 2.0.0
4    title: hypershop
5    description: "Example App for hypercontract"
6    license:
7      name: MIT
8  servers:
9    - url: https://example.hypercontract.org
10 paths:
11   "/products":
12     get:
13       operationId: "getProducts"
14       summary: "Returns a list of all Products"
15       responses:
16         "200":
17           description: "A list of all Products"
18           content:
19             application/json:
20               schema:
21                 $ref: "#/components/schemas/Products"
22           links:
23             addToShoppingCart:
24               operationId: addToShoppingCart
25   "/shoppingCart/items":
26     post:
27       operationId: addToShoppingCart
28       summary: "Adds a Product to the Shopping Cart"
29       requestBody:
30         content:
31           application/json:
32             schema:
33               $ref: "#/components/schemas/AdditionToShoppingCart"
34       responses:
35         "201":
36           description: "Product successfully added to the Shopping
   Cart."
37           headers:
38             Location:
39               description: "The URL of the Shopping Cart."
40               schema:
41                 type: string
42                 format: uri
43 components:
44   schemas:
45     Products:
46       description: "A list of Products."
47       type: array
48       items:
49         $ref: "#/components/schemas/Product"
50     Product:
51       description: "A Product that can be ordered."
52       type: object
53       properties:
54         _id:
55           description: "The Product's unique identifier."
56           type: string
57         productName:
58           description: "The name of the Product."
59           type: string
60         productDescription:
61           description: "A description of the Product's features and
   qualities."
62           type: string
```

```
63          price:
64            description: "The price per item."
65            type: number
66            minimum: 0
67          image:
68            description: "A picture of the Product."
69            type: string
70            format: uri
71      AdditionToShoppingCart:
72        description: "Represents all information necessary to add a
    product to the shopping cart."
73        type: object
74        properties:
75          product:
76            $ref: "#/components/schemas/Product/properties/_id"
77          quantity:
78            description: "The number of items to be ordered."
79            type: number
80            minimum: 1
```

Listing 11: Beispiel für eine Schnittstellenbeschreibung mit OpenAPI

Der Aufbau der Nachrichteninhalte kann über wiederverwendbare Schemas beschrieben werden – im Beispiel sind dies `Products`, `Product` und `AdditionToShoppingCart`. Hierfür kommt eine Variante von JSON Schema zum Einsatz (vgl. STURGEON 2018; WRIGHT, ANDREWS 2018). Mit der `links`-Eigenschaft bietet OpenAPI seit der Version 3.0 auch die Möglichkeit, Links zu spezifizieren. Diese basieren jedoch nicht auf der Web-Linking-Spezifikation (vgl. NOTTINGHAM 2010) und müssen auch nicht Teil der Serverantwort sein (vgl. SWAGGER 2018).

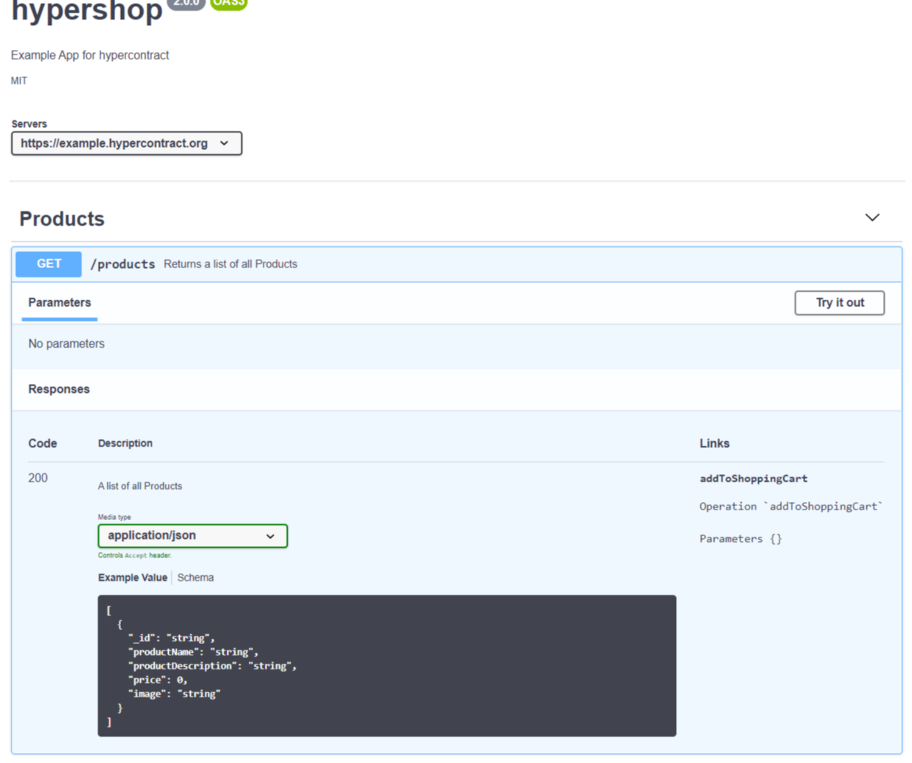


Abbildung 9: Beispiel für die aus einer OpenAPI-Spezifikation generierte Dokumentation

Neben den umfassenden Dokumentationsmöglichkeiten ist die Popularität von OpenAPI auch auf das Ökosystem aus Werkzeugen zurückzuführen, das sich rund um die API-Spezifikation entwickelt hat (vgl. TRASK, STURGEON 2019). Neben Code-Generierung für verschiedene Programmiersprachen und Frameworks ist hier insbesondere die automatische Generierung einer entwicklerfreundlichen API-Dokumentation erwähnenswert (siehe *Abbildung 9*).

Vereinbarkeit mit Webarchitektur und REST: Wie in *2.3.3 Contracts in REST* bereits erläutert läuft das grundsätzliche Konzept von OpenAPI, eine Webschnittstelle ausgehend den URLs ihrer Ressourcen zu beschreiben, dem REST-Prinzip von *Hypermedia as the Engine of Application State* zuwider. Da dem Client die URLs der API somit ohnehin bekannt sind, ist auch die Spezifikation von möglichen Links in vielen Fällen überflüssig und wenig flexibel. Die mangelnde Unterstützung des Features durch OpenAPI-basierte Werkzeuge verdeutlicht dies.

Grundsätzlich ermöglicht OpenAPI eine umfassende Beschreibung von HTTP-basierten APIs, die daraus entstehende Spezifikation führt jedoch stets zu einer engeren Kopplung von Client und Server, als dies im Rahmen einer REST-API notwendig wäre. Verschärft wird dieses Problem durch die auf OpenAPI aufbauenden Werkzeuge zur Code-Generierung und automatisierten Validierung der Spezifikation. Eine evolutionäre, interoperable Weiterentwicklung der API wird erschwert, wenn jede Änderung eine neue Version des generierten Clients erfordert oder automatisierte Validierung von Schemas gefährdet (vgl. WILDE, PAUTASSO 2011, S. 6).

Integration in REST-konforme Web-APIs: Theoretisch lassen sich existierende Web-APIs, die nach den Prinzipien von REST gestaltet sind und JSON oder XML als Nachrichtenformat verwenden, mit Hilfe von OpenAPI vollständig beschreiben. Es gelten jedoch die zuvor genannten Einschränkungen in Hinblick auf Hypermedia-getriebene Schnittstellen. Da die Spezifikation der API unabhängig von der Implementierung erfolgen kann, ist eine Integration in bestehende Applikationen in den meisten Fällen möglich.

Verfügbarkeit im Kontext der Nutzung: OpenAPI sieht keine direkte Verknüpfung von Nachrichten oder Nachrichtenteilen und deren Beschreibung in der API-Spezifikation vor. Ein menschlicher wie maschineller Nutzer muss ausgehend von einer Nachricht somit eigenständig den passenden Teil der Spezifikation ausfindig machen.

Aufbereitung für Mensch und Maschine: Die JSON- bzw. YAML-basierte Spezifikation der API erlaubt eine einfache automatisierte Auswertung, was sich im Ökosystem

von Werkzeugen, die sich um OpenAPI herum entwickelt hat, widerspiegelt. Die Aufbereitung als menschenlesbare Dokumentation ist über auf OpenAPI aufbauende Lösungen möglich und ein wesentliches Argument für den Einsatz von OpenAPI (vgl. TILKOV U. A. 2015, 165ff.).

Beschreibung der Schnittstellenmechanik: Die Abbildung der Protokollsemantik über die Dokumentation von Methoden, Statuscodes und Media Types ist ein Kernfeature von OpenAPI. Die syntaktische Beschreibung der Nachrichteninhalte ist dank Integration von JSON Schema ebenfalls eine Stärke der Lösung. Die Abhängigkeit zum JSON-Schema-Standard hat jedoch zur Folge, dass nur JSON- und XML-basierte Nachrichteninhalte unterstützt werden. Zwar können auch andere Media Types als Nachrichtenformat spezifiziert werden, eine formale Beschreibung der Inhalte durch eine andere Schemasprache ist jedoch nicht möglich.

Beschreibung der fachlichen Semantik: OpenAPI erlaubt auf vielen Ebenen die Nutzung von `description`-Elementen zur natürlichsprachigen Beschreibung von Auszeichnungselementen, Nachrichten und Operationen. Dadurch ist eine kleinteilige Dokumentation der fachlichen Semantik möglich. Durch den Einsatz von Querverweisen via `$ref` können zudem auch Äquivalenzen einzelner Eigenschaften oder Modelle über HTTP-Operationen hinweg ausgedrückt werden.

Der Contract, der auf Grundlage einer OpenAPI-basierten Spezifikation entsteht, ist restriktiver als er sein müsste. Die Popularität von OpenAPI zeigt jedoch, dass dies in vielen Fällen kein Ausschlussgrund ist. Im Vergleich zu einem impliziten Contract ist ein restriktiver Contract die attraktivere Option.

Blickt man über die Schwächen in Hinblick auf REST-konforme APIs hinweg, zeigt sich anhand von OpenAPI, welche Möglichkeiten eine maschinenlesbare API-Spezifikation als Ausgangspunkt bietet. Ein durchdachtes Beschreibungsmodell ermöglicht nicht nur eine benutzerfreundliche, menschenlesbare Dokumentation der Schnittstelle, sondern auch eine kleinteilige, detaillierte Abbildung der fachlichen Semantik.

3.2 hRESTS

hRESTS ist ein RDF-basiertes Vokabular zur Annotation von HTML-basierten API-Dokumentationen. Ausgangsbasis ist somit nicht eine maschinenlesbare Spezifikation, sondern eine oftmals ohnehin vorhandene menschenlesbare Dokumentation. Die Auszeichnung der HTML-Elemente kann mit verschiedenen Annotations-Standards erfolgen (vgl. KOPECKÝ, GOMADAM, VITVAR 2008, 619ff.).

```
1  <section about="shop:shoppingCart" typeof="hr:Service"
2      xmlns:shop="https://example.hypercontract.org/profile/"
3      xmlns:hr="http://www.wsmo.org/ns/hrests#"
4      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
5
6    <h1 property="rdfs:label">Shopping Cart</h1>
7
8    <ul rel="hr:hasOperation">
9      <li about="shop:addToShoppingCart" typeof="hr:Operation">
10       <h2 property="rdfs:label">Add to Shopping Cart</h2>
11       <p property="rdfs:comment">Adds a Product to the Shopping
   Cart.</p>
12
13       <code>
14         <span property="hr:hasMethod">POST</span>
15         <span property="hr:hasAddress" datatype="hr:URITemplate">
   https://example.hypercontract.org/shoppingCart/items</span>
16       </code>
17
18       <div rel="hr:hasInputMessage">
19         <h3>Request</h3>
20         <code>
           { product: '01c31228-d5ba-44b3-84b1-7cccc7b9f707',
   quantity: 1 }
         </code>
21
22         <ul about="shop:AdditionToShoppingCart"
   typeof="hr:Message">
23           <li rel="hr:hasMandatoryPart">
24             <span about="shop:product" typeof="hr:MessagePart">
25               <code property="rdfs:label">product</code>
26               <span property="rdfs:comment">The Product's unique
   identifier.</span>
27             </span>
28           </li>
29           <li rel="hr:hasMandatoryPart">
30             <span about="shop:quantity" typeof="hr:MessagePart">
31               <code property="rdfs:label">quantity</code>
32               <span property="rdfs:comment">The quantity that is
   added to the Shopping Cart.</span>
33             </span>
34           </li>
35         </ul>
36       </div>
37
38       <div rel="hr:hasOutputMessage">
39         <h3>Response</h3>
40         <h4><code>201 Created</code></h4>
41         <p>Product successfully added to the Shopping Cart</p>
42       </div>
43     </li>
44   </ul>
45 </section>
```

Listing 12: Annotation einer HTML-basierten API-Dokumentation mit hRESTS

Listing 12 greift erneut das Beispiel aus *3.1 OpenAPI* auf und beschreibt das Hinzufügen eines Produkts zum Warenkorb in Form einer HTML-Dokumentation. Zusätzlich sind Elemente mit Hilfe des Annotations-Standards RDFa ausgezeichnet (im Beispiel hervorgehoben) (vgl. HERMAN U. A. 2015). Über so genannte Distiller

lassen sich diese Informationen als RDF-Daten extrahieren und auf diese Weise maschinell verarbeiten (vgl. McCarron u. a. 2015).

Das Modell von hRESTS orientiert sich an der *Web Services Architecture* des W3C und beschreibt REST-APIs daher in Form von Services und den von ihnen angebotenen Operationen – in diesem Fall der `shop:ShoppingCart`-Service und die zugehörige `shop:addToShoppingCart`-Operation. Über die Eigenschaften `hr:hasMethod`, `hr:hasAddress`, `hr:hasInputMessage` und `hr:hasOutputMessage` werden HTTP-Methode, URL sowie Request und Response Body ausgezeichnet. Die Bestandteile der ein- und ausgehenden Nachrichten werden als `hr:MessageParts` definiert (vgl. Kopecký, Gomadam, Vitvar 2008).

Das Datenmodell von hRESTs ist einfach gehalten und beschränkt sich auf die im Beispiel enthaltenen Sprachelemente.[13] Für detailliertere API-Spezifikationen wird auf weiterführende Standards wie SAWSDL und WSMO-Lite verwiesen (vgl. Fensel u. a. 2010; Farrell, Lausen 2007).

```
1  <li rel="hr:hasMandatoryPart">
2    <span about="shop:product" typeof="hr:MessagePart">
3      <code property="rdfs:label">product</code>
4      <span property="rdfs:comment">
5        The <a rel="sawsdl:modelReference"
   href="https://example.hypercontract.org/profile/Product">
   Product</a>'s unique identifier.
6      </span>
7      (<a rel="sawsdl:loweringSchemaMapping"
   href="https://example.hypercontract.org/mapping/Product.xsparql">Sc
   hema-Mapping</a>)
8    </span>
9  </li>
```

Listing 13: SAWSDL-Mapping eines Nachrichtenteils auf eine applikationsspezifische Ontologie

In *Listing 13* wird für die Dokumentation des `product`-Nachrichtenteils über `sawsdl:modelReference` das korrespondierende Konzept in einer applikationsspezifischen Ontologie referenziert. `sawsdl:loweringSchemaMapping` verweist auf ein XSPARQL-Dokument, das die Serialisierung des Nachrichtenteils als XML beschreibt (vgl. Farrell, Lausen 2007; Polleres u. a. 2009).

Vereinbarkeit mit Webarchitektur und REST: Technologisch orientiert sich hRESTS klar an den Standards des W3C. Die Definition als RDF-Vokabular ermöglicht neben der Unterstützung verschiedener Annotations-Standards auch die Nutzung im Kontext anderer RDF-basierter Anwendungen. Die Offenheit von RDF fördert

[13] Genau genommen sind schon die `hr:MessagePart`-Klasse und die `hr:hasMandatoryPart`-Eigenschaft nicht Teil des hRESTS-Vokabulars, da sie nur in Kopecký u. a. 2011 vorgestellt werden, nicht aber in der Ontologie definiert sind, die über den URI des Vokabulars bereitgestellt wird.

zudem die Erweiterbarkeit der Lösung. Dies ist allerdings auch erforderlich, da hRESTS für sich genommen nur wenige grundlegende Begriffe für die Beschreibung von Web-APIs bietet. Eine differenzierte Spezifikation von HTTP-Headern, Statuscodes und Media Types ist mit ihnen nicht möglich. Hierfür wird auf andere W3C-Standards verwiesen, die mehr noch als hRESTS selbst konzeptionell in der Tradition service-orientierter Webservice-Architekturen stehen (vgl. FARRELL, LAUSEN 2007; FENSEL U. A. 2010). Statt miteinander verlinkter Ressourcen werden APIs als HTTP-Endpunkte mit definierter URL beschrieben, deren Ein- und Ausgabeparameter den Signaturbeschreibungen RPC-basierter Modelle ähneln. Im Gegensatz zu SOAP mit WSDL oder OpenAPI präsentiert sich hRESTS aufgrund seines eingeschränkten Vokabulars jedoch offener in Hinblick auf das verwendete Nachrichtenformat und ähnelt so eher dem Ansatz von WADL (vgl. HADLEY 2009).

Integration in REST-konforme Web-APIs: Da auch hRESTS APIs aus der Sicht von HTTP-Endpunkten beschreibt, gelten hinsichtlich der Integration in existierende REST-APIs ähnliche Einschränkungen wie im Falle von OpenAPI. Eine Hypermedia-zentrierte API lässt sich mit hRESTS und den weiterführenden Standards nur unvollständig beschreiben, da auf Hypermedia-Elemente und deren Bedeutung nicht eingegangen wird und die URLs von Ressourcen als Teil der Dokumentation betrachtet werden.

Verfügbarkeit im Kontext der Nutzung: Die mit hRESTS annotierte Dokumentation kann grundsätzlich auch zur Laufzeit zur Verfügung gestellt und somit ausgewertet werden. Eine direkte Verknüpfung im Kontext der Nutzung wird jedoch nicht beschrieben.

Aufbereitung für Mensch und Maschine: Die Integration von maschinenlesbarer API-Spezifikation in menschenlesbare API-Dokumentation ist die grundlegende Idee hinter hRESTS. Der Detailgrad der maschinellen Beschreibung hängt von den ergänzend genutzten Vokabularen ab.

Beschreibung der Schnittstellenmechanik: hRESTS selbst bietet keine Elemente zur syntaktischen Beschreibung der Nachrichten (vgl. KOPECKÝ, GOMADAM, VITVAR 2008, S. 623). Die Technologien, die in diesem Zusammenhang genannt werden, sind aufgrund ihrer Nähe zu WSDL tendenziell XML-basiert. Die Nutzung anderer Schemasprachen ist grundsätzlich denkbar, wird jedoch an keiner Stelle beschrieben.

Beschreibung der fachlichen Semantik: In Hinblick auf die semantische Beschreibung der Schnittstelle profitiert hRESTS von der Offenheit des RDF-Modells.

Grundsätzlich lassen sich beliebige RDF-Vokabulare zur Beschreibung der Fachlichkeit heranziehen. SAWSDL oder WSMO-Lite wurden explizit mit dem Ziel entwickelt, Web-APIs semantisch zu beschreiben.

Auch wenn das Projekt in der Praxis kaum eine Rolle spielt, dient es als anschauliches Beispiel für die Erweiterbarkeit und Ausdrucksstärke von RDF und den Mehrwert, der durch die Orientierung an den Architekturprinzipien und Standards des Webs entsteht.

3.3 ALPS

ALPS ist ein Datenformat, das für die Definition von Profilen konzipiert wurde. Es dient somit in erster Linie dazu, die semantische Bedeutung von Daten in generischen Formaten wie JSON oder XML zu beschreiben (vgl. RICHARDSON, AMUNDSEN 2013). ALPS stellt somit einen Lösungsansatz für das in *2.3.3.3 Standardisierte und benutzerdefinierte Media Types* beschriebene Problem dar, dass Nachrichten mit generischen Media Types nicht wie von REST gefordert selbstbeschreibend sind (vgl. ebd., S. 144).

Listing 14 zeigt die ALPS-Beschreibung für eine Repräsentation der `Product`-Ressource aus dem Beispiel in *3.1 OpenAPI*. Zentrales Element von ALPS sind Deskriptoren. Sie beschreiben die Auszeichnungselemente des Datenformats, in welchem eine Repräsentation kodiert ist (vgl. AMUNDSEN, RICHARDSON, FOSTER 2015, S. 11). Die Unterscheidung zwischen Elementen zur Auszeichnung von statischen Daten und Hypermedia-Elementen erfolgt über die `type`-Eigenschaft. Auf ein HTML-Dokument angewandt finden sich die IDs der mit `semantic` typisierten Deskriptoren in den `class`-Attributen der HTML-Elemente wieder (vgl. ebd., 14ff.). In einem JSON-Dokument entsprechen sie hingegen den Namen der Eigenschaften eines Objekts. Deskriptoren die als `safe`, `unsafe` oder `idempotent` gekennzeichnet sind, beschreiben Hypermedia-Elemente (vgl. ebd., S. 17). In HTML sind dies `a`- oder `form`-Elemente mit entsprechendem `rel`-Attribut. Eine Abbildung in reinem JSON ist hingegen nicht möglich, da dieses Format keine Hypermedia-Unterstützung bietet. Hier sind andere Formate wie JSON-HAL erforderlich (vgl. KELLY 2013). Die `rt`-Eigenschaft dokumentiert für jeden Link-Deskriptor, was den Client als Antwort erwartet. Im Beispiel verweist sie hierfür auf den `ShoppingCart`-Deskriptor im selben Dokument.

```
1  {
2    "alps": {
3      "version": "1.0",
4      "doc": { "href": "https://example.hypercontract.org/profile/"
   },
5      "descriptor": [
6        {
7          "id": "Product",
8          "doc": { "value": "A Product that can be ordered." },
9          "descriptor": [
10           {
11             "id": "_id",
12             "doc": { "value": "The Product's unique identifier." },
13             "type": "semantic"
14           },
15           {
16             "id": "productName",
17             "doc": { "value": "The name of the Product." },
18             "type": "semantic"
19           },
20           {
21             "id": "productDescription",
22             "doc": { "value": "A description of the Products
   features and qualities." },
23             "type": "semantic"
24           },
25           {
26             "id": "price",
27             "doc": { "value": "The price per item." },
28             "type": "semantic"
29           },
30           {
31             "id": "image",
32             "doc": { "value": "A picture of the Product." },
33             "type": "safe"
34           },
35           {
36             "id": "addToShoppingCart",
37             "doc": { "value": "Adds the Product to the Shopping
   Cart." },
38             "type": "unsafe",
39             "rt": "#ShoppingCart"
40           }
41         ]
42       },
43       {
44         "id": "ShoppingCart",
45         "doc": { "value": "The current Shopping Cart." },
46         "descriptor": [
   […]
47         ]
48       }
49     ]
50   }
51 }
```

Listing 14: ALPS-Beschreibung für eine Produktrepräsentation

ALPS verzichtet auf Elemente, mit denen die Anfrage beschrieben wird, die für den Aufruf eines Links erforderlich ist. Das mag aus Perspektive einer herkömmlichen Schnittstellenbeschreibung ungewöhnlich erscheinen. Das Ziel von ALPS ist jedoch keine Beschreibung der API, sondern lediglich eine Dokumentation der

fachlichen Semantik einer Ressourcenrepräsentation. Idealerweise enthält die Repräsentation selbst alle notwendigen Informationen, die der Client für die Nutzung eines Hypermedia-Elements benötigt (vgl. RICHARDSON, AMUNDSEN 2013, S. 150).[14]

Vereinbarkeit mit Webarchitektur und REST: ALPS ist klar an den Designprinzipien des Webs ausgerichtet und löst ein konkretes Problem des REST-Architekturstils, ohne dabei gegen dessen Anforderungen zu verstoßen. Dabei abstrahiert ALPS über den Wertebereich der `type`-Eigenschaft selbst HTTP als Applikationsprotokoll und bewahrt so eine Unabhängigkeit vom verwendeten Protokoll.

Integration in REST-konforme Web-APIs: Die lose Kopplung an konkrete Technologien ermöglicht eine Integration mit vielfältigen Formen von REST-konformen APIs – theoretisch selbst solchen, die nicht auf dem üblichen Technologiestack des Webs basieren. Die detaillierte Beschreibung von Zustandsübergängen überlässt ALPS dabei den verwendeten Media Types und beschränkt sich auf die Charakterisierung entlang von sicheren, unsicheren und idempotenten Operationen.

Verfügbarkeit im Kontext der Nutzung: ALPS wurde als Beschreibungssprache für Profile konzipiert und kann daher über einen als `profile` typisierten Link in die Kopfzeilen oder den Nachrichteninhalt eingebunden werden (vgl. AMUNDSEN, RICHARDSON, FOSTER 2015, S. 22). Eine Verknüpfung von einzelnen Nachrichtenteilen mit dem ALPS-Profil findet über die `id`- bzw. `name`-Eigenschaften der Deskriptoren statt. Die Identifier sind somit nur innerhalb des Kontextes einer Nachricht eindeutig zuordenbar.

Aufbereitung für Mensch und Maschine: ALPS ist ein maschinenlesbares Format, die semantischen Beschreibungen der Deskriptoren sind allerdings natürlichsprachig. Eine Aufbereitung als menschenlesbare Dokumentation wäre daher mit geringem Aufwand möglich.

Beschreibung der Schnittstellenmechanik: ALPS dient allein der semantischen Beschreibung von Deskriptoren. Die Beschreibung von Datenformaten und Protokollsemantik wird den verwendeten Media Types überlassen, als dessen Ergänzung es vorgesehen ist. Dementsprechend umfasst es auch keine Möglichkeiten zur Spezifikation anderer Schnittstelleneigenschaften.

Beschreibung der fachlichen Semantik: Die Beschreibung der fachlichen Semantik eines Datenformats ist die Kernkompetenz von ALPS. Allerdings beschränkt es sich hierbei auf eine Abbildung als natürlichsprachige Beschreibungen einzelner

[14] Ein Beispiel hierfür sind HTML-Formulare, die den Request Body der durch sie ausgelöste Anfrage über Formularelemente beschreiben.

Deskriptoren. Eine maschinenlesbare Auswertung auch nur von Teilaspekten der Semantik ist daher nicht möglich.

Auf den ersten Blick erscheint der Mehrwert von ALPS überschaubar. Der hohe Abstraktionsgrad macht die Profilbeschreibungen für viele Einsatzzwecke zu unspezifisch. Beispielsweise kann nicht zwischen den HTTP-Methoden `PUT` und `DELETE` unterschieden werden, da beide als idempotent spezifiziert sind (vgl. RICHARDSON, AMUNDSEN 2013, 147f.). Der Blick auf ALPS ist dennoch hilfreich, da das Format ein Beispiel dafür ist, wie eine Profilbeschreibungssprache aussehen kann, die den Prinzipien des Webs und von REST gerecht wird.

3.4 Hydra

Hydra ist im Wesentlichen ein Vokabular für Hypermedia-basierte Web-APIs und verfolgt das Ziel, den REST-Architekturstil mit den Prinzipien von Linked Data zusammenzubringen (vgl. LANTHALER 2014). Es adressiert dabei das Problem, dass die Verknüpfung von Ressourcen über RDF-Statements zwar auf dem Linkkonzept des Webs aufsetzt, RDF jedoch keine Protokollsemantik kommuniziert. Der Konsument eines RDF-Dokuments weiß daher nicht, wie der Aufruf einer verlinkten Ressource zu erfolgen hat. Hydra löst dieses Problem über eine maschinenlesbare API-Dokumentation im JSON-LD-Format (RICHARDSON, AMUNDSEN 2013, 276).

Die Beschreibung der Schnittstelle erfolgt über eine Ressource des Typs `hydra:ApiDocumentation`. In *Listing 15* wird diese über den URI `https://example.hypercontract.org/vocabulary` identifiziert. Neben einigen Metadaten listet sie über die `hydra:supportedClass`-Eigenschaft die Klassen auf, die die Ressourcen der API beschreiben (vgl. LANTHALER 2019).

```
1  {
2    "@context": {
3      "hydra": "http://www.w3.org/ns/hydra/core#",
4      "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
5      "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
6      "shop": "https://example.hypercontract.org/vocabulary#"
7    },
8    "@id": "https://example.hypercontract.org/vocabulary",
9    "@type": "hydra:ApiDocumentation",
10   "hydra:title": "hypershop",
11   "hydra:description": "Example App for hypercontract",
12   "hydra:supportedClass": [
13     {
14       "@id": "shop:Product",
15       "@type": "hydra:Class",
16       "hydra:title": "Product",
17       "hydra:description": "A Product that can be ordered.",
18       "hydra:supportedOperation": […],
19       "hydra:supportedProperty": […]
20     },
```

```
21      {
22        "@id": "shop:AdditionToShoppingCart",
23        "@type": "hydra:Class",
24        "hydra:description": "All information necessary to add the
   Product to the Shopping Cart.",
25        "hydra:supportedOperation": [],
26        "hydra:supportedProperty": […]
27      },
28      {
29        "@id": "shop:Products",
30        "@type": "hydra:Class",
31        "hydra:description": "A list of Products.",
32        "rdfs:subClassOf": "hydra:Collection",
33        "hydra:supportedOperation": […],
34        "hydra:supportedProperty": […]
35      }
36    ]
37  }
```

Listing 15: Beispiel für eine Schnittstellenbeschreibung mit Hydra (verkürzt)

Listing 16 zeigt die Antwort auf die Abfrage einer `Product`-Ressource. Der Link in den Kopfzeilen verweist auf die API-Dokumentation aus *Listing 15*. Die `@type`-Eigenschaft identifiziert die Ressource als eine Instanz der `shop:Product-Klasse`.

```
1   HTTP/1.1 200 OK
2   Content-Type: application/ld+json
3   Link: <https://example.hypercontract.org/vocabulary>;
    rel="http://www.w3.org/ns/hydra/core#apiDocumentation"
4
5   {
6     "@context": {
7       "shop": "https://example.hypercontract.org/vocabulary#",
8       "name": "shop:name",
9       "description": "shop:description",
10      "price": "shop:price",
11      "image": "shop:image" ,
12      "addToShoppingCart": "shop:addToShoppingCart"
13    },
14    "@id": "/products/01c31228-d5ba-44b3-84b1-7cccc7b9f707",
15    "@type": "shop:Product",
16    "productName": "Handcrafted Steel Mouse",
17    "productDescription": "Deserunt accusamus vitae natus quia. […]",
18    "price": 66.07,
19    "image": "http://[…]",
20    "addToShoppingCart": "/shoppingCart/items"
21  }
```

Listing 16: Beispiel für die Repräsentation einer Produkt-Ressource in einer mit Hydra beschriebenen Web-API

Die Eigenschaften von `Product` werden in dem in *Listing 17* dargestellten Ausschnitt aus der API-Dokumentation ausführlicher beschrieben. So erfährt der Konsument, dass `image` und `addToShoppingCart` im Gegensatz zu den anderen Eigenschaften Links sind und ihre Werte daher als Linkziele zu verstehen sind. Während `image` auf eine nicht näher definierte Ressource verlinkt, verweist `addToShoppingCart` auf eine Ressource vom Typ `shop:ShoppingCartItems` und spezifiziert eine `POST`-

Operation, die auf dieser Ressource möglich ist. Diese erwartet im Request Body eine Repräsentation vom Typ `shop:AdditionToShoppingCart` und liefert als Antwort einen `shop:ShoppingCart`. Alternativ könnte diese Operation auch als `hydra:supportedOperation` der `shop:ShoppingCartItems`-Klasse definiert werden.

```
1   {
    […]
2     "hydra:supportedClass": [{
3       "@id": "shop:Product",
4       "@type": "hydra:Class",
    […]
5       "hydra:supportedProperty": [{
6           "@type": "hydra:SupportedProperty",
7           "hydra:description": "The name of the Product.",
8           "hydra:property": "shop:productName",
9           "hydra:readable": true,
10          "hydra:required": true,
11          "hydra:writeable": false,
12          "hydra:title": "Product Name"
13        }, […] {
14          "@type": "hydra:SupportedProperty",
15          "hydra:description": "The URL of a picture of the
    Product.",
16          "hydra:property": {
17            "@id": "shop:image",
18            "@type": "hydra:Link",
19            "rdfs:domain": "shop:Product",
20            "rdfs:range": "hydra:Resource",
21          },
22          "hydra:readable": true,
23          "hydra:required": false,
24          "hydra:writeable": false,
25          "hydra:title": "Product Image"
26        }, {
27          "@type": "hydra:SupportedProperty",
28          "hydra:description": "Adds a Product to the Shopping
    Cart.",
29          "hydra:property": {
30            "@id": "shop:addToShoppingCart",
31            "@type": "hydra:Link",
32            "rdfs:domain": "shop:Product",
33            "rdfs:range": "shop:ShoppingCartItems",
34            "hydra:supportedOperation": [{
35              "@type": "Operation",
36              "hydra:title": "Add to Shopping Cart",
37              "hydra:method": "POST",
38              "hydra:expects": "shop:AdditionToShoppingCart",
39              "hydra:returns": "shop:ShoppingCart"
40            }]
41          },
42          "hydra:readable": true,
43          "hydra:required": false,
44          "hydra:writeable": false
45        }
46      ]
      }, […]]
47  }
```

Listing 17: Beispiel für die Beschreibung von Eigenschaften einer Hydra-Klasse (verkürzt)

Hydra beschränkt sich jedoch nicht allein auf die API-Dokumentation, sondern bietet mit Collections auch ein Konzept, über welches Sammlungen von zusammenhängenden Ressourcen in standardisierter Form abgebildet werden können. Hierfür kommt das Hydra-Vokabular auch in der Repräsentation selbst zum Einsatz (vgl. LANTHALER 2019).

```
1  {
2    "@context": "/contexts/SearchResults.jsonld",
3    "@id": "/products",
4    "@type": "Products",
5    "hydra:member": [{
6        "@id": "/products/01c31228-d5ba-44b3-84b1-7cccc7b9f707",
7        "@type": "Product"
8      }, {
9        "@id": "/products/003eb3e2-a155-4067-a4b3-12e4916b66ac",
10       "@type": "Product"
11     },
   […]
12   ],
13   "hydra:search": {
14     "@type": "hydra:IriTemplate",
15     "hydra:template": "/products{?query}",
16     "hydra:variableRepresentation": "hydra:BasicRepresentation",
17     "hydra:mapping": [
18       {
19         "@type": "hydra:IriTemplateMapping",
20         "hydra:variable": "query",
21         "hydra:property": "hydra:freetextQuery",
22         "hydra:required": true
23       }
24     ]
25   },
26   "hydra:totalItems": 30,
27   "hydra:view": {
28     "@id": "/products?page=1",
29     "@type": "hydra:PartialCollectionView",
30     "hydra:first": "/products?page=1",
31     "hydra:last": "/products?page=3"
32   }
33 }
```

Listing 18: Beispiel für die Repräsentation einer Sammlung von Product-Ressourcen als Hydra-Collection (verkürzt)

Listing 18 zeigt die Repräsentation einer Produktliste als Hydra-Collection. `hydra:member` verweist auf die Product-Ressourcen, die der Collection angehören. Über `hydra:search` wird ein URI-Template beschrieben, mit welchem eine Suchanfrage an die Collection formuliert werden kann. `hydra:totalItems` und `hydra:view` beschreiben die vorliegende Collection als erste Seite innerhalb einer zehnseitigen Sammlung mit insgesamt 30 Produkten. Die Abbildung von Sammlungen unabhängig von ihrem fachlichen Inhalt ist hilfreich bei der Entwicklung standardisierter Clients.

Vereinbarkeit mit Webarchitektur und REST: Hydra ist im Umfeld des W3Cs entstanden und hat derzeit den Status eines W3C Drafts. Es weist sowohl konzeptionell als auch technologisch eine klare Nähe zu den nativen Technologien des Webs auf. Der Einsatz etablierter Standards spricht grundsätzlich für die Interoperabilität des Ansatzes. Die Offenheit von RDF als Beschreibungssprache fördert die Evolvierbarkeit und Erweiterbarkeit der Lösung und erlaubt die Verwendung existierender Vokabulare (vgl. LANTHALER 2014, S. 146).

Im Gegensatz zu vielen anderen Beschreibungssprachen für Web-APIs dokumentiert Hydra die Schnittstelle nicht ausgehend von ihren URLs, sondern anhand von Ressourcentypen. Dies erlaubt Hypermedia-getriebene Applikationen im Sinne von REST (vgl. LANTHALER 2014, 131). Allerdings geht damit auch einher, dass das Repräsentationsformat diese Zuordnung zu einem Ressourcentyp erlaubt. Hydra kann daher nur in Kombination mit einem RDF-basierten Nachrichtenformat eingesetzt werden und setzt selbst hauptsächlich auf JSON-LD (vgl. ebd., S. 141). In Hinblick auf die Schnittstellensemantik ergibt sich damit ein interessantes Problem: Der Media Type `application/ld+json` vermittelt für sich genommen keine Semantik. Diese wird über die verwendeten Vokabulare in den RDF-Statements des Nachrichteninhalts kodiert. HTTP bietet jedoch keine Möglichkeit, diese im Rahmen der Content Negotiation zwischen Client und Server auszuhandeln. Es ist somit nicht sichergestellt, dass ein JSON-LD-kompatibler Client die Hydra-Elemente in den Nachrichteninhalten versteht.

Eine weitere Problematik ergibt sich aus der Tatsache, dass Hydra viele Sprachelemente von RDF und RDF Schema in eigene Konstrukte verpackt. Statt Ressourcentypen als `rdf:Class` zu beschreiben, werden sie als `hydra:Class` definiert, um sie als dereferenzierbar zu kennzeichnen. Die `hydra:expects`-Eigenschaft einer Operation nutzt als Wertebereich ebenfalls `hydra:Class` (vgl. LANTHALER 2014, 131f.). Effektiv bedeutet dies, dass der Anfrageinhalt einer Operation stets eine dereferenzierbare Ressource sein muss. Eine Unterscheidung zwischen Ressourcen und ihren Repräsentationen findet nicht statt. Die Klasse `shop:AdditionToShoppingCart` aus *Listing 15* müsste somit eine abfragbare Ressource darstellen, auch wenn sie lediglich eine vereinfachte Repräsentation eines Warenkorbelements beschreibt.

Eine getrennte Betrachtung von Repräsentationen würde auch die Bedeutung der Hydra-Elemente `readable`, `required` und `writeable` zur Definition von Eigenschaften einer Ressource vereinfachen. Das Modell von Hydra beschreibt jedoch eine Sichtweise auf Ressourcen, bei denen die Repräsentationen in den Anfragen

und Antworten grundsätzlich die gleiche Struktur haben und sich lediglich in den drei genannten Eigenschaften unterscheiden.

Integration in REST-konforme Web-APIs: Der Umstand, dass Hydra nur mit RDF-basierten Nachrichtenformaten sinnvoll genutzt werden kann, erschwert die Integration in bestehende Anwendungen. Denkbar wäre allerdings, dass bestehende JSON-Formate über im Antwortheader verlinkte JSON-LD-Kontexte nachträglich zu RDF-Daten aufgewertet werden (vgl. SPORNY, KELLOGG, LANTHALER 2014). Weitere Probleme können sich aus dem starren Modell von Hydra in Hinblick auf Ressourcen und ihre Repräsentationen ergeben. Andererseits setzen viele REST-APIs auf identische Repräsentationen für Anfragen und Antworten und könnten somit über Hydra abgebildet werden. Insofern ist stets eine Einzelfallbetrachtung notwendig.

Verfügbarkeit im Kontext der Nutzung: Hier profitiert Hydra von den Linked-Data-Prinzipen, auf die es aufsetzt. Da alle Auszeichnungselemente und Konzepte über URIs eindeutig identifiziert werden können, ist das Dereferenzieren zur Laufzeit möglich. Dafür ist lediglich erforderlich, dass URIs auf die entsprechenden Elemente innerhalb der API-Dokumentation verweisen.

Aufbereitung für Mensch und Maschine: Der RDF-zentrierte Ansatz von Hydra ist ideal für eine maschinelle Nutzung der API-Beschreibung. Die Erweiterbarkeit von RDF ermöglicht zudem den Einsatz beliebiger Vokabulare zur Beschreibung einzelner API-Elemente in menschenlesbarer Form. Dennoch hat sich bislang keine Lösung für die Erzeugung menschenlesbarer Dokumentation etabliert.

Beschreibung der Schnittstellenmechanik: Für die Spezifikation der Protokollsemantik einer Schnittstelle sind im Hydra-Vokabular alle notwendigen Elemente enthalten. Die schematische Beschreibung der Nachrichteninhalte ist hingegen komplexer. Da Hydra auf allen Ebenen mit RDF arbeitet, werden die Struktur der Ressourcen und die verwendeten Datentypen grundsätzlich mit den Mitteln von RDF Schema und OWL beschrieben (vgl. LANTHALER 2014, S. 145). Die schematische Beschreibung des Nachrichtenformats ist demnach über die Spezifikation der jeweiligen RDF-Serialisierung geregelt. Allerdings ergänzt Hydra dies um das Konzept der `SupportedProperty`, über welches festlegt wird, ob eine Eigenschaft lesbar, schreibbar oder überhaupt notwendig ist. Hier existieren Überschneidungen mit den Sprachelementen von OWL, was die syntaktische Beschreibung der Nachrichteninhalt vergleichsweise unübersichtlich macht. Zudem erschwert dies die Nutzung von Hydra durch JSON-kompatible Clients ohne RDF-Unterstützung.

Beschreibung der fachlichen Semantik: Mit `title` und `description` bietet das Hydra-Vokabular zwei Eigenschaften, mit denen sich Klassen, Eigenschaften, Links, Operationen, HTTP-Statuscodes sowie die API-Dokumentation selbst natürlichsprachig beschreiben lassen. Letztendlich sind sie aber nur Ableitungen von `rdfs:label` und `rdfs:comment`, die grundsätzlich auf jede Ressource in RDF anwendbar und somit für eine noch kleinteiligere Beschreibung geeignet sind.

3.5 Zusammenfassung

Eine umfassende Betrachtung aller relevanten Lösungen zur Beschreibung von REST-APIs ist im gegebenen Rahmen nicht möglich. Neben den hier vorgestellten Lösungen existiert eine Vielzahl weiterer Projekte, die sich in verschiedensten Aspekten unterscheiden, in Hinblick auf die Zielsetzung der Arbeit jedoch wiederkehrende Vor- und Nachteile aufweisen. So existieren mit RAML und API Blueprint valide Alternativen zu OpenAPI, sie alle eint jedoch der Ansatz einer URL-zentrierten Betrachtung der Schnittstelle (vgl. NEMEC 2019; RAML 2019). Gleiches trifft auf die W3C-nahe Web Application Description Language (WADL) zu (vgl. HADLEY 2009). Mit JSON Hyper-Schema existiert andererseits eine Erweiterung für JSON Schema, die durchaus für die Beschreibung REST-konformer, Hypermedia-getriebener Web-APIs geeignet ist, aber wie viele andere Lösungen an JSON als Nachrichtenformat gebunden und in Hinblick auf die semantische Aussagekraft mit OpenAPI und anderen Lösungen vergleichbar ist (vgl. HENRY, AUSTIN 2018).

Umso wichtiger ist daher, dass die in diesem Kapitel gewonnenen Erkenntnisse in Konzept einfließen, das im weiteren Verlauf der Arbeit entwickelt wird.

Vereinbarkeit mit Webarchitektur und REST: Anhand von hRESTS und Hydra wird deutlich, dass RDF einen wichtigen Faktor für die Vereinbarkeit der Technologien mit der Architektur des Webs darstellt. Die Offenheit von RDF fördert die Evolvierbarkeit und Erweiterbarkeit der Lösung, während der konsequente Einsatz von URIs als formatübergreifendes Identifikationssystem sich positiv auf die Interoperabilität auswirkt.

ALPS und Hydra liefern darüber hinaus Beispiele für die Beschreibung von Web-APIs ohne URL-zentrierten Ansatz, was für die Vereinbarkeit mit dem Hypermedia-Prinzip von REST spricht. Der deskriptororientierte Ansatz von ALPS ist hierbei im Gegensatz zum RDF-abhängigen Vorgehen von Hydra unabhängig vom Nachrichtenformat und führt somit nicht zu Einschränkungen bei der Content Negotiation von HTTP, dem zentralen Mechanismus für die Aushandlung des Contracts zwischen Konsumenten und Anbieter der API.

Integration in REST-konforme Web-APIs: Die Betrachtung von OpenAPI und Hydra zeigt, dass die Einschränkung des Nachrichtenformats die Integration in existierende, REST-konforme Anwendungen stark behindert. Wesentlich weniger invasiv ist die von ALPS eingesetzte Erweiterung eines Media Types über ein Profil. Für eine weitere Integration ist zudem die Erweiterung von Nachrichteninhalten über im Header verlinkte JSON-LD-Kontexte möglich. Dadurch lassen sich Deskriptoren in einem JSON-basierten Nachrichtenformat nachträglich zu URIs aufwerten und so eindeutig referenzieren.

Verfügbarkeit im Kontext der Nutzung: Die Nutzung von URIs als Deskriptoren in Kombination mit den Prinzipien von Linked Data ermöglicht das Dereferenzieren von Nachrichtenelementen zur Laufzeit. So dienen die URIs nicht nur als eindeutiger Identifier, sondern auch als Locator für die zugehörige Dokumentation. Wie mächtig dieses Konzept ist, zeigt die Betrachtung von Hydra.

Aufbereitung für Mensch und Maschine: Sowohl hRESTS als auch OpenAPI zeigen, dass eine menschenlesbare Dokumentation und eine maschinenlesbare Beschreibung der Schnittstelle nicht zwingend einen doppelten Pflegeaufwand bedeuten. Die Popularität von OpenAPI spricht für den Ansatz, menschenlesbare Dokumentation automatisiert aus einer maschinenlesbaren Spezifikation zu erzeugen.

Beschreibung der Schnittstellenmechanik: Für die Beschreibung von Protokollsemantik liefert Hydra mit dem Ansatz, diese ausgehend von Links und Operationen zu beschreiben, einen praktikablen Ansatz, der den Anforderungen von REST gerecht wird. Die schematische Beschreibung der Nachrichteninhalte ist hingegen eine der wesentlichen Stärken von OpenAPI. Weder hRESTS, ALPS noch Hydra kommen an den Detailgrad der auf JSON Schema basierenden Spezifikation mit OpenAPI heran. Andererseits ist eine Einschränkung des Nachrichtenformats wie sie OpenAPI vorgibt nicht wünschenswert. Die Beschreibung von Datenschemas in RDF, wie sie von Hydra praktiziert wird, ist ebenfalls nur bei RDF-basierten Nachrichtenformaten sinnvoll.

Beschreibung der fachlichen Semantik: In Hinblick auf die Abbildung der fachlichen Semantik zeugen hRESTS und Hydra erneut von den Vorteilen, die der Einsatz von RDF bringt. Nur hRESTS ist in der Lage, fachliche Aspekte noch detaillierter darzustellen, allerdings primär in menschenlesbarer Form.

Die im weiteren Verlauf der Arbeit beschriebene Lösung baut auf den hier beschriebenen Erkenntnissen auf und versucht so, die Stärken der verschiedenen Ansätze miteinander zu kombinieren.

4 hypercontract

Mit hypercontract wird im Folgenden eine Lösung beschrieben, die im Einklang mit der Architektur und den Prinzipien des Webs eine menschen- und maschinenlesbare Beschreibung von REST-konformen APIs ermöglicht. Um die Funktionsweise von hypercontract anschaulich zu erläutern, wurde eine einfache Webapplikation namens hypershop entwickelt, deren REST-API im Rahmen dieses Kapitels mit Hilfe von hypercontract dokumentiert werden soll.[15]

Als Leitfaden dient die „Seven-Step Design Procedure“ für REST-basierte Web-APIs von Richardson, Amundsen 2013:

1. Semantische Deskriptoren erfassen
2. Zustandsdiagramm zeichnen
3. Bezeichnungen konsolidieren
4. Media Type auswählen
5. Profil schreiben
6. Implementierung
7. Veröffentlichung

(vgl. Richardson, Amundsen 2013, S. 158)

In Abschnitt *4.1* wird anhand der Schritte 1 bis 4 der fachliche Kontext vorgestellt, für den die REST-Schnittstelle von hypershop entwickelt wird. Anschließend wird in Abschnitt *4.2* mit Hilfe von hypercontract das Profil erstellt (Schritt 5), welches den Contract für die Nutzung der Schnittstelle beschreibt. Da hypershop bereits über eine REST-API verfügt, erläutert der Abschnitt *4.3* in Hinblick auf die Schritte 6 und 7 die Integration und Bereitstellung der erarbeiteten Schnittstellendokumentation.

4.1 Anwendungsbeispiel: hypershop

hypershop ist eine Applikation, die mit dem Ziel entwickelt wurde, die Konzepte von hypercontract zu illustrieren. Dieser Abschnitt befasst sich hauptsächlich mit der Darstellung des fachlichen Kontextes, den die Applikation über Ihre Schnittstelle veröffentlicht. Auf die konkrete Implementierung der Anwendung wird nur insofern eingegangen, wie sie für die Betrachtung von hypercontract relevant ist.

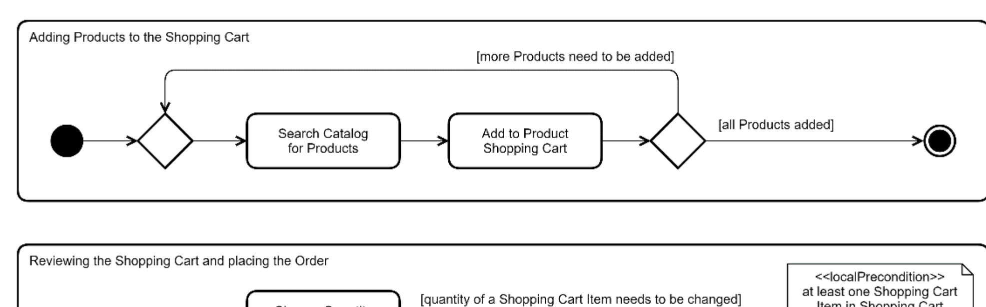
[15] Die Beispiele in diesem Kapitel können über https://example.hypercontract.org nachvollzogen werden. Der Quellcode von hypercontract und hypershop ist Open Source und wird auf GitHub unter https://github.com/hypercontract/hypercontract bereitgestellt.

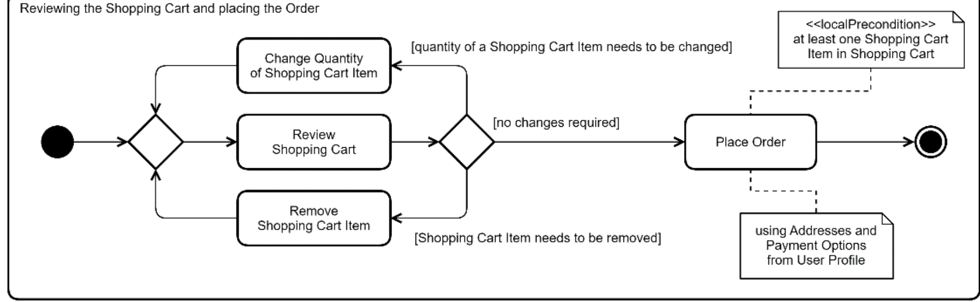
4.1.1 Fachlicher Kontext

Aus fachlicher Sicht soll hypershop eine Reihe funktionaler Anforderungen erfüllen. Nutzer der Applikation können

- einen Katalog mittels einer Suchanfrage nach Produkten durchsuchen,
- Produkte unter Angabe einer Bestellmenge dem Warenkorb hinzufügen,
- ggf. die Bestellmenge von Warenkorbpositionen ändern,
- ggf. Warenkorbpositionen wieder entfernen,
- den Warenkorb zusammen mit Liefer- und Rechnungsadresse sowie Zahlungsinformationen aus dem Benutzerprofil als Bestellung aufgeben (sofern mindestens eine Warenkorbpositionen vorhanden ist),
- die Bestellhistorie einsehen und
- ggf. eine Bestellung unter Angabe einer Begründung stornieren (sofern sie noch nicht bezahlt ist).

Abbildung 10 visualisiert diese Anforderungen in Form eines Aktivitätsdiagramms.





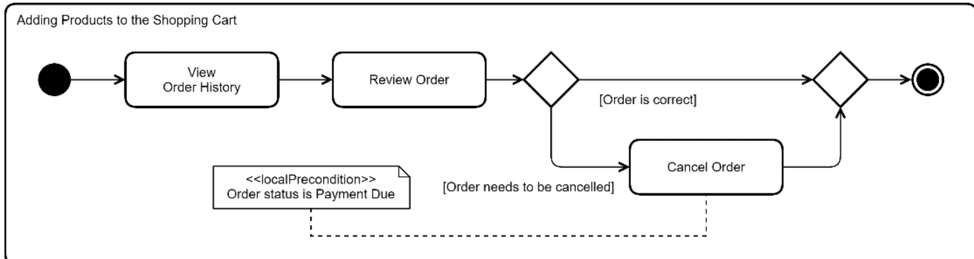


Abbildung 10: Der in hypershop abgebildeten Aktivitäten

Diese Darstellung ist hilfreich, um die wesentlichen *Aktivitäten* zu identifizieren, die Nutzer mit Hilfe der Anwendung durchführen und welchen *Vorbedingungen* diese Aktivitäten unterliegen.

Konzentriert man sich bei der Betrachtung der Anforderungen auf die Substantive, erhält man wichtige Hinweise darauf, welche *Konzepte* in der Interaktion mit der Applikation eine Rolle spielen (vgl. EVANS 2011, S. 24ff.). Daraus entsteht das in *Abbildung 11* dargestellte semantische Datenmodell, welches die Eigenschaften und Zusammenhänge der Konzepte grafisch dokumentiert. Technische Implementierungsaspekte spielen bei dieser Betrachtung noch keine Rolle.

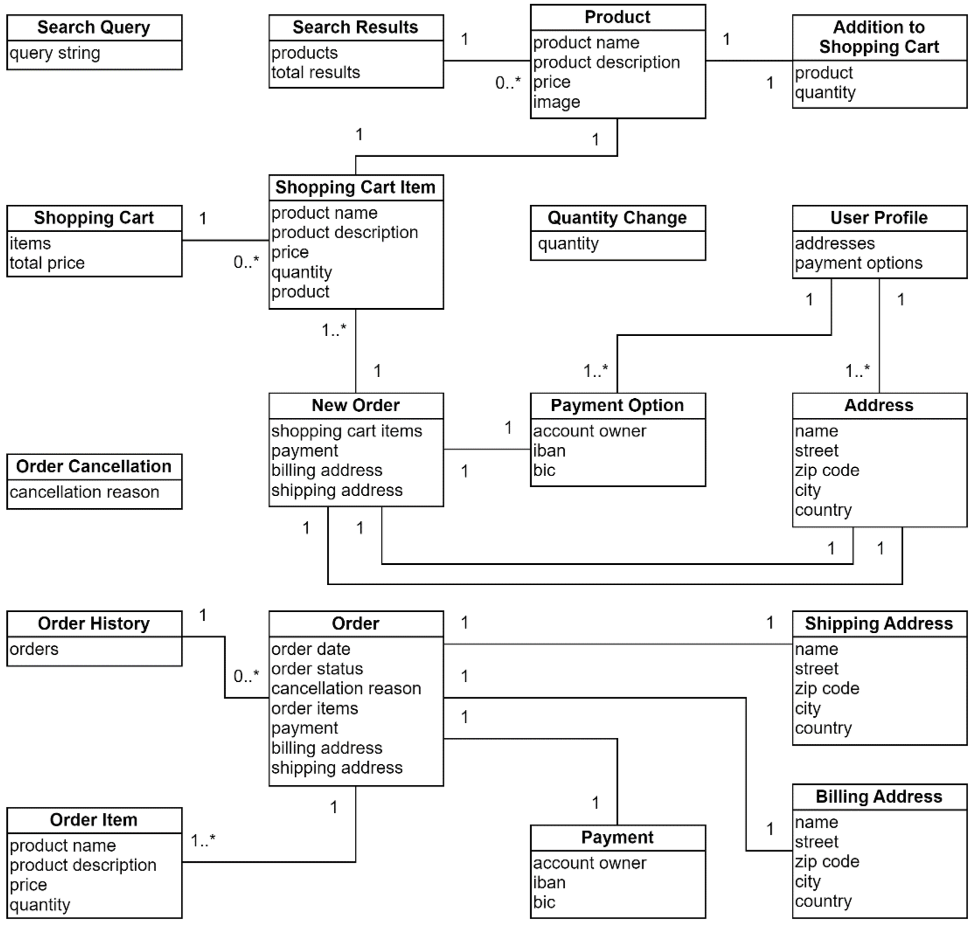


Abbildung 11: Das semantische Datenmodell von hypershop

Dennoch verrät die Abbildung viele fachliche Details über die Anwendung. Angefangen mit der Suchanfrage an den Katalog (*Search Query*), die lediglich eine textbasierte Suche erlaubt. Die Suchergebnisse (*Search Results*) verweisen auf eine Reihe von Produkten (*Product*). Um diese dem Warenkorb hinzuzufügen, ist eine Referenz auf das Produkt und die Bestellmenge erforderlich (*Addition To Shopping Cart*). Auffallend ist hierbei, dass sich Produkt und Warenkorbposition (*Shopping Cart Item*) viele Eigenschaften teilen. Gleiches gilt für die Bestellpositionen (*Order Item*), die erzeugt werden, wenn Warenkorbpositionen zusammen mit Adressen (*Address*) und gewählter Zahlungsoption (*Payment Option*) aus dem Benutzerprofil (*User Profile*) als neue Bestellung (*New Order*) aufgegeben werden. Die Bestellung

(*Order*) ist anschließend über die Bestellhistorie (*Order History*) verfügbar. Auch die Änderung der Bestellmenge (*Quantity Change*) und die Stornierung einer Bestellung mit Angabe des Stornierungsgrunds (*Order Cancellation*) sind als eigene Konzept erfasst.

Dieser fachliche Kontext ist die Grundlage für die Dokumentation der semantischen Deskriptoren und der möglichen Zustandsübergänge.

4.1.2 Semantische Deskriptoren

Der als Repräsentation übermittelte Ressourcenzustand wird über semantische Deskriptoren ausgezeichnet. In einem XML-basierten Media Type geschieht dies üblicherweise in Form von XML-Elementen, in JSON-Nachrichten werden sie als Eigenschaftsnamen abgebildet.

Unabhängig vom verwendeten Nachrichtenformat kann das Profil die semantische Bedeutung des Deskriptors festhalten, sowie dessen Kardinalität und grundlegender Datentyp.[16] Die syntaktische Beschreibung des Deskriptors ist hingegen Aufgabe der Spezifikation des Media Types.

Der Ausgangspunkt für die Definition der Deskriptoren ist der fachliche Kontext und insbesondere das semantische Datenmodell. Zunächst werden die natürlichsprachigen Benennungen der Konzepte in technische Bezeichner übersetzt und anschließend fachlich beschrieben (siehe *Tabelle 1*).

Konzept	Deskriptor	Beschreibung	Typ
Addition to Shopping Cart	`AdditionToShoppingCart`	All information necessary to add a Product to the Shopping Cart.	Klasse
Product	`Product`	A Product that can be ordered.	Klasse
product	`product`	A reference to a Product from the catalog.	Eigenschaft
quantity	`quantity`	The number of items ordered or to be ordered.	Eigenschaft

Tabelle 1: Beispiel für die Beschreibung semantischer Deskriptoren

Unterschieden wird dabei zwischen den Entitäten, die als Klassen beschrieben werden, und ihren Eigenschaften. Für letztere wird zusätzlich noch der Definitionsbereich und der Wertebereich inklusive Kardinalität festgehalten (siehe *Tabelle 2*).

[16] Die möglichen Datentypen unterscheiden sich abhängig vom Datenformat. Während JSON beispielsweise über den Datentyp `number` alle numerischen Werte abbildet (vgl. Ecma 2017), unterscheidet XML unter anderem zwischen Ganzzahlen, Dezimalzahlen und Fließkommazahlen mit unterschiedlichen Wertebereichen (vgl. Biron, Malhotra 2004).

Deskriptor	Definitionsbereich	Wertebereich	Kardinalität
`orderItems`	`Order`	`OrderItem`	1-n
`product`	`AdditionToShoppingCart` `ShoppingCartItem`	`Product`	1

`quantity`	`AdditionToShoppingCart`
	`ShoppingCartItem`
	`OrderItem`
	`QuantityChange`
		Integer
		(min: 1)
			1

Tabelle 2: Beispiel für die Beschreibung semantischer Deskriptoren mit Definitions- und Wertebereichen für Eigenschaften

Eine vollständige Beschreibung der semantischen Deskriptoren befindet sich in Anhang *C Semantische Deskriptoren von hypershop*.

4.1.3 Zustandsübergänge

Die möglichen Zustandsübergänge in der REST-API ergeben sich einerseits aus den identifizierten Aktivitäten im Aktivitätsdiagramm. Andererseits sind aber auch Beziehungen zu anderen Konzepten im semantischen Datenmodell potenzielle Zustandsübergänge. Das Zustandsdiagramm in *Abbildung 12* enthält daher auch Zustandsübergänge, die in *4.1.2 Semantische Deskriptoren* bereits als Eigenschaften beschrieben wurden.

Ähnlich zur Startseite einer Website ist `ApiRoot` der Startpunkt für die Interaktion mit der Schnittstelle. Von hier kann der Nutzer den Katalog durchsuchen (`searchCatalog`), den Warenkorb (`shoppingCart`) und die existierenden Bestellungen einsehen (`orderHistory`) und das Benutzerprofil abfragen (`userProfile`).

Folgen wir dem Ablauf im Aktivitätsdiagramm, gelangt der Nutzer über die Suchergebnisse zu einem Produkt (`products`) und kann dieses dem Warenkorb hinzufügen (`addToShoppingCart`). Im Warenkorb hat er die Möglichkeit, Warenkorbpositionen zu entfernen (`remove`) und deren Bestellmenge zu ändern (`changeQuantity`). Ausgehend vom Warenkorb ist es möglich, eine Bestellung aufzugeben (`placeOrder`), welche auch wieder storniert werden kann (`cancel`).

Zwei der Zustandsübergänge sind mit Vorbedingungen verknüpft. Eine Bestellung kann nur aufgegeben werden, wenn mindestens eine Position im Warenkorb vorhanden ist. Auf etwas abstrakterer Ebene steht darüber die Frage, ob ein Warenkorb bestellbar ist (`isShoppingCartOrderable`). Durch diese Abstraktion ist es möglich, die konkrete Bedingung zu einem späteren Zeitpunkt anzupassen ohne dass dies konkrete Änderungen auf Nutzerseite zur Folge hat. Beispielsweise könnte die Vorbedingung um einen Mindestbestellwert ergänzt werden. Analog ist

die Stornierung einer Bestellung nur erlaubt, wenn sie noch nicht bezahlt ist (`isOrderCancellable`).

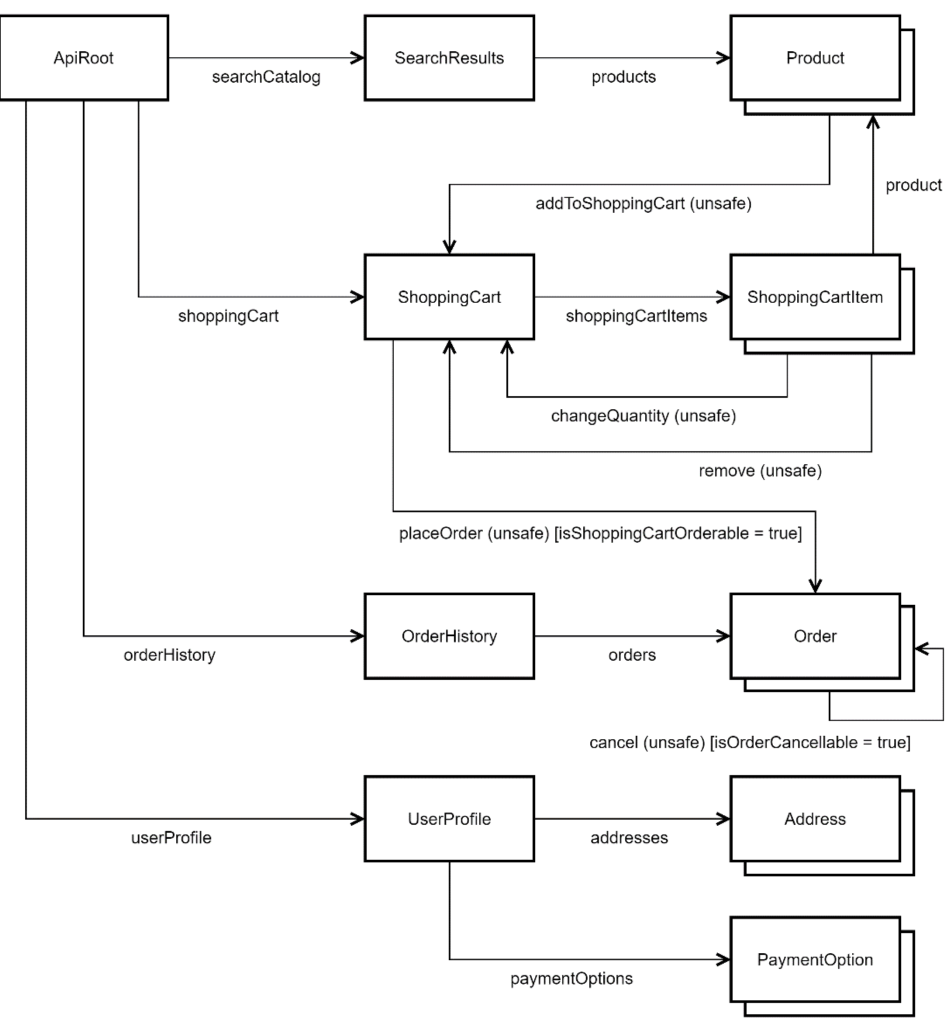


Abbildung 12: Zustandsdiagramm der hypershop REST-API

Bei der Betrachtung der Zustandsübergänge fällt auf, dass sichere Zustandsübergänge, also solche, die den Applikationszustand nicht verändern (`product`, `shoppingCart`, `addresses`, usw.), tendenziell Bezeichnern aus dem semantischen Datenmodell entsprechend. Die unsicheren Zustandsübergängen, die potenziell auch eine Veränderung des Ressourcenzustands zur Folge haben können (`addToShoppingCart`, `changeQuantity`), basieren hingegen auf den Aktivitäten des Aktivitätsdiagramms.[17]

Anhang *D Zustandsübergänge von hypershop* enthält eine vollständige Auflistung aller Zustandsübergänge mit ihren Eigenschaften und Vorbedingungen.

[17] `searchCatalog` ist hier die Ausnahme, die die Regel bestätigt

4.1.4 Einheitliches Vokabular

Die explizite Benennung von semantischen Deskriptoren und Zustandsübergängen schafft ein einheitliches Vokabular, das für das weitere Vorgehen unerlässlich ist. Eine konsistente Sprache ist die Voraussetzung um Konzepte eindeutig identifizieren und im weiteren Schritt auch beschreiben zu können.

Dabei dient das Vokabular nicht nur der Beschreibung der Schnittstelle, sondern sollte diese auch in ihrer Implementierung prägen. Im Sinne eines von der fachlichen Domäne getriebenen Schnittstellendesigns (vgl. EVANS 2011, S. 24ff.) ist es wichtig, dass sich die Fachsprache des Anwendungskontextes auch in der Schnittstelle wiederfindet.

Im Ergebnis entsteht so eine für alle Beteiligten einheitliche Sprache, die das gemeinsame Verständnis für den fachlichen Kontext fördert.

4.1.5 Repräsentationsformate

Eine wesentliche Eigenschaft von REST ist die Trennung Ressourcen und ihren Repräsentationen und die Aushandlung des Media Types zur Laufzeit. Dadurch ist es möglich, dass eine Schnittstelle dieselbe Funktionalität in unterschiedlichen Nachrichtenformaten anbietet.

Die REST-Schnittstelle von hypershop nutzt diese Eigenschaft, um anhand von HTML, JSON, JSON-HAL und JSON-LD die Konzepte von hypercontract zu vermitteln.[18] Die in *4.1.1 Fachlicher Kontext* identifizierten Aktivitäten sind daher in jedem dieser Nachrichtenformate durchführbar.

4.1.5.1 HTML

Die Unterstützung von HTML ermöglicht die Nutzung der Schnittstelle durch einen Anwender direkt im Webbrowser. Dieser sendet Anfragen an die vom Benutzer eingegebenen URLs grundsätzlich mit dem Accept-Header `text/html` (neben einigen Fallback-Formaten). *Listing 19* zeigt die vom Webbrowser gesendet Anfrage der Produktdetailseite von hypershop.

```
1   GET /products/01c31228-d5ba-44b3-84b1-7cccc7b9f707 HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
    */*;q=0.8
```

Listing 19: Clientanfrage (verkürzt) für die Abfrage eines Produkts als HTML-Repräsentation

[18] Eine Betrachtung XML-basierter Formate ist in Hinblick auf den Umfang der Arbeit nicht möglich, für die Validierung von hypercontract als formatunabhängige Lösung zur Schnittstellenbeschreibung jedoch ein sinnvoller nächster Schritt.

Die Antwort des Servers bestätigt den angefragten Dokumenttyp (siehe *Listing 20*) und einigt sich so mit der Applikation auf die HTML-basierte Repräsentation, die mit der Antwort ausgeliefert wird.

```
1   HTTP/1.1 200 OK
2   Content-Type: text/html; charset=utf-8
3
4   <!DOCTYPE html>
5   <html>
6     <head>[…]</head>
7     <body>
    […]
8       <main>
9         <h1>Handcrafted Steel Mouse</h1>
10        <div>
11          <figure><img src="[…]" /></figure>
12          <div>
13            <p>Price: <strong>66.07</strong></p>
14            <form method="POST"
    action="https://example.hypercontract.org/shoppingCart/items">
15              <input type="hidden" name="product" value="01c31228-
    d5ba-44b3-84b1-7cccc7b9f707" />
16              <div>
17                <input type="text" name="quantity" value="1">
18                <div><button type="submit">Add</button></div>
19              </div>
20            </form>
21            <p>Deserunt accusamus vitae natus quia. Ullam […]</p>
22          </div>
23        </div>
24      </main>
25    </body>
26  </html>
```

Listing 20: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und HTML-Repräsentation

Zu erkennen ist, dass die Repräsentation mit dem `img`- und dem `form`-Element zwar Hypermedia-Elemente enthält, die auf verwandte Ressourcen verweisen, diese aber keine Link-Relationstypen ausweisen. Die Informationen über den Ressourcenzustand – Produktname, Preis, Produktbeschreibung – sind mit Hilfe von semantischen Deskriptoren des HTML-Standards ausgezeichnet (`h1`, `strong`, `p`) und verlieren somit einen wesentlichen Teil ihrer fachlichen Bedeutung.

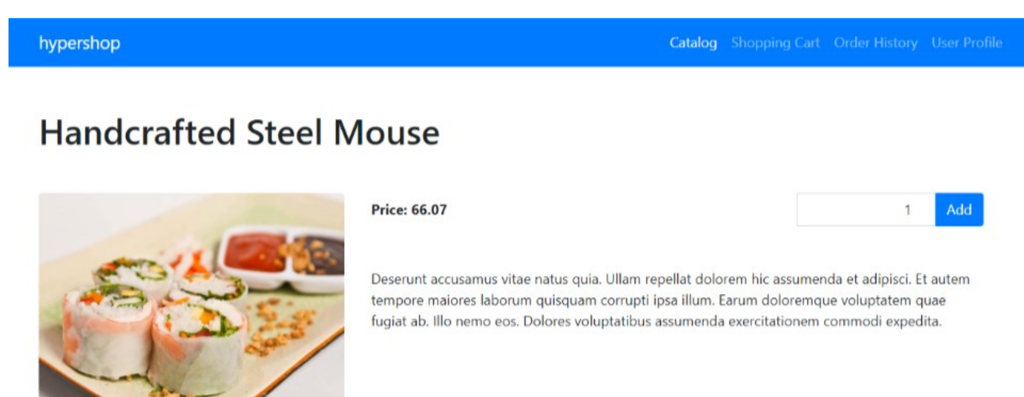


Abbildung 13: Darstellung der HTML-Repräsentation eines Produkts im Webbrowser

Allerdings reicht die semantische Aussagekraft von HTML aus, um eine für den Menschen verständliche Darstellung der Produktressource anzuzeigen (siehe *Abbildung 13*). Grundsätzlich ließe sich dieses Repräsentationsformat auch in einer Programmschnittstelle nutzen. Geeigneter sind hierfür jedoch Formate, die auf die dokumenten-zentrierten Features von HTML verzichten.

4.1.5.2 JSON

Ein wesentlich populäreres Format für den Datenaustausch zwischen Client und Server ist JSON. Die Abfrage derselben Produkt-Ressource als JSON-Repräsentation erfolgt über das Setzen eines `Accept`-Headers mit dem Wert `application/json` (siehe *Listing 21*).

```
1   GET /products/01c31228-d5ba-44b3-84b1-7cccc7b9f707 HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: application/json
```

Listing 21: Clientanfrage für die Abfrage eines Produkts als JSON-Repräsentation

Die Antwort des Servers enthält nun eine wesentlich kompaktere Darstellung des Produkts, vermittelt jedoch dieselben Informationen über den Ressourcenzustand (siehe *Listing 22*).

```
1   HTTP/1.1 200 OK
2   Content-Type: application/json; charset=utf-8
3
    {
4     "_id": "https://example.hypercontract.org/products/01c31228-d5ba-
    44b3-84b1-7cccc7b9f707",
5     "productName": "Handcrafted Steel Mouse",
6     "productDescription": "Deserunt accusamus vitae natus quia. […]",
7     "price": 66.07,
8     "image": "https://[…]"
9   }
```

Listing 22: Antwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-Repräsentation

Was im Vergleich zur HTML-Repräsentation fehlt ist ein Hypermedia-Element, mit dem das Produkt zum Warenkorb hinzugefügt werden kann. Da JSON kein Hypermedia-Format ist, bietet es keine Unterstützung für das Verlinken von Ressourcen. Zwar sind die Werte Eigenschaften `_id` und `image` URIs, das JSON-Format ermöglicht aber keine semantische Auszeichnung, die darüber informiert, dass es sich hierbei um Linkkontext und Linkziel im Sinne der Web-Linking-Spezifikation handelt (vgl. NOTTINGHAM 2017). Für JSON-kompatible Clients unterscheidet sich dieser Wert ebenso wie der URI der `_id`-Eigenschaft nicht von irgendeiner anderen Zeichenkette.

Im Gegensatz zur HTML-Repräsentation finden sich in dieser Darstellung allerdings die semantischen Deskriptoren wieder, die in *4.1.2 Semantische Deskriptoren*

definiert wurden. Diese sind zwar nicht global eindeutig, wie sie es in Form eines URI wären, aber immerhin entsteht so für den menschlichen Betrachter im Kontext der Anwendung eine Beziehung zum fachlichen Modell. Nichtsdestotrotz sind sie für den Client ohne Spezifikation über den Media Type ein Profil lediglich Auszeichungselemente ohne semantische Bedeutung.

4.1.5.3 JSON-HAL

JSON-HAL ist eines von verschiedenen JSON-basierten Formaten, die das reine Datenformat JSON um die Möglichkeit erweitern, Hypermedia-Elemente einzubetten[19]. Hierfür werden Verweise auf andere Ressourcen in einem `_links`-Objekt abgebildet. Die Abfrage einer JSON-HAL-Repräsentation erfolgt wie in *Listing 23* dargestellt über den Media Type `application/hal+json` (vgl. KELLY 2013).

```
1   GET /products/01c31228-d5ba-44b3-84b1-7cccc7b9f707 HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: application/hal+json
```

Listing 23: Clientanfrage für die Abfrage eines Produkts als JSON-HAL-Repräsentation

Die ausgelieferte Repräsentation des Produkts enthält ähnlich der HTML-Darstellung Hypermedia-Elemente für die Produktabbildung und das Hinzufügen zum Warenkorb (siehe *Listing 24*).

```
1   HTTP/1.1 200 OK
2   Content-Type: application/hal+json; charset=utf-8
3
4   {
5     "productName": "Handcrafted Steel Mouse",
6     "productDescription": "Deserunt accusamus vitae natus quia. […]",
7     "price": 66.07,
8     "_links": {
9       "self": {
10        "href": "https://example.hypercontract.org/products/01c31228-
    d5ba-44b3-84b1-7cccc7b9f707"
11      },
12      "https://example.hypercontract.org/profile/addToShoppingCart": {
13        "href": "https://example.hypercontract.org/shoppingCart/
    items"
14      },
15      "https://example.hypercontract.org/profile/image": {
16        "href": "https://[…]"
17      }
18    }
19  }
```

Listing 24: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-HAL-Repräsentation

Die Link-Relationstypen der Links werden als Objektschlüssel im `_links` -Objekt hinterlegt (im Beispiel hervorgehoben) und sind daher in JSON-HAL obligatorisch.

[19] Zu den weiteren JSON-basierten Formaten mit Hypermedia-Unterstützung gehören unter anderem Siren, JSON:API und Collection+JSON.

Die Namensgebung basiert auf den identifizierten Zustandsübergängen aus *4.1.3 Zustandsübergänge*. Zusätzlich ersetzt der `self`-Link[20] die `_id`-Eigenschaft zur Auszeichnung der Produkt-ID als Identifier und als Kontext der anderen Links.

Der Ressourcenzustand wird auf die gleiche Weise transportiert, wie dies im Falle von JSON geschieht. Der semantische Mehrwert von JSON-HAL gegenüber regulären JSON-Repräsentationen liegt in der Möglichkeit, Links als solche für den Client verständlich auszuzeichnen. Die fachliche Bedeutung des Links wird ebenso wenig vermittelt, wie dies bei den Auszeichnungselementen des Ressourcenzustands der Fall ist.

4.1.5.4 JSON-LD

JSON-LD baut syntaktisch wie JSON-HAL auf JSON auf, ist darüber hinaus aber auch eine konkrete RDF-Syntax (vgl. CYGANIAK, WOOD, LANTHALER 2014), basiert also auf dem Datenmodell von RDF (vgl. SPORNY, KELLOGG, LANTHALER 2014). Der Ressourcenzustand und die Beziehungen der Ressource zu anderen Ressourcen werden bei der Abfrage eines Produkts als JSON-LD-Repräsentation (siehe Listing 25) in Form eines RDF-Graphen serialisiert.

```
1  GET /products/01c31228-d5ba-44b3-84b1-7cccc7b9f707 HTTP/1.1
2  Host: example.hypercontract.org
3  Accept: application/ld+json
```

Listing 25: Clientanfrage für die Abfrage eines Produkts als JSON-LD-Repräsentation

Der Ressourcenzustand wird auf die gleiche Weise abgebildet, wie dies schon bei JSON und JSON-HAL der Fall ist: über die Objektschlüssel `productName`, `productDescription` und `price` (siehe *Listing 26*). JSON-LD-Kontext übersetzt diese jedoch in vollwertige URIs. Ein JSON-LD-kompatibler Client kombiniert hierfür den in der `@vocab`-Eigenschaft definierten Standardnamensraum mit dem Schlüsselnamen. Damit sind alle semantischen Deskriptoren und Link-Relationstypen über URIs eindeutig identifizierbar.

[20] Im Gegensatz zu den selbstdefinierten Link-Relationstypen von hypershop (die Spezifikation bezeichnet diese als *Extension Relation Types*), handelt es sich bei `self` um einen bei der IANA registrierten Link-Relationstypen. Er ist somit eindeutig und wird daher nicht als URI dargestellt (vgl. NOTTINGHAM 2010, S. 15).

```
1  HTTP/1.1 200 OK
2  Content-Type: application/ld+json; charset=utf-8
3
4  {
5    "@context": {
6      "@vocab": "https://example.hypercontract.org/profile/",
7  […]
8      "addToShoppingCart": { "@type": "@id" },
9  […]
10   },
11   "@id": "https://example.hypercontract.org/products/01c31228-d5ba-
   44b3-84b1-7cccc7b9f707",
12   "@type": "Product",
13   "productName": "Handcrafted Steel Mouse",
14   "productDescription": "Deserunt accusamus vitae natus quia. […]",
15   "price": 66.07,
16   "image": "https://[…]",
17   "addToShoppingCart": "https://example.hypercontract.org/
   shoppingCart/items"
18 }
```

Listing 26: Serverantwort auf eine Produkt-Anfrage mit Kopfzeilen (verkürzt) und JSON-LD-Repräsentation

Links werden in JSON-LD, anders als in anderen Hypermedia-Formate, nicht durch eine spezielle Syntax ausgezeichnet, sondern ergeben sich über die Typisierung der Eigenschaft als `@id` im JSON-LD-Kontext (im Beispiel hervorgehoben). Der Linkkontext, der in JSON-HAL über den `self`-Link definiert wurde, ist in JSON-LD über die `@id`-Eigenschaft angegeben.

Im Ergebnis entspricht die JSON-LD-Repräsentation der Produktressource damit dem in *Abbildung 14* dargestellten RDF-Graphen.

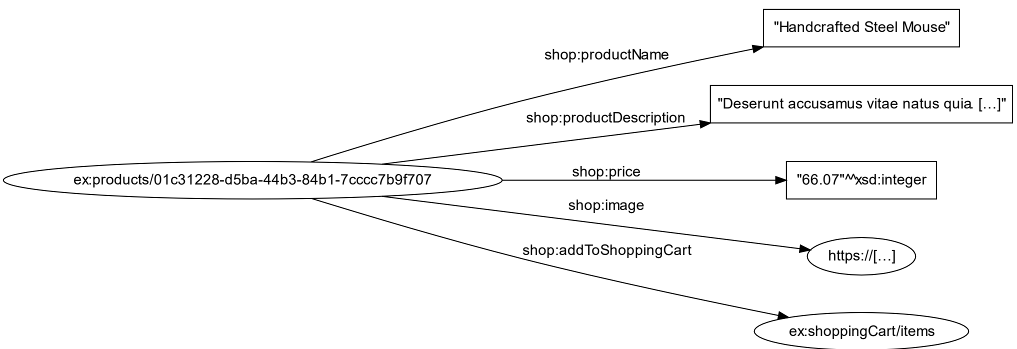


Abbildung 14: Die Produkt-Ressource als RDF-Graph

Der JSON-LD-Kontext muss nicht zwingend in die Repräsentation eingebettet sein. Alternativ kann über einen `Link`-Header auf ein separates JSON-LD-Dokument verwiesen werden, welches den Kontext zur übermittelten JSON-Repräsentation definiert (vgl. Sporny, Kellogg, Lanthaler 2014). Somit ist die in *Listing 27* dargestellte Serverantwort mit JSON-Repräsentation abgesehen vom fehlenden `addToShoppingCart`-Link äquivalent zu *Listing 26*.

```
1   HTTP/1.1 200 OK
2   Content-Type: application/ld+json; charset=utf-8
3   Link: <https://example.hypercontract.org/profile/context.jsonld>;
    rel="http://www.w3.org/ns/json-ld#context"
4
    {
5     "@id": "https://example.hypercontract.org/products/01c31228-d5ba-
    44b3-84b1-7cccc7b9f707",
6     "@type": "Product",
7     "productName": "Handcrafted Steel Mouse",
8     "productDescription": "Deserunt accusamus vitae natus quia. […]",
9     "price": 66.07,
10    "image": "https://[…]",
11    "addToShoppingCart": "https://example.hypercontract.org/
    shoppingCart/items"
12  }
```

Listing 27: Serverantwort auf eine Produkt-Anfrage mit Verweis auf einen JSON-LD-Kontext via Link-Header

Auf diese Weise ist es möglich auch die Deskriptoren der JSON- und JSON-HAL-basierten Nachrichten zu URIs aufzuwerten. Dies ist eine wesentliche Voraussetzung, um sie im folgenden Abschnitt über ein Profil beschreiben zu können.

4.2 Profildefinition mit hypercontract

Durch die Nutzung typisierter Hypermedia-Elemente in den Ressourcenrepräsentationen ist hypershop dem Ziel, eine selbstbeschreibende REST-API anzubieten, einen großen Schritt nähergekommen. Für den menschlichen Betrachter mit Wissen über den fachlichen Kontext ist der Weg durch den Einkaufsprozess dank der Link-Relationstypen bereits nachvollziehbar. Auch die Eigenschaften des Ressourcenzustands sind auf den ersten Blick verständlich.

Bei genauerer Betrachtung bleiben aber Fragen offen. Was muss der Client mit welcher Methode an die mit `shop:addToShoppingCart` referenzierte Ressource schicken, um ein Produkt dem Warenkorb hinzuzufügen? Und was ist als Antwort zu erwarten? Besitzen die `productName`-, `productDescription`- und `price`-Eigenschaften in der `Product`- und der `ShoppingCartItem`-Ressource dieselbe fachliche Bedeutung oder haben sie nur zufällig identische Namen? Ist der `price` des `ShoppingCartItems` der Einzelpreis oder bereits der Gesamtpreis der Position entsprechend der Bestellmenge? Und wie errechnet sich der `totalPrice` des Warenkorbs?

hypercontract beantwortet diese Fragen mit Hilfe eines RDF-basierten Profils, welches den eingesetzten Media Type um zusätzliche Protokoll- und fachliche Semantik ergänzt.

4.2.1 RDF als Beschreibungssprache

Ausgangspunkt bilden die semantischen Deskriptoren und Link-Relationstypen. Da diese eindeutig über URIs identifizierbar sind können sie mit Hilfe von RDF beschrieben werden.

Das Graph-basierte Datenmodell von RDF bietet die notwendige Flexibilität, um die beliebig komplexen fachlichen Konzepte und Zusammenhänge einer Schnittstelle abzubilden. So erübrigt sich die Spezifikation eines neuen Datenformats. Stattdessen genügt der in den folgenden Abschnitten beschriebene Einsatz etablierter RDF-Technologien und die Definition eines hypercontract-spezifischen Vokabulars.

4.2.2 Schnittstellenmechanik

Für die Beschreibung der Schnittstellenmechanik wird zunächst das semantische Datenmodell maschinenlesbar festgehalten. Anschließend erfolgt ausgehend von den identifizierten Zustandsübergängen die Spezifikation der Protokollsemantik. Nach der Dokumentation der Einstiegsressource werden im letzten Schritt die eingesetzten Nachrichtenformate über formatspezifische Datenschemas abgebildet.

4.2.2.1 Datenmodell

Bislang ist das semantische Datenmodell lediglich in grafischer Form (siehe *Abbildung 11*) und als textuelle Beschreibung dokumentiert (siehe *4.1.2 Semantische Deskriptoren*). Um es auch für die maschinelle Verarbeitung zugänglich zu machen, erfolgt eine Abbildung mit Hilfe von RDF Schema und OWL.

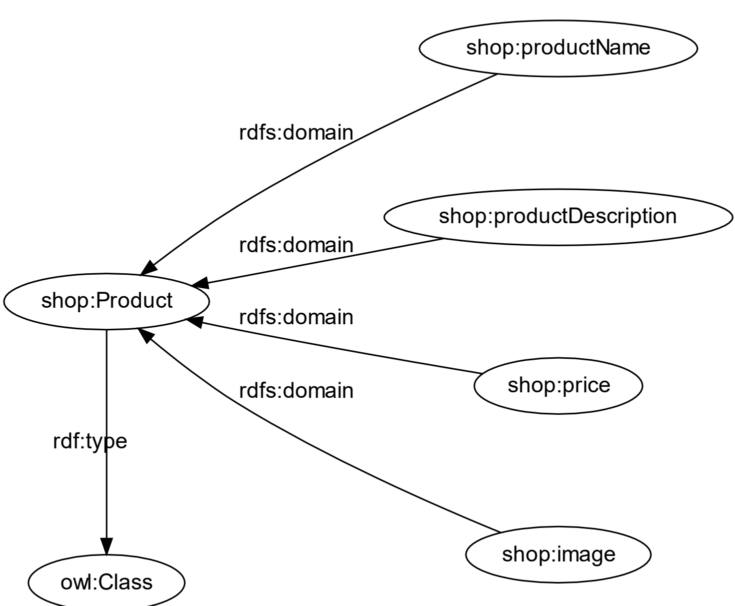


Abbildung 15: Die `Product`-Klasse und ihrer Eigenschaften als RDF-Graph

Die Definition einer Klasse in RDF erfolgt über die Typisierung als `owl:Class`. Eigenschaften werden zunächst unabhängig von der Klasse definiert und anschließend gemäß des Definitionsbereichs aus der Beschreibung der Deskriptoren über `rdfs:domain` mit der Klasse verknüpft. *Abbildung 15* zeigt die Definition der `Product`-Klasse und ihrer Eigenschaften.

Durch die Definition von Eigenschaften als eigenständige Konzepte ist es möglich, dieselbe Eigenschaft mit mehreren Klassen zu verknüpfen. Auf diese Weise wird eindeutig kommuniziert, dass es sich um dasselbe Konzept handelt und nicht um eine zufällig identische Benennung.

Abbildung 16: Die `Product`- und die `ShoppingCartItem`-Klasse mit gemeinsamen Eigenschaften

Das Beispiel in *Abbildung 16* macht deutlich, dass die Eigenschaften `productName`, `productDescription` und `price` von Produkt und Warenkorbposition semantisch identisch sind – im jeweiligen Kontext also dieselbe Bedeutung haben. Dies war aus der Darstellung des semantischen Datenmodells in *Abbildung 11* bislang nicht zu erkennen und wurde erst durch die Beschreibung der semantischen Deskriptoren deutlich.

Die Wertebereiche und Kardinalität der Eigenschaften werden mit den Mitteln von RDF Schema und OWL beschrieben. *Abbildung 17* zeigt dies am Beispiel einiger Eigenschaften von Warenkorb und Warenkorbposition.

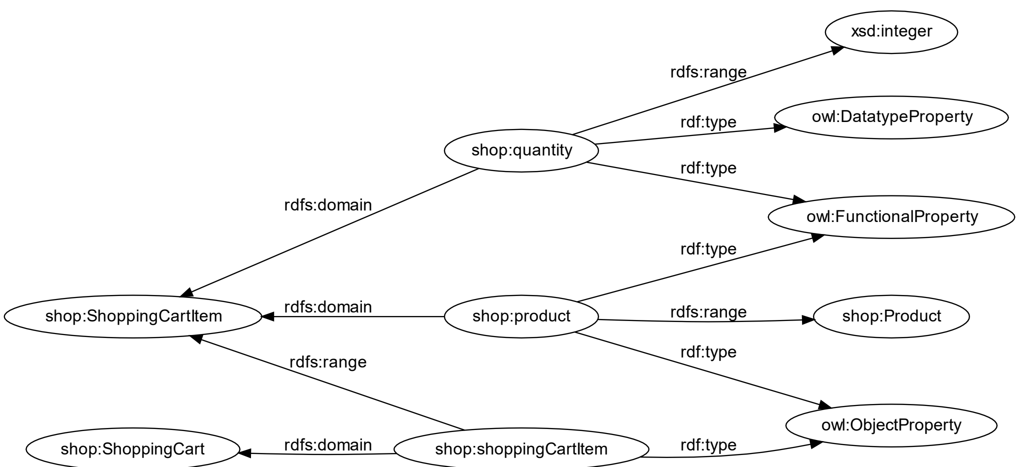


Abbildung 17: Definition von Eigenschaften am Beispiel von Warenkorb und Warenkorbposition

Die `quantity`-Eigenschaft verweist auf einen ganzzahligen Wert und wird daher als `DatatypeProperty` mit dem Wertebereich `xsd:integer` definiert. Die Eigenschaften `product` und `shoppingCartItem` hingegen stellen eine Beziehung zu einer anderen Ressource her. Sie sind deshalb `ObjectPropertys` und haben als Wertebereich die `Product`- beziehungsweise die `ShoppingCartItem`-Klasse. Da ein Warenkorb mehr als eine Warenkorbposition haben kann, wird `shoppingCartItem` im Gegensatz zu `product` nicht als `FunctionalProperty` typisiert.

4.2.2.2 Zustandsübergänge

Die Abbildung der fachlichen Zustandsübergänge (siehe *4.1.3 Zustandsübergänge*) auf Basis des Applikationsprotokolls (HTTP) ist die Protokollsemantik der Schnittstelle.[21] Bei der Betrachtung der Zustandsübergänge und deren Dokumentation mit RDF muss zwischen den sicheren und den unsicheren Zustandsübergängen unterschieden werden.

Ein sicherer Zustandsübergang erfolgt immer dann, wenn auf die verlinkte Ressource mit einer als sicher definierten Methode zugegriffen wird – im Kontext von HTTP also mit `GET`. Eine Abfrage der Ressource verändert demnach nicht den Ressourcenzustand. Ein unsicherer Zustandsübergang geschieht dementsprechend beim Zugriff auf eine Ressource mit einer Methode, die als unsicher spezifiziert ist – beispielsweise mit den HTTP-Methoden `POST`, `PUT` oder `DELETE` (siehe *2.1.2 Interaktion via HTTP*).

hypercontract bildet sichere Zustandsübergänge als `ObjectPropertys` ab. Die in *Abbildung 17* beschriebene `product`-Eigenschaft kann daher als Link-Relationstyp für

[21] Grundsätzlich ist dieser Teil des Profils optional, sofern ein Hypermedia-fähiger Media Type eingesetzt wird, welcher alle notwendigen Informationen als Teil der Repräsentation kodiert (vgl. Richardson, Amundsen 2013, S. 138).

einen Verweis von einem `ShoppingCartItem` auf die zugehörige `Product`-Ressource genutzt werden (siehe *Listing 28*).

```
1  {
2    "productName": "Handcrafted Steel Mouse",
3    "productDescription": "Deserunt accusamus vitae natus quia. […]",
4    "price": 66.07,
5    "quantity": 2,
6    "_links": {
7      "self": {
8        "href": "https://example.hypercontract.org/shoppingCart/
   items/efe6acab-0047-40ea-8f2a-dfddd53a8fa5"
9      },
10     "https://example.hypercontract.org/profile/product": {
11       "href": "https://example.hypercontract.org/products/
   01c31228-d5ba-44b3-84b1-7cccc7b9f707"
12     },
   […]
13   }
14 }
```

Listing 28: JSON-HAL-Repräsentation einer Warenkorbposition mit Link auf das Produkt

Die Definition von `product` als `ObjectProperty` und `FunctionalProperty` signalisiert dem Client, dass der entsprechende Link ein sicherer Zustandsübergang ist und daher mit `GET` erfolgt. Der `range`-Wert von `product` dokumentiert zudem, dass der Aufruf die Repräsentation einer `Product`-Ressource liefert (siehe *Abbildung 18*).

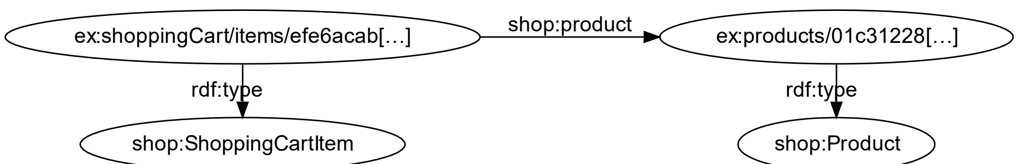


Abbildung 18: Darstellung des `product`-Links von der Warenkorbposition auf das Produkt als RDF-Graph

Für die Beschreibung von unsicheren Zustandsübergängen reicht dies nicht aus, da im Kontext von HTTP nicht eindeutig ist, mit welcher Methode die Abfrage der verlinkten Ressourcen erfolgen muss. Außerdem stellt sich die Frage, wie Repräsentationen beschrieben werden, die als Teil der Anfrage an den Server gesendet werden.

hypercontract stellt daher ein Vokabular bereit, mit dem Zustandsübergänge als `Operation` spezifiziert werden können. Request Body und Query-Parameter des Aufrufs werden über die Eigenschaften `expectedBody` und `expectedQueryParams` dokumentiert.

Zusätzlich ist die Angabe eines `returnedType` möglich. Dies ist erforderlich, da die `range`-Eigenschaft von RDF Schema den Typen der verlinkten Ressource definiert. Es ist aber möglich, dass der Client über einen `Location`-Header auf eine andere

Ressource umgeleitet wird. In dem Fall wäre der `range`-Wert entweder missverständlich oder nicht korrekt.

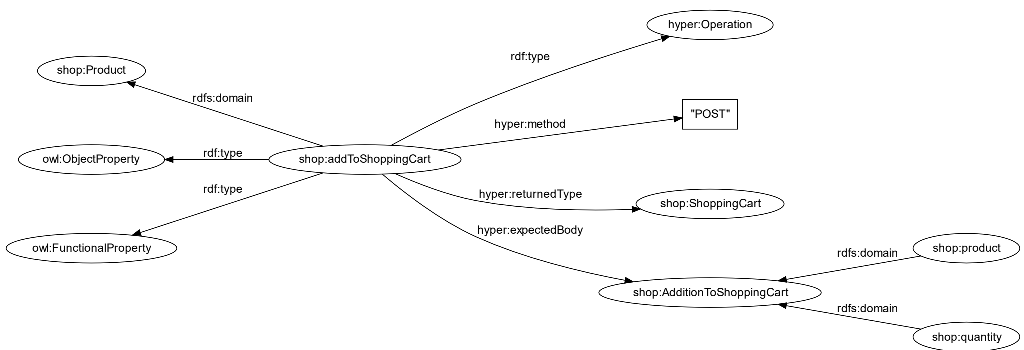


Abbildung 19: Beschreibung der `addToShoppingCart`-Operation mit Hilfe des hypercontract-Vokabulars

Abbildung 19 zeigt, wie das Hinzufügen eines Produkts zu Warenkorb mit hypercontract als `Operation` beschrieben wird. Zunächst wird `addToShoppingCart` der `Product`-Klasse zugeordnet und als `ObjectProperty` und `FunctionalProperty` definiert. Zusätzlich erfolgt eine Typisierung als `Operation`. Diese erwartet eine Anfrage via `POST` und einen Request Body, dessen Datenstruktur der `AdditionToShoppingCart`-Klasse entspricht. Für die Angabe von Produkt und Bestellmenge werden die Eigenschaften `product` und `quantity` genutzt. Hierbei handelt es sich um dieselben Eigenschaften, die auch beim `ShoppingCartItem` zum Einsatz kommen (siehe *4.2.2.1 Datenmodell*). Als Antwort auf die Anfrage erhält der Client den (aktualisierten) `ShoppingCart`. *Listing 29* zeigt eine Anfrage, die dieser Spezifikation entspricht.

```
1   POST /shoppingCart/items HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: application/hal+json
4   Content-Type: application/json
5
6   {
7     "product": "https://example.hypercontract.org/products/01c31228-
    d5ba-44b3-84b1-7cccc7b9f707",
8     "quantity": 1
9   }
```

Listing 29: Clientanfrage für das Hinzufügen eines Produkts zum Warenkorb

In *Listing 30* ist zu sehen, wie der Server dem dokumentierten `returnedType` gerecht wird, indem er über den `Location`-Header auf den Warenkorb umleitet.

```
1   HTTP/1.1 201 Created
2   Location: https://example.hypercontract.org/shoppingCart
```

Listing 30: Serverantwort auf das Anlegen einer Warenkorbposition mit Umleitung auf den Warenkorb

Analog zum `expectedBody` lassen sich mit `expectedQueryParams` die Query-Parameter einer URL spezifizieren. *Abbildung 20* zeigt die Definition des Parameters `queryString` für den Link-Relationstyp `searchCatalog`.

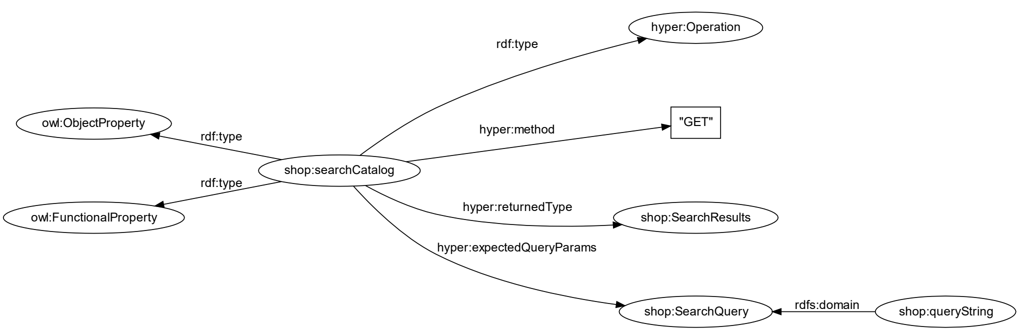


Abbildung 20: Beschreibung der `searchCatalog`-Operation mit Hilfe des hypercontract-Vokabulars

Im Gegensatz zu `addToShoppingCart` ist die Katalogsuche ein sicherer Zustandsübergang, zu erkennen an der `GET`-Methode. Sie wird hier dennoch explizit als `Operation` typisiert, um die einfache Beziehung zweier Ressourcen (wie im Falle von `product`) von einem parametrisierten Zustandsübergang unterscheiden zu können.

4.2.2.3 Einstiegs-URL

Ein wesentliches Merkmal von REST-APIs ist die Tatsache, dass ein Client die URLs einer Schnittstelle zur Laufzeit über Links in den Ressourcenrepräsentationen erfährt. Analog zum Besucher einer Website muss er daher nur eine einzige URL kennen: die Einstiegs-URL. In hypercontract wird diese als Ressource vom Typ `EntryPoint` dokumentiert. (siehe *Abbildung 21*). Die Zustandsübergänge, die von dieser Startressource aus potenziell möglich sind, werden mit den bekannten Mitteln aus *4.2.2.2 Zustandsübergänge* spezifiziert. Im Falle von hypershop existiert hierfür die `ApiRoot`-Klasse, welche über `rdfs:domain` mit den entsprechenden Link-Relationstypen verknüpft ist.

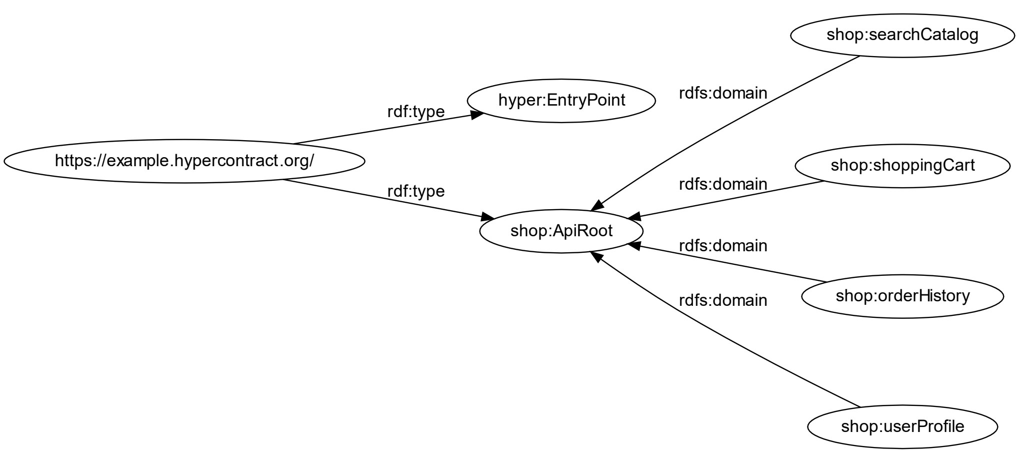


Abbildung 21: Beschreibung der Einstiegsressource der hypershop-API als `EntryPoint`

Der Einstiegspunkt von hypershop verweist demnach auf die Katalogsuche, den Warenkorb, die bereits getätigten Bestellungen und das Benutzerprofil.

4.2.2.4 Datenschemas

Durch die Abbildung der Fachkonzepte als OWL-Klassen und ihrer Eigenschaften als RDF-Propertys wird das semantische Datenmodell zusammen mit den Zustandsübergängen bereits nachvollziehbar dokumentiert. Auch der Einstiegspunkt der Schnittstelle ist als `EntryPoint` eindeutig gekennzeichnet. Für einen zuverlässigen Contract fehlen aber formatspezifische Beschreibungen der Nachrichteninhalte (siehe *4.1.5 Repräsentationsformate*). Um diese Lücke zu schließen, können Eigenschaften und Klassen mit Schemas beschrieben werden.

Grundsätzlich ist denkbar hierfür die erweiterten Sprachmittel von OWL zu nutzen. Am Ende steht aber immer ein Übersetzungsaufwand, um die in OWL formulierte Spezifikation auf das Zielformat zu übertragen. Auch lassen sich nie alle Konzepte der einen Sprache verlustfrei in jeder anderen Sprache ausdrücken.

Daher gibt hypercontract selbst keine Schematechnologie vor, sondern ermöglicht lediglich das Referenzieren von Datenschemas auf Ebene von Eigenschaften und Klassen. Die eingesetzte Schematechnologie kann so passend zum eingesetzten Nachrichtenformat gewählt werden. Im Falle einer JSON-basierten API ist der Einsatz von JSON Schema naheliegend, bei XML-basierten Repräsentationen bietet sich XML Schema an.

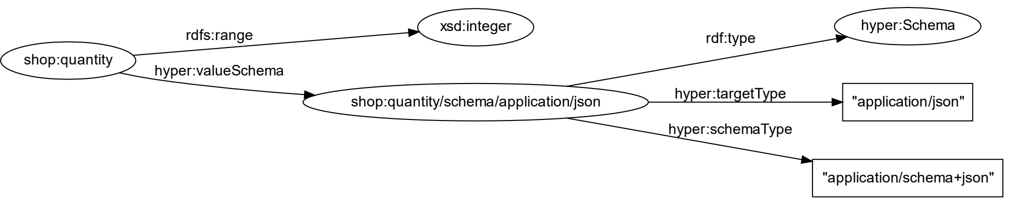


Abbildung 22: Verknüpfung der `quantity`-Eigenschaft mit einem JSON-Schema

Abbildung 22 zeigt wie für die `quantity`-Eigenschaft über `valueSchema` (zusätzlich zur `range`-Angabe) ein passendes JSON-Schema definiert wird, welches den Wertebereich der Eigenschaft exakt spezifiziert. Das Schema stellt eine Ressource dar, die über `targetType` den Media Type angibt, für den das Schema gilt. So können für verschiedene Repräsentationsformate unterschiedliche Schemas definiert werden. Das Format des Schemas wird über die `schemaType`-Eigenschaft angegeben. Der Zugriff auf die eigentliche Schemadefinition erfolgt über eine Anfrage an die Schema-Ressource (siehe *4.3.1 Bereitstellung des Profils*).

Listing 31 zeigt das Schema für Werte der `quantity`-Eigenschaft für das Zielformat JSON. Die Definition könnte in diesem Beispiel in gleicher Form auch für JSON-HAL und JSON-LD verwendet werden. Da dies jedoch nicht grundsätzlich für alle Eigenschaften zutrifft, werden sie aus Sicht des API-Nutzers als separate Schema-Ressourcen abgebildet.

```
1  {
2    "$schema": "http://json-schema.org/draft-07/schema#",
3    "$id": "https://example.hypercontract.org/profile/quantity/
   schema/application/json"
4    "type": "number",
5    "multipleOf": 1,
6    "minimum": 1
7  }
```

Listing 31: Schemadefinition für die `quantity`-Eigenschaft als JSON Schema

Die Definition von Schemas auf Ebene einzelner Eigenschaften geschieht, um eine Wiederverwendung auf Klassenebene zu ermöglichen. Eine Klasse kann mit einem für sie passenden Schema über die `instanceSchema`-Eigenschaft verknüpft werden. Dieses kann dann wiederum auf Schemas für die zugehörigen Eigenschaften verweisen.

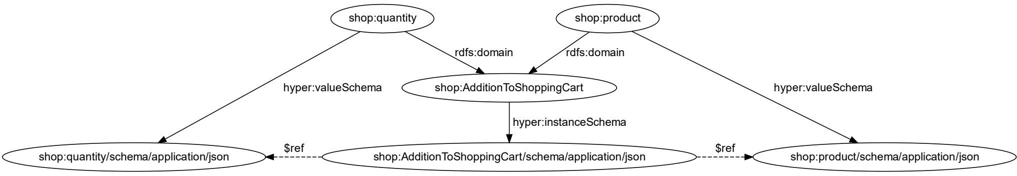


Abbildung 23: Verknüpfung der `AdditionToShoppingCart`-Klasse mit einem JSON-Schema

Das Beispiel in *Abbildung 23* zeigt den Unterschied zwischen der Definition von Schemas für Eigenschaftswerte (`valueSchema`) und solchen für Instanzen einer Klasse (`instanceSchema`). Wie die zuvor erwähnte Wiederverwendung auf Klassenebene aussieht, hängt von der Schemasprache ab. In diesem Beispiel geschieht dies mit der `$ref`-Funktionalität von JSON Schema (vgl. Wright, Andrews 2018, S. 14). In *Listing 32* wird über dieses Feature das Schema für `AdditionToShoppingCart` aus den Schemas von `product` und `quantity` zusammengesetzt.

```
1  {
2    "$schema": "http://json-schema.org/draft-07/schema#",
3    "$id": " https://example.hypercontract.org/profile/
   AdditionToShoppingCart/schema/application/json",
4    "type": "object",
5    "properties": {
6      "product": {
7        "$ref": "https://example.hypercontract.org/profile/product/
   schema/application/json"
8      },
9      "quantity": {
10       "$ref": "https://example.hypercontract.org/profile/quantity/
   schema/application/json"
11     }
12   },
13   "required": [
14     "product",
15     "quantity"
16   ]
17 }
```

Listing 32: Schemadefinition für die `AdditionToShoppingCart`-Klasse als JSON Schema

`instanceSchema` kann auch genutzt werden, um zu spezifizieren, wie Klassen auf Query-Parameter in der URL abgebildet werden. In *Abbildung 24* verweist die `SearchQuery`-Klasse auf ein Schema für den Media Type `text/uri-list` (vgl. MEALLING, DANIEL 1999, S. 11). Die Schemadefinition erfolgt als XML-basierte *URI Template Description*, welche das URI-Template `?{queryString}` beschreibt. Entsprechend ist der Schematyp als `application/td+xml` angegeben (vgl. GREGORIO U. A. 2012; WILDE, DAVIS, LIU 2012, S. 5).[22]

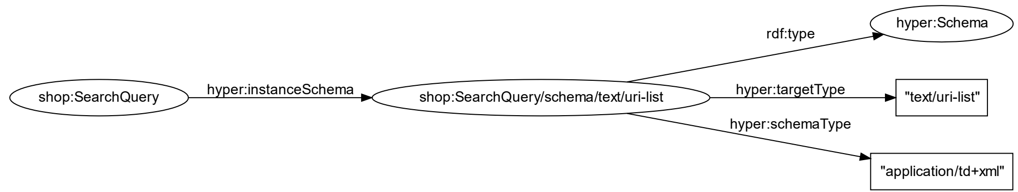


Abbildung 24: Verknüpfung der `SearchQuery`-Klasse mit einem URI-Template

Datenschemas sind der letzte Baustein, der notwendig ist, um zusammen mit der Definition des Datenmodells und der Zustandsübergänge die Schnittstellenmechanik einer REST-API maschinenlesbar festzuhalten. Im Ergebnis ist ein RDF-Graph entstanden, auf dessen Basis im folgenden Abschnitt die fachliche Semantik der Schnittstelle beschrieben wird.

---

[22] URI Templates bieten eine Syntax, mit der sich der Aufbau von URIs beschreiben lässt. WILDE, DAVIS, LIU 2012 schlagen einen XML-basierten Standard für die Beschreibung solcher URI-Templates vor, welcher es erlaubt, die Bestandteile eines URI-Templates wiederum als URIs eindeutig zu identifizieren. URI Template Descriptions besitzen im Gegensatz zur URI-Template-Spezifikation einen Media Type, sind allerdings weder standardisiert noch vollständig spezifiziert. Alternativ wäre daher denkbar, die Registrierung eines Media Types für die URI-Template-Spezifikation voranzutreiben.

4.2.3 Fachliche Schnittstellensemantik

Damit der Client eine Schnittstelle sinnvoll nutzen kann, muss er deren fachliche Semantik verstehen. Durch die eindeutige Identifizierung einzelner Schnittstellenelemente über URIs ist bereits ein kleiner Schritt in diese Richtung getan: Für Mensch wie Maschine ist beispielsweise verständlich, dass die `quantity`-Eigenschaft sowohl in der `AdditionToShoppingCart`-Klasse als auch im `ShoppingCartItem` dieselbe fachliche Bedeutung hat, da es in beiden Fällen dieselbe Eigenschaft ist. Offen bleibt jedoch die Frage, wie diese fachliche Bedeutung definiert ist.

Üblicherweise erfährt der Nutzer einer API dies aus einer parallel bereitgestellten Dokumentation, im schlimmsten Fall muss er die Schnittstelle per Trial-and-Error explorativ erkunden und läuft dabei Gefahr, falsche Annahmen zu treffen oder ungewollte Effekte auszulösen.

hypercontract dokumentiert die Elemente einer Schnittstelle stattdessen im Kontext ihrer Nutzung. Da sie bereits im Rahmen der Schnittstellenmechanik über URIs eindeutig definiert werden, bietet es sich an, die RDF-Schema-Eigenschaft `rdfs:comment` für eine semantische Beschreibung heranzuziehen. *Abbildung 25* illustriert dies anhand der `AdditionToShoppingCart`-Klasse.

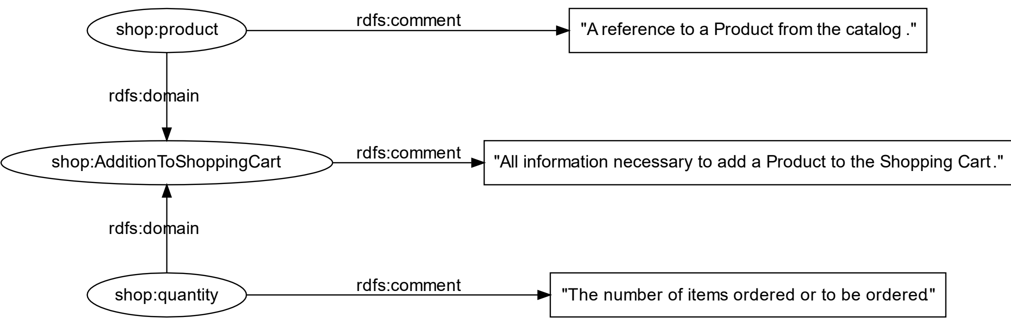


Abbildung 25: Beschreibung der `AdditionToShoppingCart`-Klasse und ihrer Eigenschaften mit `rdfs:comment`

Zukünftig kann dies im Sinne einer nutzerfreundlichen Dokumentation dank Sprachinformationen an den Literalen auch mehrsprachig geschehen (siehe *Abbildung 26*).

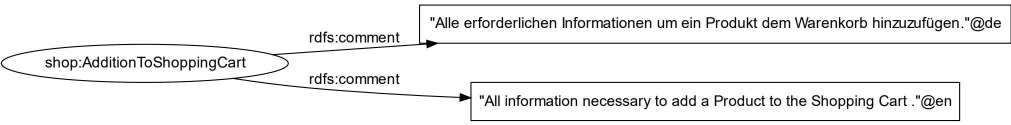


Abbildung 26: Mehrsprachige Beschreibung der `AdditionToShoppingCart`-Klasse

Mit der `Precondition`-Klasse bietet hypercontract darüber hinaus eine Möglichkeit, die Vorbedingungen von Operationen zu spezifizieren. In *Abbildung 27* wird die Bedingung, die zum Bestellen eines Warenkorbs erfüllt sein muss, in Form des `isShoppingCartOrderable`-Konzepts als `Precondition` definiert und mit `rdfs:comment` fachlich beschrieben.

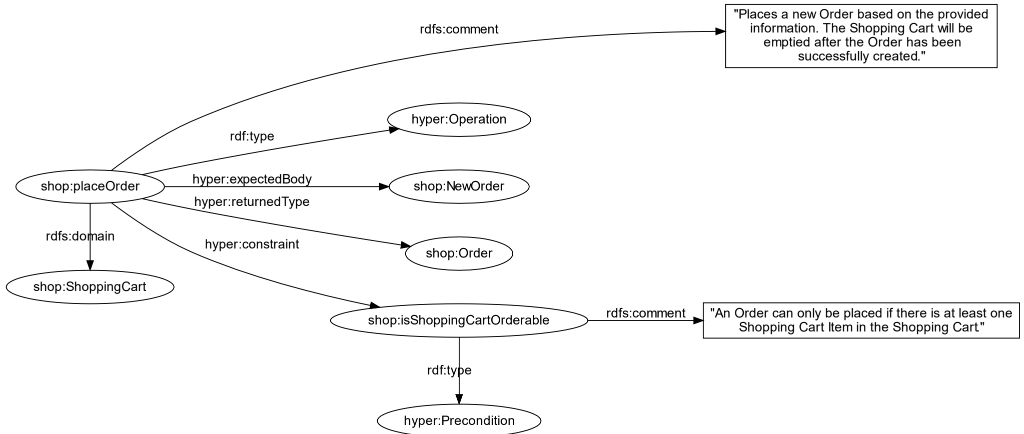


Abbildung 27: Abbildung der Vorbedingung für das Bestellen eines Warenkorbs als `hyper:Precondition`

Dank der offenen Natur von RDF können auch weitere Vokabulare zur semantischen Beschreibung der Schnittstellenkonzepte herangezogen werden. Mit Hilfe des SKOS-Standards lassen sich beispielsweise Beziehungen zu Konzepten aus anderen fachlichen Vokabularen herstellen (vgl. MILES, BECHHOFER 2009), sofern nicht schon im Rahmen von *4.1.4* Einheitliches *Vokabular* von vornherein auf ein bestehendes Vokabular gesetzt wurde.

Werden alle Konzepte der Schnittstelle – Klassen, Eigenschaften, Zustandsübergänge, Vorbedingungen – auf diese Weise beschrieben, ergibt sich ein weitgehend vollständiges Bild des fachlichen Kontextes der Schnittstelle. Für eine umfassende Beschreibung der Konzepte der hypershop-API sei auf Anhang *C Semantische Deskriptoren von hypershop* und Anhang *D Zustandsübergänge von hypershop* verwiesen. Einen Überblick über die Klassen und Eigenschaften des hypercontract-Vokabulars gibt *E hypercontract Vokabular*.

4.3 Integration in die Schnittstelle

Das Ergebnis von *4.2 Profildefinition mit hypercontract* ist ein Profil, welches in Kombination mit dem über HTTP ausgehandelten Media Type den Contract zwischen Client und Server beschreibt. Nun gilt es dieses Profil in die Schnittstelle zu integrieren.

4.3.1 Bereitstellung des Profils

Für die Bereitstellung erhält das Profil einen URI, über welchen der vollständige RDF-Graph veröffentlicht wird. Im Falle von hypershop liefert eine Anfrage an `https://example.hypercontract.org/profile` somit eine Serialisierung des RDF-Modells. Das Serialisierungsformat wird via Content Negotiation ausgehandelt und ist entweder eine konkrete RDF-Syntax, wie beispielsweise JSON-LD, oder eine menschenlesbare Darstellung in HTML.

```
1   HTTP/1.1 200 OK
2   Content-Type: application/ld+json; charset=utf-8
3
4   {
5     "@context": {
6       "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
7       "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
8       "owl": "http://www.w3.org/2002/07/owl#",
9       "xsd": "http://www.w3.org/2001/XMLSchema#",
10      "hyper": "https://hypercontract.org/",
11      "shop": "https://example.hyperconract.org/profile/",
    […]
12    },
13    "@graph": [
    […]
14      {
15        "@id": "shop:addToShoppingCart",
16        "@type": [
17          "owl:FunctionalProperty",
18          "owl:ObjectProperty",
19          "hyper:StateTransition",
20          "hyper:Operation"
21        ],
22        "rdfs:comment": "Adds a Product as a new Shopping Cart […]",
23        "rdfs:domain": "shop:Product",
24        "rdfs:label": "add to shopping cart",
25        "rdfs:range": "shop:ShoppingCartItems",
26        "hyper:expectedBody": "shop:AdditionToShoppingCart",
27        "hyper:method": "POST",
28        "hyper:returnedType": "shop:ShoppingCart",
29        "hyper:valueSchema": [
30          "shop:addToShoppingCart/schema/application/hal+json",
31          "shop:addToShoppingCart/schema/application/ld+json"
32        ]
33      },
34  […]
35    ]
36  }
```

Listing 33: Serverantwort (verkürzt) auf eine Abfrage des hypershop-Profils als JSON-LD

Die in *Listing 33* dargestellte Ausschnitt der JSON-LD-Serialisierung des Profils zeigt beispielhaft den aus *4.2.2.2 Zustandsübergänge* bekannten Link-Relationstypen `addToShoppingCart`.[23]

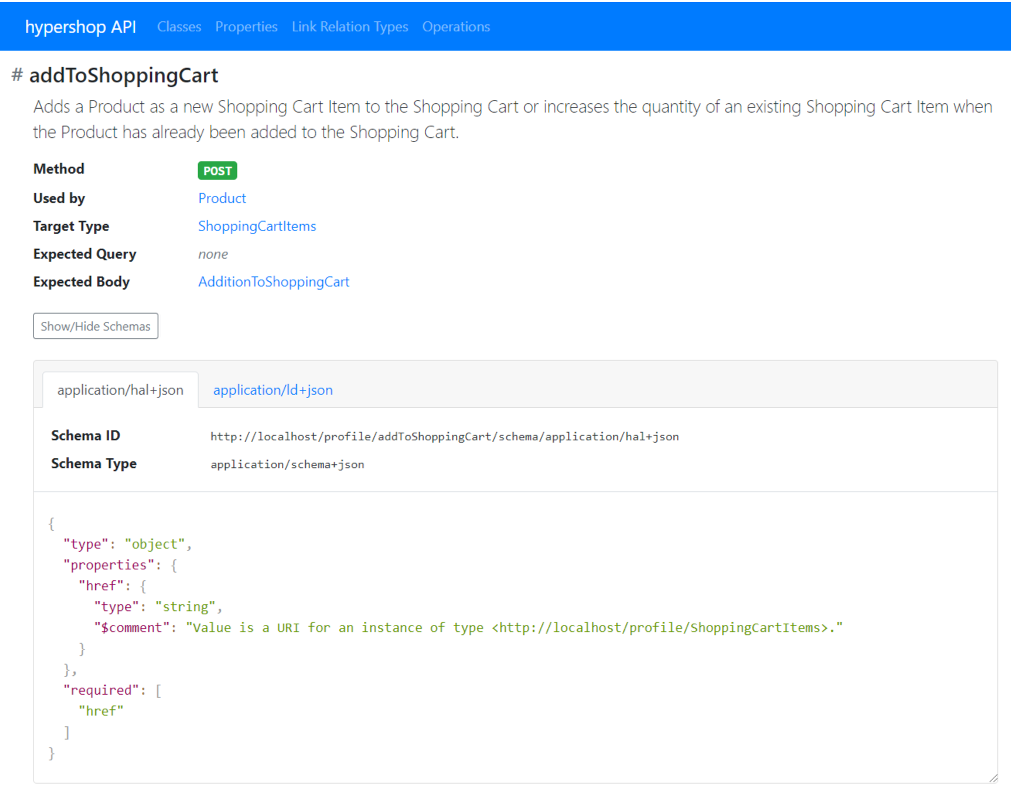


Abbildung 28: Darstellung des hypershop-Profils als HTML-Seite

Abbildung 28 zeigt den gleichen Link-Relationstyp in der vom Browser gerenderten HTML-Repräsentation. Über Links kann der Leser zwischen den Konzepten innerhalb des Dokuments navigieren und so beispielsweise von der `addToShoppingCart`-Definition zur Beschreibung der `AdditionToShoppingCart`-Klasse wechseln.

Zur Dokumentation von Klassen und Eigenschaften gehören auch die Schemadefinitionen, die im Falle der HTML-Darstellung als Teil des Dokuments angezeigt werden (siehe *Abbildung 28*). In der RDF-Repräsentation sind sie als `rdf:value` enthalten, können aber über eine Anfrage an die Schema-Ressource mit entsprechendem `Accept`-Header auch im eigentlichen Format angefordert werden.

[23] Die Beschreibung wird in englischer Sprache ausgeliefert. Über die Angabe des Request-Headers `Accept-Language` könnte aber zukünftig auch eine andere Sprache angefordert werden (vgl. Fielding, Reschke 2014, S. 42). Die hier vorstellte prototypische Implementierung unterstützt dies jedoch noch nicht.

```
1  GET /profile/AdditionToShoppingCart/schema/application/json
   HTTP/1.1
2  Host: example.hypercontract.org
3  Accept: application/schema+json
4
5  HTTP/1.1 200 OK
6  Content-Type: application/schema+json
7
8  {
9    "$schema": "http://json-schema.org/draft-07/schema#",
10   "$id": "https://example.hypercontract.org/profile/
   AdditionToShoppingCart/schema/application/json",
11   "type": "object",
12   "properties": {
13     "product": {
14       "$schema": "http://json-schema.org/draft-07/schema#",
15       "$id": "https://example.hypercontract.org/profile/
   product/schema/application/json",
16       "type": "string",
17       "$comment": "Value is a URI for an instance of type
   <https://example.hypercontract.org/profile/Product>."
18     },
19     "quantity": {
20       "$schema": "http://json-schema.org/draft-07/schema#",
21       "$id": "https://example.hypercontract.org/quantity/schema/
   application/json",
22       "type": "number",
23       "multipleOf": 1,
24       "minimum": 1
25     }
26   },
27   "required": [
28     "product",
29     "quantity"
30   ]
31 }
```

Listing 34: Clientanfrage und Serverantwort (verkürzt) für das JSON-Schema der `AdditionToShoppingCart`*-Klasse*

Listing 34 illustriert dies anhand der Schemadefinition für die `AdditionToShoppingCart`-Klasse aus *4.2.2.4 Datenschemas*. Im Rahmen der Bereitstellung des Schemas werden die `$ref`-Verweise in der ursprünglichen Definition dereferenziert. Die Schemadefinitionen der Eigenschaften `product` und `quantity` sind somit Teil der Repräsentation, auch wenn sie eigentlich eigenständige Definitionen darstellen.

4.3.2 Verknüpfung mit der Schnittstelle

Den Prinzipien des Webs folgend wird das Profil über Links mit der Schnittstelle verknüpft. Das ermöglicht eine lose Kopplung zwischen Schnittstelle und Dokumentation – sowohl im Hinblick auf Auslieferung und Betrieb als auch auf die technische Implementierung.

4.3.2.1 Verknüpfung über Link-Header

Die Verknüpfung des Profils mit der REST-Schnittstelle von hypershop erfolgt grundsätzlich zunächst über einen `Link`-Header in allen Antworten des Servers.

```
1  HTTP/1.1 200 OK
2  Content-Type: application/ld+json; charset=utf-8
3  Link: <https://example.hypercontract.org/profile>; rel="profile"
```

Listing 35: Serverantwort (verkürzt) mit `Link`-Header für das Profil

Listing 35 zeigt die Kopfzeilen einer Serverantwort der hypershop-API. Der mit `profile` typisierte `Link`-Header verweist auf den URI des Profils (vgl. WILDE 2013, S. 5). Durch die Nutzung von JSON-LD als Austauschformat ist mit diesem Verweis die Anforderung einer selbstbeschreibenden Schnittstelle bereits erfüllt. Dank Media Type und Profil weiß der Client alles Notwendige, um mit der API interagieren zu können.

Wird statt JSON-LD ein weniger aussagekräftiges JSON-Format eingesetzt, sind zusätzliche Informationen erforderlich. Die fehlende Unterstützung für Hypermedia-Elemente im Falle von JSON könnte in einer zukünftigen Version von hypershop mit Hilfe von JSON Hyper-Schema ergänzt werden. Dabei wird über einen `describedby`-Link in den Kopfzeilen der Serverantwort auf ein JSON-Schema verwiesen, mit welchem sich Links in Form von Link Description Objects abbilden lassen (vgl. WRIGHT, ANDREWS 2018).

Die aktuelle Implementierung beschränkt sich darauf, JSON-basierte Nachrichten mit einem Link auf den JSON-LD-Kontext im Response Header auszuzeichnen (siehe *Listing 36*).

```
1  HTTP/1.1 200 OK
2  Content-Type: application/json; charset=utf-8
3  Link: <https://example.hypercontract.org/profile>; rel="profile",
   <https://example.hypercontract.org/context.jsonld>;
4  rel="http://www.w3.org/ns/json-ld#context"
```

Listing 36: Serverantwort (verkürzt) mit `Link`-Header für Profil und JSON-LD-Kontext

Durch diesen minimalen Eingriff wird der Nachrichteninhalt zu einem RDF-Graphen aufgewertet und alle Deskriptoren über URIs eindeutig identifizierbar (siehe *2.4.2 Serialisierung von RDF-Daten*).

4.3.2.2 Dereferenzieren semantischer Deskriptoren

Neben der Verlinkung des vollständigen Profils bietet es sich im Sinne der Prinzipien von *Linked Data* an, dass auch die URIs der einzelnen Konzepte dereferenzierbar sind (siehe *2.4 RDF und Linked Data*). Damit ist es möglich, die Definition von semantischen Deskriptoren und den damit verbundenen Konzepten im Kontext der

Nutzung zu verlinken. Eine Anfrage an den URI eines Konzepts liefert daher alle RDF-Statements des Profils, die das Konzept beschreiben (siehe *Listing 37*).

```
1  GET /profile/addToShoppingCart HTTP/1.1
2  Host: example.hypercontract.org
3  Accept: application/ld+json

4  HTTP/1.1 200 OK
5  Content-Type: application/ld+json; charset=utf-8
6
7  {
8    "@context": {
9      "rdf": "http://www.w3.org/1999/02/22-rdf-syntax-ns#",
10     "rdfs": "http://www.w3.org/2000/01/rdf-schema#",
11     "owl": "http://www.w3.org/2002/07/owl#",
12     "xsd": "http://www.w3.org/2001/XMLSchema#",
13     "hyper": "https://hypercontract.org/",
14     "shop": "https://example.hypercontract.org/profile/",
15 […]
16   },
17   "@id": "shop:addToShoppingCart",
18   "@type": [
19     "owl:FunctionalProperty",
20     "owl:ObjectProperty",
21     "hyper:StateTransition",
22     "hyper:Operation"
23   ],
24   "rdfs:comment": "Adds a Product as a new Shopping Cart […]",
25   "rdfs:domain": "shop:Product",
26   "rdfs:label": "add to shopping cart",
27   "rdfs:range": "shop:ShoppingCartItems",
28   "hyper:expectedBody": "shop:AdditionToShoppingCart",
29   "hyper:method": "POST",
30   "hyper:returnedType": "shop:ShoppingCart",
31   "hyper:valueSchema": [
32     "shop:addToShoppingCart/schema/application/hal+json",
33     "shop:addToShoppingCart/schema/application/ld+json"
34   ]
35 }
```

Listing 37: Clientanfrage und Serverantwort (verkürzt) für die JSON-LD-Repräsentation der `AddToShoppingCart`-Operation

Fragt der Client eine HTML-Repräsentation an, erfolgt eine Umleitung auf das vollständige Profil. Dabei wird die URL des Profils zusätzlich um einen Fragmentbezeichner ergänzt, der auf die Konzeptbeschreibung innerhalb des HTML-Dokuments verweist (siehe *Listing 38*).[24]

[24] Bei Abfragen der RDF-Repräsentation ist dies nicht möglich, da die Bedeutung von Fragmentbezeichnern abhängig vom Media Type ist und sie für RDF-Formate anders interpretiert werden als im Falle von HTML. Beim Aufruf eines HTML-Dokuments verweist der Fragmentbezeichner auf eine Sprungmarke, die über das `id`- oder `name`-Attribut definiert wird. Bei RDF-Formaten wird der Fragmentbezeichner mit der Aufruf-URL kombiniert und identifiziert so ein Konzept innerhalb des RDF-Dokuments über dessen URI. `https://example.hypercontract.org/profile#addToShoppingCart` verweist somit auf ein Konzept innerhalb des Dokuments mit identischer URI, der tatsächliche URI des Link-Relationstypen ist jedoch `https://example.hypercontract.org/profile/addToShoppingCart` (vgl. KLYNE, CARROLL 2004).

```
1   GET /profile/addToShoppingCart HTTP/1.1
2   Host: example.hypercontract.org
3   Accept: text/html

4   HTTP/1.1 303 See Other
5   Location:
    https://example.hypercontract.org/profile#addToShoppingCart
```

Listing 38: Clientanfrage und Serverantwort (verkürzt) für die HTML-Repräsentation der `AddToShoppingCart`-Operation

Grundsätzlich ist auch die Nutzung von Hash-URIs direkt im Profil möglich (vgl. SAUERMANN, CYGANIAK 2008). Über die 303-Umleitung entsteht jedoch eine Fassade, die zukünftige Refactorings ermöglicht.

In gleicher Weise wird auch das Vokabular von hypercontract veröffentlicht. Dies geschieht jedoch über eine Infrastruktur, die von der Beispielapplikation unabhängig ist.

4.3.3 Semantische Annotation von HTML-Elementen

Das Profil ist grundsätzlich agnostisch gegenüber dem eingesetzten Media Type. Dies lässt sich sehr anschaulich daran zeigen, dass es zur semantischen Annotation von HTML-Dokumenten genutzt werden kann. Neben der Auszeichnung von Hypermedia-Elementen über das `rel`-Attribut können auch die semantischen Deskriptoren des HTML-Standards, die vorwiegend für die Auszeichnung von Dokumenten entworfen wurden, dank Microdata mit zusätzlicher Semantik aus dem fachlichen Kontext der Anwendung angereichert werden. Dabei handelt es sich um eine Erweiterung von HTML mit der sich maschinenlesbare Metadaten in ein HTML-Dokument einbetten lassen (vgl. NEVILE, BRICKLEY 2018). Über die Attribute `itemtype`, `itemid` und `itemprop` können hierfür HTML-Elemente mit Konzepten aus einem Vokabular verknüpft werden.

```
1   <!DOCTYPE html>
2   <html>
3     <head>[…]</head>
4     <body>
    […]
5       <main itemscope
6     itemtype="https://example.hypercontract.org/profile/Product"
7     itemid="https://example.hypercontract.org/products/01c31228-d5ba-
    44b3-84b1-7cccc7b9f707">
8         <h1 itemprop="productName">Handcrafted Steel Mouse</h1>
9   […]
```

```
10          <figure>
11            <img itemprop="image"
   src="https://example.hypercontract.org/assets/product-01c31228-
   d5ba-44b3-84b1-7cccc7b9f707.jpg" alt="Product Image of Handcrafted
   Steel Mouse" />
12          </figure>
13 […]
14          <p><strong itemprop="price">Price: 66.07</strong></p>
15
16          <form
   action="https://example.hypercontract.org/shoppingCart/items"
   method="POST"
   rel="https://example.hypercontract.org/profile/addToShoppingCart"
   itemscope itemtype="https://example.hypercontract.org/profile/
   AdditionToShoppingCart">
17            <input type="hidden" name="product"
   value="https://example.hypercontract.org/products/01c31228-d5ba-
   44b3-84b1-7cccc7b9f707" itemprop="product" />
18
19            <div>
20              <label for="quantity">Quantity</label>
21              <input type="number" name="quantity" title="Quantity"
   value="1" itemprop="quantity" />
22              <div>
23                <button type="submit" title="Add to Shopping
   Cart">Add</button>
24              </div>
25            </div>
26          </form>
27
28          <p itemprop="productDescription">Deserunt accusamus vitae
   natus quia. Ullam […]</p>
29        </div>
30      </div>
31    </main>
32  </body>
33 </html>
```

Listing 39: HTML-Repräsentation eines Produkts mit Link-Relationstypen und Microdata-Auszeichnungen

Die aus *4.1.5.1 HTML* bekannte Produktansicht wurde in *Listing 39* um semantische Annotationen ergänzt (im Beispiel hervorgehoben). Das `form`-Element ist nun über das `rel`-Attribut mit dem Link-Relationstyp `addToShoppingCart` typisiert. Mit Hilfe von Microdata-Attributen wurde zudem das `main`-Element als Microdata-Item vom Typ `Product` definiert. Dessen Eigenschaften sind über `itemprop`-Attribute ausgezeichnet. Gleiches geschieht auf Ebene des Formulars, welches als `AdditionToShoppingCart` mit den entsprechenden Eigenschaften beschrieben wird.

Durch diese Ergänzungen besitzt das HTML-Dokument aus semantischer Sicht dieselbe fachliche Aussagekraft wie die entsprechende JSON-LD-Repräsentation.

4.3.4 Aushandlung des Contracts

Durch die Verlinkung des Profils via `Link`-Header und URI-basierte semantische Deskriptoren ist eine selbstbeschreibende Schnittstelle entstanden. Der Contract

zwischen Client und Server wird jedoch nicht auf Basis des Profils ausgehandelt, eine Einigung findet lediglich auf Ebene des Media Types statt (siehe *2.3.3 Contracts in REST*).

Um das Profil bei der Content Negotiation zu berücksichtigen wird daher der `Accept-Profile`-Header genutzt. Mit diesem kann der Client eine Repräsentation konform zum hypershop-Profil anfragen. Der Server antwortet mit dem entsprechenden `Content-Profile`-Header (siehe *Listing 40*). Erfolgt eine Anfrage mit einem Profil, das dem Server unbekannt ist, antwortet dieser mit dem Statuscode `406 Not Acceptable` (vgl. SVENSSON, VERBORGH 2019).

```
1  GET / HTTP/1.1
2  Host: example.hypercontract.org
3  Accept: application/ld+json
4  Accept-Profile: <https://example.hypercontract.org/profile>

5  HTTP/1.1 200 OK
6  Content-Type: application/ld+json; charset=utf-8
7  Content-Profile: <https://example.hypercontract.org/profile>
8  Link: <https://example.hypercontract.org/profile>; rel="profile"
```

Listing 40: Clientanfrage und Serverantwort (verkürzt) mit Aushandlung von Media Type und Profil via Content Negotiation

Auch wenn die *Content Negotiation by Profile* noch nicht standardisiert ist (siehe *2.3.3.4 Profile*), verdeutlicht das Beispiel im Kontext dieser Arbeit das Potenzial, das sich hinter dem Konzept verbirgt. Es ist vorstellbar, dass sie sich in Zukunft als wichtiges Werkzeug für die evolutionäre Weiterentwicklung von Web-APIs erweist.

Im Hinblick auf das Ziel dieser Arbeit ist die Aushandlung des Contracts über das Profil der letzte Baustein, um mit Hilfe von RDF und hypercontract die REST-API von hypershop für Mensch und Maschine selbstbeschreibend zu gestalten.

5 Evaluation

Mit dieser Arbeit wurde das Ziel verfolgt, eine Lösung für die Beschreibung von REST-konformen Webschnittstellen zu entwickeln, die mit den Konzepten und Architekturprinzipien des Webs vereinbar ist, den Anforderungen des REST-Architekturstils gerecht wird und in REST-konforme Web-APIs integriert werden kann. Die Beschreibung der API soll im Kontext der Nutzung verfügbar sein und sowohl für Menschen als auch für die maschinelle Verarbeitung aufbereitet werden. Sie umfasst dabei sowohl die Schnittstellenmechanik als auch die fachliche Semantik der Schnittstelle. Im Folgenden wird das Konzept von hypercontract anhand dieser Ziele bewertet.

5.1 Vereinbarkeit mit Webarchitektur

Evolvierbarkeit und *Interoperabilität* sind die zwei wesentlichen Architekturziele, nach denen alle Technologien des W3C entwickelt werden (vgl. BERNERS-LEE 1998a). Darüber hinaus ziehen sich die Designprinzipien *Einfachheit*, *Modularität*, *Toleranz* und *Rule of Least Power* durch die gesamte Architektur des Webs (vgl. BERNERS-LEE 1998b).

Mit RDF profitiert hypercontract von einem technischen Fundament, das vielen dieser Ziele und Prinzipien gerecht wird. Die konzeptionellen Gemeinsamkeiten mit REST und die Nähe zu anderen Webstandards bieten eine ganze Reihe von Integrationsmöglichkeiten und fördern die *Interoperabilität* der Lösung. Hinzu kommt, dass RDF nicht an ein konkretes Serialisierungsformat gebunden ist und die Dokumentation daher in verschiedenen Formaten ausgegeben werden kann. Durch JSON- oder XML-basierte Serialisierungen ist so beispielsweise auch Interoperabilität mit Clients gegeben, die RDF nicht unterstützen aber grundsätzlich in der Lage sind, JSON oder XML zu verarbeiten. Die Darstellung der Dokumentation als HTML-Seite zeigt zudem, dass die Struktur der Daten flexibel genug ist, um auch Ausgabeformate jenseits von RDF-basierten Serialisierungen zu bedienen.

Im Kontext der API-Beschreibung kann RDF in Kombination mit dem hypercontract-Vokabular im Vergleich zu Lösungen mit eigenen Datenformaten als *Language of Least Power* betrachtet werden. Einerseits lassen sich auch komplexere Schnittstellen beschreiben, andererseits kann das Profil trotzdem auch von Clients verarbeitet werden, die keine Kompatibilität zu hypercontract aufweisen. Im Sinne der *Einfachheit* wird zudem in vielen Bereichen auf etablierte Standards zurückgegriffen, anstatt das hypercontract-Vokabular unnötig aufzublähen (vgl. BERNERS-

LEE 2013). Auch dies verbessert die Zugänglichkeit für Clients ohne hypercontract-Vorwissen. Verbesserungspotenzial gibt es allerdings noch in Hinblick auf die Wiederverwendung existierender Vokabulare.

Die Offenheit von RDF vereinfacht zudem die Erweiterung um neue Sprachelemente oder applikationsspezifische Aspekte. Sollte ein Konsument diese nicht verstehen, ignoriert er sie im Sinne des *Toleranz*-Prinzips.

Nicht zuletzt fördert der Einsatz von RDF und damit einhergehend die Identifizierung von Konzepten über URIs auch die *Modularität* der Lösung. Vokabulare lassen sich beliebig aufteilen und wiederverwenden und durch verschiedene Systeme unabhängig voneinander ausliefern. Dadurch sind Szenarien denkbar, in denen domänenspezifische Vokabulare herangezogen werden, die nicht primär für die API-Dokumentation entwickelt wurden oder Vokabulare über mehrere Applikationen hinweg verwendet werden.

Auf der anderen Seite kann die enge Bindung an RDF auch von Nachteil sein. Die Technologie ist nach wie vor nicht im Mainstream der Webentwicklung angekommen, dementsprechend ist auch das Toolunterstützung nicht vergleichbar mit dem, was etablierte Lösungen wie beispielsweise OpenAPI anbieten. Und trotz der verschiedenen Serialisierungen, die RDF bietet, kann hypercontract seine Stärken nur dann ausspielen, wenn der Konsument in der Lage ist, RDF zu verarbeiten.

Darüber hinaus stellt hypercontract jedoch keine Anforderungen an konkrete Formate. Grundsätzlich ist denkbar, dass auch Formate abseits von XML und JSON beschrieben werden, sofern diese adressierbare Auszeichnungselemente aufweisen. Ebenso ist die Wahl der Technologie für die Schemadefinition freigestellt. Das erlaubt eine ausdrucksstarke und somit sichere Beschreibung des Nachrichtenformats. Auch dahinter steckt ein wichtiges Prinzip des Internets: Duplizierung vermeiden und auf etablierte Lösungen setzen (vgl. CARPENTER 1996).

Die Auszeichnung der hypershop-Website mit Hilfe des fachlichen Vokabulars über Microdata ist schließlich ein anschauliches Beispiel für die Interoperabilität, die durch den Einsatz etablierter, offener Webstandards entsteht.

5.2 Vereinbarkeit mit REST

Die Betrachtung existierender Lösungen zur Beschreibung von Webschnittstellen (siehe *3 Existierende Lösungen*) hat gezeigt, dass viele nicht dafür geeignet sind, REST-konforme APIs zu dokumentieren, ohne dabei die Prinzipien des Architekturstils zu untergraben. Häufig ist dies darauf zurückzuführen, dass Schnittstellen

ausgehend von ihren URLs beschrieben werden oder eng an bestimmte Nachrichtenformate gebunden sind und so zu einer unnötig engen Kopplung zwischen Server und Client führen. Auch die Beschreibung von Links ist in der Regel nicht oder nur über Umwege möglich.

Bei hypercontract steht daher der Hyperlink als wesentliches Merkmal von REST-APIs (vgl. FIELDING 2008) im Mittelpunkt der Dokumentation. Da Links das entscheidende Bindeglied für die Verknüpfung einzelner Ressourcen darstellen, wird die Schnittstelle ausgehend von Link-Relationstypen beschrieben. Dieser Ansatz wird auch der Tatsache gerecht, dass viele Elemente einer Schnittstelle über ein geeignetes Nachrichtenformat in der Repräsentation direkt abgebildet werden können. HTML ist hierfür ein gutes Beispiel: Die Möglichkeit Formulare in die Repräsentation der Webressource zu integrieren macht die Dokumentation von Datenschemas und HTTP-Methoden überflüssig. Die Beschreibung von Links hingegen ist obligatorisch.

Über die `expectedType`- und `returnedType`-Eigenschaften von `hyper:Operation` können in hypercontract Repräsentationen zudem unabhängig von Ressourcen beschrieben werden. Zusammen mit der Spezifikation von Nachrichteninhalten über formatspezifische Schemas ist so eine flexiblere Definition von Repräsentationen möglich, als dies beispielsweise bei Hydra der Fall ist (siehe *3.4 Hydra*).

Darüber hinaus weist die bereitgestellte API-Dokumentation dieselben Eigenschaften auf, die auch für REST-Schnittstellen gelten. Sie funktioniert nach dem Client-Server-Prinzip, ist zustandslos und kann problemlos über Caches zwischengespeichert werden. Die Bestandteile der API und deren Dokumentation werden als Ressourcen abgebildet, über URIs eindeutig identifiziert und sind über Hyperlinks miteinander verknüpft. So entsteht auch für die API-Dokumentation ein *Uniform Interface*, in welchem die Anbieter und Konsumenten sich auf ein von beiden Seiten unterstütztes Repräsentationsformat einigen. In der hier beschriebenen prototypischen Implementierung sind dies JSON-LD und HTML, denkbar wären in Zukunft aber auch noch weitere Formate.

5.3 Integration in REST-konforme Web-APIs

Der Integrationsgrad einer mit hypercontract verfassten API-Dokumentation hängt vom eingesetzten Nachrichtenformat der Schnittstelle ab. Grundsätzlich gilt für alle Systeme, dass eine minimale Integration über einen Verweis auf das Profil in den Kopfzeilen der Nachrichten erfolgen kann. Die Unabhängigkeit vom genutzten Media Type macht dies zu einer vielseitig anwendbaren Integrationsstrategie, die

darüber hinaus mit überschaubarem Aufwand auch bei bereits existierenden Systemen anwendbar ist.

Die Integration über die URIs von Link-Relationstypen ist ebenfalls unabhängig vom Nachrichtenformat und damit von der Implementierung der Applikation möglich.

Der höchste Integrationsgrad wird durch die Verwendung von JSON-LD oder einem anderen RDF-basierten Nachrichtenformat erreicht. Hierbei werden nicht nur die Link-Relationstypen über ihre URIs mit dem korrespondierenden Dokumentationsteil verknüpft, sondern auch die semantischen Deskriptoren der Nachricht, die ebenfalls über URIs eindeutig identifizierbar sind.

Allerdings muss einschränkend darauf hingewiesen werden, dass im Rahmen dieser Arbeit lediglich der Einsatz JSON-basierter Formate praktisch verprobt und XML-Formate nur theoretisch betrachtet wurden. Wie die Beschreibung von Formaten jenseits von JSON oder XML auf Basis von Profilen erfolgen kann, bedarf weiterer Forschung. Gleiches gilt für die Identifikation von Nachrichtenbestandteilen über URIs.

5.4 Verfügbarkeit im Kontext der Nutzung

Die Identifikation von semantischen Deskriptoren und Link-Relationstypen über URIs spielt eine wichtige Rolle, wenn es darum geht, den Zugriff auf die API-Beschreibung im Kontext der Nutzung zu erlauben. Erst die Möglichkeit, eine Beschreibung von Nachrichtenelementen über ihre URIs abzufragen, ermöglicht den direkten Zugriff auf relevante Dokumentation in der Situation, in der sie benötigt wird.

Ist dem Entwickler die Bedeutung eines Links oder eines Auszeichnungselements nicht ersichtlich, genügt ein Klick auf den URI des Bezeichners und der Browser präsentiert eine Beschreibung des Elements. Gleiches gilt aus maschineller Sicht: Folgt ein REST-Client einem Link innerhalb einer Nachricht, so kann er über den URI des Link-Relationstypen in Erfahrung bringen, mit welcher HTTP-Methode dies geschehen muss und könnte den Nachrichteninhalt gegen das Datenschema validieren, bevor er die Abfrage abschickt. Daran wird deutlich, dass die Nachrichten in der REST-Schnittstelle durch hypercontract tatsächlich selbstbeschreibend werden. Es sind keine zusätzlichen Informationen notwendig, die über die Nachricht hinausgehen, um diese zu verarbeiten.

Eine Integration rein über die Nachrichtenkopfzeilen ist daher im Sinne dieses Teilziels nicht ausreichend, da das Problem der entkoppelten Dokumentation im Wesentlichen weiter bestehen bleibt. Seine volle Aussagekraft entfaltet hypercontract erst im Zusammenspiel mit einem RDF-basierten Nachrichtenformat. Zumindest bei der Verwendung eines JSON-basierten Media Types besteht immer die Möglichkeit über das Referenzieren eines JSON-LD-Kontextes in den Kopfzeilen den Nachrichteninhalt nachträglich zu einem RDF-Dokument aufzuwerten.

5.5 Aufbereitung für Mensch und Maschine

Bei der Bereitstellung der Dokumentation für unterschiedliche Konsumenten setzt hypercontract auf die Flexibilität von HTTP. Über Content Negotiation entscheidet der Server, der die API-Beschreibung ausliefert, ob der angefragte Teil der Dokumentation maschinenlesbar als JSON-LD (oder potenziell auch in anderen RDF-Serialisierungen) zurückgeliefert wird oder als menschenlesbares HTML-Dokument.

Auf diese Weise erfolgt eine empfängergerechte Bereitstellung der Schnittstellenbeschreibung. Bei einem Zugriff über den Browser werden die Informationen so aufbereitet, dass sie für den Menschen lesbar und leicht verständlich sind. Über Links kann er zwischen verschiedenen Teilen der Dokumentation springen. Die Schema-Definitionen sind als Quellcode formatiert in die Dokumentation integriert. Erfolgt die Abfrage über einen REST-Client (ohne Angabe eines `Accept`-Headers oder mit `application/ld+json`), kann der Inhalt mit jedem JSON-kompatiblen Client ausgewertet werden. In zukünftigen Versionen sind weitere Formate denkbar, beispielsweise die Darstellung in Form von Grafiken oder als SPARQL-Endpunkt.

5.6 Beschreibung der Schnittstellenmechanik

Die Wahl von RDF als technisches Datenmodell für hypercontract bringt in Hinblick auf die Abbildung des semantischen Datenmodells sowohl Vor- als auch Nachteile mit sich. Seine Stärken spielt RDF aus, wenn es um die Definition von Konzepten und ihren Beziehungen untereinander geht – unabhängig von dem Format, in dem die Informationen letztendlich dargestellt werden (vgl. Berners-Lee 1998b). Diese Formatunabhängkeit hat allerdings zur Konsequenz, dass die Beschreibung von Datenschemas stets eine Übersetzung in die Datentypen des Zielformats zur Folge hat. Daher trennt hypercontract die Abbildung der fachlichen Konzepte von der syntaktischen Spezifikation über Datenschemas. Die Flexibilität hinsichtlich des Nachrichtenformats wird mit einer gewissen Redundanz bezahlt, die bei der

Beschreibung von Klassen und Eigenschaften in RDF und einem oder mehreren Datenschemas entsteht.

5.7 Beschreibung der fachlichen Semantik

Eine strukturierte, maschinenlesbare Abbildung der fachlichen Semantik stellt im Vergleich zur Schnittstellensemantik eine ganz andere Herausforderung dar und wird vermutlich nie mit vertretbarem Aufwand ohne menschenlesbare Prosa möglich sein (vgl. RICHARDSON, AMUNDSEN 2013, S. 138). Durch eine kleinteilige Beschreibung fachlicher Konzepte auf Basis von RDF ermöglicht hypercontract allerdings semantische Konzepte über die Grenzen der API-Dokumentation hinaus wiederverwendbar zu machen.

Klassen, Eigenschaften und Zustandsübergänge sind jeweils als Ressource identifizierbar und können daher mit Hilfe der `rdfs:comment`-Eigenschaft von RDF Schema fachlich beschrieben werden. Mit der Typisierung von Link-Relationstypen als `Operation` und der Abbildung von Vorbedingungen als `Preconditions` findet zudem eine teilstrukturierte Abbildung fachlicher Aspekte statt, die zukünftig um weitere Typen ausgebaut werden könnte. Denkbar wäre beispielsweise eine Definition möglicher Fehlerfälle oder die Beschreibung von Nachbedingungen (`Postconditions`).

Ein weiteres Alleinstellungsmerkmal von hypercontract gegenüber anderen etablierten Lösungen ist die Definition von Klasseneigenschaften als RDF-Propertys und somit eine Auflösung der festen Zuordnung von Eigenschaften zu einer einzigen Klasse. Dies erlaubt die Abbildung semantischer Äquivalenz von Eigenschaften über Klassengrenzen hinweg. Für die verarbeitenden Systeme eröffnet diese eindeutige Identifizierbarkeit von Eigenschaften interessante Optionen. So könnten aus Formulardaten automatisiert Nachrichten an den Server erzeugt werden, ohne dass die Gefahr von Namenskollisionen besteht.

6 Zusammenfassung und Ausblick

In der Einleitung wird beispielhaft das mühsame und oftmals frustrierende Erlebnis beschrieben, das Entwickler tagtäglich im Umgang mit webbasierten Schnittstellen erleben. Fachliche Prozesse werden auf die Mechanik von HTTP runtergebrochen, die zugehörigen Daten in ausdrucksschwache Datenformate übersetzt. Im Ergebnis entsteht eine API, die für den Konsumenten nur unter Zuhilfenahme einer entkoppelten Dokumentation verständlich wird. Zieht man einen Vergleich zum klassischen, vom Menschen genutzten Web, drängt sich die Frage auf, ob Web-APIs ähnlich intuitiv gestaltet werden können wie Websites. REST, als eine Abstraktion der Architektur des Webs, bietet eine Antwort auf diese Frage.

Im Kern steht dabei das Prinzip selbstbeschreibender Nachrichten: Alle Informationen, die zum Verständnis und zur Verarbeitung einer Nachricht erforderlich sind, müssen in ihr enthalten sein oder werden von ihr referenziert. Aufbauend auf diesem Grundsatz wurde im Rahmen dieser Arbeit ein Konzept entwickelt, das REST-konforme Webschnittstellen im Kontext ihrer Nutzung sowohl maschinenlesbar als auch für den Menschen intuitiv verständlich dokumentiert.

Auf Grundlage dieser Dokumentation können Anbieter und Konsumenten einer Schnittstelle ein gemeinsames Verständnis über deren fachliche Bedeutung und praktische Benutzung aufbauen. Dies ist die Voraussetzung für die Aushandlung des Contracts, den beide Seiten miteinander eingehen.

Die Vorstellung des REST-Architekturstils und seiner Merkmale hat jedoch deutlich gemacht, dass insbesondere das mit dem Uniform Interface verbundene Hypermedia-Prinzip übliche Herangehensweisen an die API-Beschreibung vor eine Herausforderung stellt. Statt einer vollständigen Beschreibung zur Designzeit, wie es beispielsweise die Webservices-Architektur rund um SOAP mit WSDL vorsieht, manifestiert sich der Contract in REST erst zur Laufzeit im Zusammenspiel von ausgehandelten Media Types und in den Nachrichten übermittelten Hypermedia-Elemente.

Bei näherer Betrachtung hat sich der Einsatz von Media Types für die Ziele dieser Arbeit jedoch als nicht praktikabel erwiesen, da existierende Formate für die Abbildung der fachlichen Semantik nicht ausreichen und die Spezifikation neuer Media Types mit hohem Aufwand verbunden ist. Daher wurde ergänzend auf die Möglichkeit eingegangen, Nachrichtenformate über die Angabe von Profilen um zusätzliche Semantik zu ergänzen.

Für die Zielsetzung dieser Arbeit war die Möglichkeit, applikationsspezifische Profile zu erstellen, von zentraler Bedeutung. Allerdings existiert kein standardisiertes, maschinenlesbares Format für deren Beschreibung. Mit dem Resource Description Framework wurde daher eine Beschreibungssprache herangezogen, die durch die konzeptionelle Nähe zum Web und aufgrund des flexiblen Datenmodells eine ideale Grundlage für die Definition von Profilen bildet.

Bestätigt wurde dies in der anschließenden Evaluierung existierender Lösungen. Sowohl Hydra als auch hRESTS setzen auf RDF und haben ihre Stärken insbesondere dieser technischen Basis zu verdanken. Mit ALPS wurde zudem ein Format für die Definition von Profilen untersucht, das unabhängig vom verwendeten Media Type ausgehend von Deskriptoren die Semantik des Nachrichtenformats dokumentiert. Der populäre OpenAPI-Standard zeugt dagegen von der Aussagekraft, formatspezifischer Datenschemas zur syntaktischen Beschreibung von Nachrichteninhalten.

Die Erkenntnisse dieser Evaluation bildeten zusammen mit der zuvor erfolgten theoretischen Betrachtung die Grundlage für die Vorstellung von hypercontract, einem Konzept zur Beschreibung REST-konformer Webschnittstellen für Mensch und Maschine.

In erster Linie handelt es sich bei hypercontract um ein RDF-Vokabular, mit welchem insbesondere die Schnittstellenmechanik, aber auch die fachliche Semantik von REST-APIs beschrieben werden kann. Darüber hinaus verbirgt sich dahinter aber auch der Ansatz, Webschnittstellen mit RDF-basierten Profilen maschinenlesbar zu beschreiben und auf dieser Basis HTML-basierte Dokumentation für den menschlichen Betrachter zu erzeugen, welche zur Laufzeit im Kontext der Nutzung bereitgestellt wird.

Illustriert wurde die Dokumentation einer Schnittstelle mit hypercontract anhand einer für diesen Zweck entwickelten, beispielhaften REST-API namens hypershop.

Ausgehend vom fachlichen Kontext der Anwendung wurde zunächst ein applikationsspezifisches Vokabular zusammengetragen, das die wesentlichen Konzepte und Aktivitäten der Schnittstelle eindeutig benennt. Dieses Vokabular, übersetzt in RDF, wurde im zweiten Schritt zusammen mit dem hypercontract-Vokabular genutzt, um die hypershop-API in Form eines RDF-basierten Profils zu beschreiben.

Die Schnittstellenmechanik konnte hierbei weitgehend maschinenlesbar abgebildet werden und umfasst das in RDF Schema und OWL abgebildete Datenmodell und die Protokollsemantik der Zustandsübergänge. Darüber hinaus wurde am

Beispiel von JSON Schema gezeigt, wie über die Einbindung zusätzlicher Schemasprachen eine formatabhängige Spezifikation der Nachrichteninhalte erfolgen kann. Die Abbildung der fachlichen Semantik der Schnittstelle in maschinenlesbarer erwies sich hingegen erwartbar schwieriger und erfolgte daher hauptsächlich in kleinteiliger, menschenverständlicher Form auf Ebene von Klassen, Eigenschaften und Link-Relationstypen. Am Beispiel von Vorbedingungen für den Aufruf von Zustandsübergängen konnte jedoch gezeigt werden, wie zumindest einzelne Aspekte der fachlichen Semantik auch strukturiert und somit für die maschinelle Auswertung geeignet beschrieben werden können.

Die Integration des Profils in die bestehende API erfolgte im dritten und letzten Schritt über die Verlinkung des Profils in den Kopfzeilen der Nachrichten sowie die Möglichkeit, Deskriptoren und Link-Relationstypen zu dereferenzieren. In Hinblick auf die Aushandlung des Contracts der Schnittstelle wurde mit der profilbasierten Content Negotiation zudem ein Ansatz vorgestellt, welcher sicherstellt, dass Client und Server ein gemeinsames Verständnis für die Nutzung der API besitzen.

Die abschließende Evaluierung des Konzepts ergab, dass hypercontract in den zentralen Punkten der eingangs formulierten Zielsetzung gerecht wird. In Hinblick auf die Vereinbarkeit mit der Architektur des Webs profitiert die vorgestellte Lösung insbesondere von der Offenheit und Flexibilität von RDF. Der Ansatz, APIs ausgehend von Link-Relationstypen zu beschreiben trägt wesentlich dazu bei, dass hypercontract dem REST-Architekturstil gerecht wird. Die Integration in bestehende APIs geschieht hauptsächlich über den konsequenten Einsatz von URIs, welcher auch ausschlaggebend für die Verfügbarkeit der Dokumentation im Kontext der Nutzung ist. Die strukturierte Spezifikation der Schnittstelle in Form eines RDF-Graphen ermöglicht diverse Optionen zur maschinellen Auswertung des Profils, einschließlich der Aufbereitung als HTML-Dokumentation für den menschlichen Betrachter. Einschränkend muss jedoch erneut betont werden, dass die Beschreibung der fachlichen Semantik nach wie vor die größte Herausforderung darstellt. Zwar bildet RDF auch hierfür eine solide Grundlage, die damit einhergehende Komplexität steht oft jedoch nicht im Verhältnis zum Ergebnis, weshalb das hier beschriebene Konzept sich größtenteils auf die natürlichsprachige Beschreibung der semantischen Aspekte beschränkt.

Im Ergebnis ist mit hypercontract eine Lösung zur Beschreibung REST-konformer Webschnittstellen entstanden, die nicht nur im Einklang mit der Architektur des Webs und den Prinzipien von REST steht, sondern darüber hinaus zur

maschinellen Auswertung geeignet ist und die explorative Herangehensweise von Entwicklern unterstützt.

Zur weiteren Evaluierung des Konzepts sollte die Beschreibung komplexer Schnittstellen mit hypercontract untersucht werden, um fehlende Konzepte zu identifizieren und die Skalierungseigenschaften des Ansatzes näher zu beleuchten. Auch beim Einsatz mit anderen Nachrichtenformaten gibt es noch Fragen, die im Rahmen dieser Arbeit nicht abschließend behandelt werden konnten.

In Hinblick auf die bislang wenig strukturierte Dokumentation der fachlichen Semantik der Schnittstelle lohnt sich ein Blick in Richtung des Domain-Driven Designs (vgl. EVANS 2011). Die Erarbeitung eines einheitlichen Vokabulars zur Beschreibung der Schnittstelle bietet interessante Anknüpfungspunkte zur *Ubiquitous Language* von DDD.

Ein Blick auf das umfangreiche Ökosystem rund um OpenAPI zeigt zudem, welche Möglichkeiten zur Entwicklerunterstützung sich durch die maschinenlesbare Beschreibung von Web-APIs ergeben. Werkzeuge zur Generierung von Client- und Server-Code sind auf Basis des hypercontract-Profils ebenso denkbar, wie die Erzeugung automatisierter Test-Suiten oder die Einrichtung von Discovery-Services zur dynamischen Vermittlung zwischen Konsumenten und Anbietern von Web-APIs. Nicht zuletzt könnte auch die Erstellung von RDF-basierten Profilen durch ein verbessertes Tooling vereinfacht werden.

Die vielfältigen Möglichkeiten, die sich mit hypercontract ergeben, zeigen nicht nur, dass die Lösung dem eigenen Anspruch an Interoperabilität und Evolvierbarkeit gerecht wird, sondern zeugen auch von den Stärken, die sich aus einer Architektur ergeben, die auf den Konzepten und Prinzipien des Webs basiert.

A Literaturverzeichnis

AAS, Patricia ; DIXIT, Shwetank ; EDEN, Terence ; LAWSON, Bruce ; MOON, Sangwhan ; WU, Xiaoqian ; O'HARA, Scott: *HTML 5.3 : W3C Working Draft*. URL https://www.w3.org/TR/html53/. – Aktualisierungsdatum: 2018-10-18 – Überprüfungsdatum 2019-10-05

ALLEN, Rob: *RESTful APIs and media-types*. URL https://akrabat.com/restful-apis-and-media-types/. – Aktualisierungsdatum: 2016-08-25 – Überprüfungsdatum 2019-04-05

AMUNDSEN, Mike ; RICHARDSON, Leonard ; FOSTER, Mark W.: *Application-Level Profile Semantics (ALPS)*. URL https://tools.ietf.org/html/draft-amundsen-richardson-foster-alps-01. – Aktualisierungsdatum: 2015-02-28 – Überprüfungsdatum 2019-11-21

AUSTIN, Daniel ; BARBIR, Abbie ; FERRIS, Christopher ; GARG, Sharad: *Web Services Architecture Requirements*. URL https://www.w3.org/TR/2004/NOTE-wsa-reqs-20040211/. – Aktualisierungsdatum: 2004-02-11 – Überprüfungsdatum 2019-09-07

BECKETT, David ; BERNERS-LEE, Tim ; PRUD'HOMMEAUX, Eric ; CAROTHERS, Gavin: *RDF 1.1 Turtle : Terse RDF Triple Language*. URL https://www.w3.org/TR/turtle/. – Aktualisierungsdatum: 2014-02-25 – Überprüfungsdatum 2019-11-21

BELSHE, Mike ; PEON, Roberto ; THOMSON, Martin: *Hypertext Transfer Protocol Version 2 (HTTP/2)*. URL https://tools.ietf.org/html/rfc7540 – Überprüfungsdatum 2019-05-29

BERNERS-LEE, Tim ; CONNOLLY, Dan: *Hypertext Markup Language 2.0*. URL https://www.w3.org/MarkUp/html-spec/html-spec_toc.html. – Aktualisierungsdatum: 1995-09-22 – Überprüfungsdatum 2019-10-05

BERNERS-LEE, Tim ; CONNOLLY, Dan: *Notation3 (N3) : A readable RDF syntax*. URL https://www.w3.org/TeamSubmission/n3/. – Aktualisierungsdatum: 2011-03-28 – Überprüfungsdatum 2019-11-21

BERNERS-LEE, Tim: *Content Negotiation of Content-type*. URL https://www.w3.org/DesignIssues/Conneg. – Aktualisierungsdatum: 2009-05-12 – Überprüfungsdatum 2019-10-28

BERNERS-LEE, Tim: *Evolvability*. URL https://www.w3.org/DesignIssues/Evolution.html. – Aktualisierungsdatum: 2009a-08-27 – Überprüfungsdatum 2019-04-09

BERNERS-LEE, Tim ; FIELDING, Roy T. ; MASINTER, Larry: *Uniform Resource Identifier (URI) : Generic Syntax*. URL https://tools.ietf.org/html/rfc3986 – Überprüfungsdatum 2019-05-31

BERNERS-LEE, Tim ; HENDLER, James ; LASSILA, Ora: *The Semantic Web*. In: *Scientific American* 284 (2001), Nr. 5, S. 96–101

BERNERS-LEE, Tim: *Linked Data*. URL https://www.w3.org/DesignIssues/LinkedData.html. – Aktualisierungsdatum: 2009-06-18 – Überprüfungsdatum 2019-02-02

BERNERS-LEE, Tim ; MASINTER, Larry ; MCCAHILL, Mark: *Uniform Resource Locators (URL)*. URL https://tools.ietf.org/html/rfc1738 – Überprüfungsdatum 2019-05-31

BERNERS-LEE, Tim: *Modularity*. URL https://www.w3.org/DesignIssues/Modularity.html. – Aktualisierungsdatum: 2010-02-28 – Überprüfungsdatum 2019-04-09

BERNERS-LEE, Tim: *Principles of Design*. URL https://www.w3.org/DesignIssues/Principles.html. – Aktualisierungsdatum: 2013-03-04 – Überprüfungsdatum 2019-04-09

BERNERS-LEE, Tim: *Semantic Web on XML : Architecture*. URL https://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html. – Aktualisierungsdatum: 2000-12-06 – Überprüfungsdatum 2019-10-08

BERNERS-LEE, Tim: *Web Architecture from 50,000 feet*. URL https://www.w3.org/DesignIssues/Architecture.html. – Aktualisierungsdatum: 2009b-08-27 – Überprüfungsdatum 2019-04-09

BOOTH, David ; HAAS, Hugo ; MCCABE, Francis ; NEWCOMER, Eric ; CHAMPION, Michael ; FERRIS, Chris ; ORCHARD, David: *Web Services Architecture*. URL https://www.w3.org/TR/ws-arch/. – Aktualisierungsdatum: 2004-02-11 – Überprüfungsdatum 2019-07-27

BOX, Don ; EHNEBUSKE, David ; KAKIVAYA, Gopal ; LAYMAN, Andrew ; MENDELSOHN, Noah ; NIELSEN, Henrik Frystyk ; THATTLE, Satish ; WINER, Dave: *Simple Object Access Protocol (SOAP) 1.1*. URL https://www.w3.org/TR/2000/NOTE-SOAP-20000508/. – Aktualisierungsdatum: 2000-05-08 – Überprüfungsdatum 2019-11-19

BRATT, Steve: *Semantic Web : and Other Technologies to Watch*. URL https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/0130-sb-W3CTechSemWeb.pdf. – Aktualisierungsdatum: 2007-01-29 – Überprüfungsdatum 2019-10-19

BRICKLEY, Dan ; GUHA, R. V.: *RDF Schema 1.1*. URL https://www.w3.org/TR/rdf-schema/. – Aktualisierungsdatum: 2014-02-25 – Überprüfungsdatum 2019-11-21

BRICKLEY, Dan ; MILLER, Libby: *FOAF Vocabulary Specification 0.99*. URL http://xmlns.com/foaf/spec/. – Aktualisierungsdatum: 2014-01-14 – Überprüfungsdatum 2019-11-21

CAROTHERS, Gavin ; SEABORNE, Andy: *RDF 1.1 N-Triples : A line-based syntax for an RDF graph*. URL https://www.w3.org/TR/n-triples/. – Aktualisierungsdatum: 2014-02-25 – Überprüfungsdatum 2019-11-21

CARPENTER, Brian E.: *Architectural Principles of the Internet*. URL https://tools.ietf.org/html/rfc1958 – Überprüfungsdatum 2019-04-09

CYGANIAK, Richard ; WOOD, David ; LANTHALER, Markus: *RDF 1.1 Concepts and Abstract Syntax*. URL https://www.w3.org/TR/rdf11-concepts/. – Aktualisierungsdatum: 2014-02-25 – Überprüfungsdatum 2019-04-18

DAIGNEAU, Robert: *Service Design Patterns : Fundamental Design Solutions for SOAP/WSDL and RESTful Web services*. Upper Saddle River : Addison-Wesley, 2012

DCMI: *DCMI Metadata Terms*. URL https://www.dublincore.org/specifications/dublin-core/dcmi-terms/. – Aktualisierungsdatum: 2012-06-14 – Überprüfungsdatum 2019-11-21

DÜRST, Martin ; SUIGNARD, Michel: *Internationalized Resource Identifiers (IRIs)*. URL https://tools.ietf.org/html/rfc3987 – Überprüfungsdatum 2019-05-29

ERL, Thomas: *Service-Oriented Architecture : Concepts, Technology, and Design*. 6. Aufl. Upper Saddle River : Prentice-Hall, 2006

EVANS, Eric: *Domain-driven design : Tackling Complexity in the Heart of Software*. Boston : Addison-Wesley, 2011

FARRELL, Joel ; LAUSEN, Holger: *Semantic Annotations for WSDL and XML Schema*. URL https://www.w3.org/TR/sawsdl/. – Aktualisierungsdatum: 2007-08-28 – Überprüfungsdatum 2019-11-21

FENSEL, Dieter ; FISCHER, Florian ; KOPECKÝ, Jacek ; KRUMMENACHER, Reto ; LAMBERT, Dave ; VITVAR, Tomas: *WSMO-Lite : Lightweight Semantic Descriptions for Services on the Web*. URL https://www.w3.org/Submission/WSMO-Lite/. – Aktualisierungsdatum: 2010-08-23 – Überprüfungsdatum 2019-11-21

FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*. Irvine, University of California. Dissertation. 2000. URL https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf – Überprüfungsdatum 2019-02-05

FIELDING, Roy T. ; RESCHKE, Julian F.: *Hypertext Transfer Protocol (HTTP/1.1) : Semantics and Content*. URL https://tools.ietf.org/html/rfc7231 – Überprüfungsdatum 2019-05-02

FIELDING, Roy T.: *REST APIs must be hypertext-driven*. URL http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven#comment-724. – Aktualisierungsdatum: 2008-10-20 – Überprüfungsdatum 2019-04-05

FREED, Ned ; KLENSIN, John C. ; HANSEN, Tony: *Media Type Specifications and Registration Procedures*. URL https://tools.ietf.org/html/rfc6838 – Überprüfungsdatum 2019-04-28

FRYSTYK NIELSEN, Henrik: *Interoperability and Evolvability*. URL https://www.w3.org/Protocols/Design/Interevol.html. – Aktualisierungsdatum: 1999-05-03 – Überprüfungsdatum 2019-05-29

GANDON, Fabien ; SCHREIBER, Guus: *RDF 1.1 XML Syntax*. URL https://www.w3.org/TR/rdf-syntax-grammar/. – Aktualisierungsdatum: 2014-02-25 – Überprüfungsdatum 2019-11-21

GREGORIO, Joe ; FIELDING, Roy T. ; HADLEY, Marc ; NOTTINGHAM, Mark ; ORCHARD, David: *URI Template*. URL https://tools.ietf.org/html/rfc6570 – Überprüfungsdatum 2019-04-28

HADLEY, Marc: *Web Application Description Language*. URL https://www.w3.org/Submission/wadl/. – Aktualisierungsdatum: 2009-08-31 – Überprüfungsdatum 2019-11-21

HENRY, Andrews ; AUSTIN, Wright: *JSON Hyper-Schema : A Vocabulary for Hypermedia Annotation of JSON*. Internet-Draft. URL https://tools.ietf.org/html/draft-handrews-json-schema-hyperschema-01. – Aktualisierungsdatum: 2018-01-19 – Überprüfungsdatum 2019-04-03

HERMAN, Ivan ; ADIDA, Ben ; SPORNY, Manu ; BIRBECK, Mark: *RDFa 1.1 Primer : Rich Structured Data Markup for Web Documents*. URL https://www.w3.org/TR/rdfa-primer/. – Aktualisierungsdatum: 2015-03-17 – Überprüfungsdatum 2019-11-21

JACOBS, Ian ; WALSH, Norman: *Architecture of the World Wide Web*. Volume One. URL https://www.w3.org/TR/webarch/. – Aktualisierungsdatum: 2004-12-15 – Überprüfungsdatum 2019-04-09

KELLY, Mike: *HAL - Hypertext Application Language : A lean hypermedia type*. URL http://stateless.co/hal_specification.html. – Aktualisierungsdatum: 2013-09-18 – Überprüfungsdatum 2019-11-21

KLUSCH, Matthias: Semantic Web Service Description. In: SCHUMACHER, Michael; SCHULDT, Helko; HELIN, Helkki (Hrsg.): *CASCOM : Intelligent Service Coordination in the Semantic Web*. Basel : Birkhäuser, 2008, 41-99

KLYNE, Graham ; CARROLL, Jeremy J.: *Resource Description Framework (RDF) : Concepts and Abstract Syntax*. URL https://www.w3.org/TR/rdf-concepts/. – Aktualisierungsdatum: 2004-02-10 – Überprüfungsdatum 2019-04-23

KOPECKÝ, Jacek ; GOMADAM, Karthik ; VITVAR, Tomas: hRESTS : An HTML Microformat for Describing RESTful Web Services. In: *Proceedings of the 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology.* Washington DC : IEEE, 2008 (3), S. 619–625

KOPECKÝ, Jacek ; VITVAR, Tomas ; PEDRINACI, Carlos ; MALESHKOVA, Maria: RESTful Services with Lightweight Machine-readable Descriptions and Semantic Annotations. In: WILDE, Erik; PAUTASSO, Cesare (Hrsg.): *REST: From Research to Practice.* New York : Springer Science+Business Media LLC, 2011, 473-506

LANE, Kin: *An API Definition As The Truth In The API Contract*. URL https://apievangelist.com/2014/07/15/an-api-definition-as-the-truth-in-the-api-contract/. – Aktualisierungsdatum: 2015-11-27 – Überprüfungsdatum 2019-05-15

LANTHALER, Markus: *Hydra Core Vocabulary : A Vocabulary for Hypermedia-Driven Web APIs*. URL https://www.hydra-cg.com/spec/latest/core/. – Aktualisierungsdatum: 2019-10-09 – Überprüfungsdatum 2019-11-21

LANTHALER, Markus: *Third Generation Web APIs : Bridging the Gap between REST and Linked Data*. Graz, TU Graz, Institute of Information Systems and Computer Media. Dissertation. 2014. URL http://www.markus-lanthaler.com/research/third-generation-web-apis-bridging-the-gap-between-rest-and-linked-data.pdf – Überprüfungsdatum 2019-02-02

LASSILA, Ora ; SWICK, Ralph: *Resource Description Framework (RDF) : Model and Syntax Specification*. URL https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/. – Aktualisierungsdatum: 1999-02-22 – Überprüfungsdatum 2019-11-19

MANOLA, Frank ; MILLER, Eric ; MCBRIDE, Brian: *RDF 1.1 Primer*. URL https://www.w3.org/TR/rdf11-primer/. – Aktualisierungsdatum: 2014-06-24 – Überprüfungsdatum 2019-04-18

MARTIN, David ; BURSTEIN, Mark ; HOBBS, Jerry ; LASSILA, Ora ; MCDERMOTT, Drew ; MCILRAITH, Sheila ; NARAYANAN, Srini ; PAOLUCCI, Massimo ; PARSIA, Bijan ; PAYNE, Terry ; SIRIN, Evren ; SRINIVASAN, Naveen ; SYCARA, Katia: *OWL-S : Semantic Markup for Web Services*. URL https://www.w3.org/Submission/OWL-S/. – Aktualisierungsdatum: 2004-11-22 – Überprüfungsdatum 2019-09-08

MARTIN, Robert C.: *The Interface Segregation Principle*. In: *C++ Report* 8 (1996)

MCCARRON, Shane ; ADAMCZYK, Paul ; BIRBECK, Mark ; KELLOGG, Gregg ; HERMAN, Ivan ; PEMBERTON, Steven: *HTML+RDFa 1.1 : Support for RDFa in HTML4 and HTML5*. URL https://www.w3.org/TR/html-rdfa/. – Aktualisierungsdatum: 2015-03-17 – Überprüfungsdatum 2019-11-21

MEALLING, Michael ; DANIEL, Ron: *URI Resolution Services Necessary for URN Resolution*. URL https://tools.ietf.org/html/rfc2483 – Überprüfungsdatum 2019-04-28

MILES, Alistair ; BECHHOFER, Sean: *SKOS Simple Knowledge Organization System Reference*. URL https://www.w3.org/TR/skos-reference/. – Aktualisierungsdatum: 2009-08-18 – Überprüfungsdatum 2019-04-29

MILLER, Darrel ; WHITLOCK, Jeremy ; GARDINER, Marsh ; RALPHSON, Mike ; RATOVSKY, Ron ; SARID, Uri: *OpenAPI Specification : Version 3.0.2*. URL http://spec.openapis.org/oas/v3.0.2. – Aktualisierungsdatum: 2018-10-08 – Überprüfungsdatum 2019-10-30

MOTIK, Boris ; PATEL-SCHNEIDER, Peter F. ; PARSIA, Bijan: *OWL 2 Web Ontology Language : Structural Specification and Functional-Style Syntax*. URL https://www.w3.org/TR/owl-syntax/. – Aktualisierungsdatum: 2012-12-11 – Überprüfungsdatum 2019-11-21

NEMEC, Zdenek: *API Blueprint Specification*. URL https://apiblueprint.org/documentation/specification.html. – Aktualisierungsdatum: 2019-09-12 – Überprüfungsdatum 2019-11-21

NEVILE, Chaals McCathie ; BRICKLEY, Dan: *HTML Microdata*. URL https://www.w3.org/TR/microdata/. – Aktualisierungsdatum: 2018-04-26 – Überprüfungsdatum 2019-05-05

NEWCOMER, Eric ; LOMOW, Greg: *Understanding SOA with Web Services*. Upper Saddle River : Addison-Wesley, 2005

NOTTINGHAM, Mark: *Web Linking*. URL https://tools.ietf.org/html/rfc5988 – Überprüfungsdatum 2019-05-02

NOTTINGHAM, Mark: *Web Linking*. URL https://tools.ietf.org/html/rfc8288 – Überprüfungsdatum 2019-04-09

PAGE, Kevin R. ; ROURE, David C. de ; MARTINEZ, Kirk: REST and Linked Data : a match made for domain driven development? In: PAUTASSO, Cesare; WILDE, Erik; ALARCON, Rosa (Hrsg.): *Proceedings of the Second International Workshop on RESTful Design*. New York : ACM Press, 2011, S. 22–25

PAUTASSO, Cesare ; ZIMMERMANN, Olaf ; LEYMANN, Frank: RESTful Web Services vs. "Big" Web Services : Making the Right Architectural Decision. In: HUAI, Jinpeng; CHEN, Robin; HON, Hsiao-Wuen; LIU, Yunhao; MA, Wei-Ying; TOMKINS, Andrew; ZHANG, Xiaodong (Hrsg.): *Proceedings of the 17th International World Wide Web Conference*. New York : ACM Press, 2008, S. 805–814

POLLERES, Axel ; KRENNWALLER, Thomas ; LOPES, Nuno ; KOPECKÝ, Jacek ; DECKER, Stefan: *XSPARQL Language Specification.* URL https://www.w3.org/Submission/xsparql-language-specification/. – Aktualisierungsdatum: 2009-01-20 – Überprüfungsdatum 2019-11-21

PRESCOD, Paul: *Second Generation Web Services*. URL https://www.xml.com/pub/a/ws/2002/02/06/rest.html. – Aktualisierungsdatum: 2002-02-06 – Überprüfungsdatum 2019-09-07

RAML: *RAML Version 1.0 : RESTful API Modeling Language*. URL https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md/. – Aktualisierungsdatum: 2019-06-06 – Überprüfungsdatum 2019-11-21

RICHARDSON, Leonard ; AMUNDSEN, Mike: *RESTful Web APIs*. Sebastopol : O'Reilly, 2013

ROBINSON, Ian: RESTful Domain Application Protocols. In: WILDE, Erik; PAUTASSO, Cesare (Hrsg.): *REST: From Research to Practice*. New York : Springer Science+Business Media LLC, 2011, S. 61–91

SAUERMANN, Leo ; CYGANIAK, Richard: *Cool URIs for the Semantic Web*. URL https://www.w3.org/TR/cooluris/. – Aktualisierungsdatum: 2008-03-31 – Überprüfungsdatum 2019-04-09

SCHEMA.ORG: *Schema.org*. URL https://schema.org/ – Überprüfungsdatum 2019-04-17

SEGARAN, Toby ; TAYLOR, Jamie ; EVANS, Colin: *Programming the Semantic Web*. Sebastopol : O'Reilly, 2009

SHADBOLT, Nigel ; BERNERS-LEE, Tim ; HALL, Wendy: *The Semantic Web Revisited*. In: *IEEE Intelligent Systems* 21 (2006), Nr. 3, S. 96–101. URL https://eprints.soton.ac.uk/262614/1/Semantic_Web_Revisted.pdf – Überprüfungsdatum 2019-10-08

SPICHALE, Kai: *API-Design : Praxishandbuch für Java- und Webservice-Entwickler*. 2., überarbeitete und erweiterte Auflage. Heidelberg : dpunkt.verlag, 2019

SPORNY, Manu ; KELLOGG, Gregg ; LANTHALER, Markus: *JSON-LD 1.0*. URL https://www.w3.org/TR/json-ld/. – Aktualisierungsdatum: 2014-01-16 – Überprüfungsdatum 2019-04-18

STATISTA: *Umsatz durch E-Commerce (B2C) in Deutschland in den Jahren 1999 bis 2018 sowie eine Prognose für 2019 (in Milliarden Euro)*. URL https://de.statista.com/statistik/daten/studie/3979/umfrage/e-commerce-umsatz-in-deutschland-seit-1999/. – Aktualisierungsdatum: 2019-05-20 – Überprüfungsdatum 2019-11-18

STURGEON, Phil: *A Response to REST is the new SOAP*. URL https://phil.tech/api/2017/12/18/rest-confusion-explained/. – Aktualisierungsdatum: 2017-12-18 – Überprüfungsdatum 2019-06-05

STURGEON, Phil: *OpenAPI and JSON Schema Divergence : Part 1*. URL https://apisyouwonthate.com/blog/openapi-and-json-schema-divergence-part-1. – Aktualisierungsdatum: 2018-05-09 – Überprüfungsdatum 2019-11-21

SVENSSON, Lars G. ; VERBORGH, Ruben: *Negotiating Profiles in HTTP*. URL https://profilenegotiation.github.io/I-D-Profile-Negotiation/I-D-Profile-Negotiation.html. – Aktualisierungsdatum: 2019-04-01 – Überprüfungsdatum 2019-05-02

SWAGGER: *Links*. URL https://swagger.io/docs/specification/links/ – Überprüfungsdatum 2019-11-21

TILKOV, Stefan ; EIGENBRODT, Martin ; SCHREIER, Silvia ; WOLF, Oliver: *REST und HTTP : Entwicklung und Integration nach dem Architekturstil des Web*. 3., aktualisierte und erweiterte Auflage. Heidelberg : dpunkt.verlag, 2015

TILKOV, Stefan: Representational State Transfer - REST. In: VERNON, Vaughn (Hrsg.): *Implementing domain-driven design*. 4. Aufl. Boston : Addison-Wesley, 2015, S. 133–138

TRASK, Matthew ; STURGEON, Phil: *OpenAPI.Tools*. URL https://openapi.tools/ – Überprüfungsdatum 2019-11-21

W3C: *Vocabularies*. URL https://www.w3.org/standards/semanticweb/ontology. – Aktualisierungsdatum: 2019-06-19 – Überprüfungsdatum 2019-11-21

W3C: *Web Services Activity*. URL https://www.w3.org/2002/ws/. – Aktualisierungsdatum: 2011-05-08 – Überprüfungsdatum 2019-11-19

WEBBER, Jim ; PARASTATIDIS, Savas ; ROBINSON, Ian: *REST in Practice : Hypermedia and Systems Architecture*. Sebastopol : O'Reilly, 2010

WHATWG: *HTML : Living Standard*. URL https://html.spec.whatwg.org/multipage/. – Aktualisierungsdatum: 2019-05-29 – Überprüfungsdatum 2019-05-29

WILDE, Erik ; DAVIS, Cornelia ; LIU, Yiming: *URI Template Descriptions*. URL https://tools.ietf.org/html/draft-wilde-template-desc-00. – Aktualisierungsdatum: 2012-12-07 – Überprüfungsdatum 2019-11-26

WILDE, Erik (Hrsg.); PAUTASSO, Cesare (Hrsg.): *REST: From Research to Practice*. New York : Springer Science+Business Media LLC, 2011

WILDE, Erik: *The 'profile' Link Relation Type*. URL https://tools.ietf.org/html/rfc6906 – Überprüfungsdatum 2019-04-09

WRIGHT, Austin ; ANDREWS, Henry: *JSON Schema : A Media Type for Describing JSON Documents*. URL https://tools.ietf.org/html/draft-handrews-json-schema-01. – Aktualisierungsdatum: 2018-03-19 – Überprüfungsdatum 2019-04-28

XML-RPC: *XML-RPC.Com : Simple cross-platform distributed computing, based on the standards of the Internet*. URL http://xmlrpc.scripting.com/. – Aktualisierungsdatum: 1999-06-14 – Überprüfungsdatum 2019-11-19

B Namensraumpräfixe

Präfix	Namensraum
`ex`	`http://example.org/`
`hyper`	`https://hypercontract.org/`
`owl`	`http://www.w3.org/2002/07/owl#`
`rdf`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`
`rdfs`	`http://www.w3.org/2000/01/rdf-schema#`
`shop`	`https://example.hypercontract.org/`
`xsd`	`http://www.w3.org/2001/XMLSchema#`

C Semantische Deskriptoren von hypershop

Klassen und Eigenschaften

Konzept	Deskriptor	Beschreibung	Typ
Addition to Shopping Cart	`AdditionToShoppingCart`	All information necessary to add a Product to the Shopping Cart.	Klasse
Address	`Address`	Postal addressed that can be used as Billing Address or Shipping Address when placing an Order. The Order will contain a copy of the address. Changes to an Address do not affect Orders that have been previously placed with that Address.	Klasse
API Root	`ApiRoot`	The entry point for the hypershop REST API.	Klasse
Billing Address	`BillingAddress`	The postal address to which the goods will be sent. This is a copy of an Address from the User Profile at the time when the Order was placed.	Klasse
New Order	`NewOrder`	All information necessary to place a new Order.	Klasse
Order	`Order`	An order that has been placed by the user. The Order may be in processing, already fulfilled or cancelled.	Klasse
Order History	`OrderHistory`	A list of all Orders that are currently being processed, already fulfilled or have been cancelled by the user.	Klasse
Order Cancellation	`OrderCancellation`	All information necessary to cancel an Order.	Klasse
Order Item	`OrderItem`	A Product and its quantity as it has been ordered. The Order Item contains a copy of all Product information at the time when the Order was placed.	Klasse
Payment	`Payment`	The payment details that are used to pay for the Order. This is a copy of a Payment Option from the User Profile at the time when the Order was placed.	Klasse
Payment Option	`PaymentOption`	Payment details that can be used for Payment when placing an Order. The Order will contain a copy of all payment details. Changes to a Payment Option do not affect Orders that have been previously placed with that Payment Option.	Klasse
Product	`Product`	A Product that can be ordered.	Klasse
Quantity Change	`QuantityChange`	All information necessary to change the quantity of a Shopping Cart Item.	Klasse
Search Query	`SearchQuery`	Search criteria used to search the catalog for Products.	Klasse

Search Results	`SearchResults`	A list of Products matching a Search Query.	Klasse
Shipping Address	`ShippingAddress`	The postal address to which the invoice will be sent. This is a copy of an Address from the User Profile at the time when the Order was placed.	Klasse
Shopping Cart	`ShoppingCart`	A list of Products that the user wants to order.	Klasse
Shopping Cart Item	`ShoppingCartItem`	A Product and the quantity the user wants to order. The Shopping Cart Item contains a copy of all Product information at the time when the Product was added to the Shopping Cart.	Klasse
Shopping Cart Items	`ShoppingCartItems`	The list of Shopping Cart Items that have been added to the Shopping Cart	Klasse
User Profile	`UserProfile`	Available Addresses and Payment Options of a user.	Klasse
account owner	`accountOwner`	The owner of a banking account.	Eigenschaft
addresses	`addresses`	A reference to all Addresses in a User Profile.	Eigenschaft
bic	`bic`	The BIC of a banking account.	Eigenschaft
billing address	`billingAddress`	A reference to the Billing Address of an Order. The name in the Billing Address does not have to be identical to the name of the account owner of the chosen Payment.	Eigenschaft
cancellation reason	`cancellationReason`	A free-text reason for why the Order was or is being cancelled.	Eigenschaft
city	`city`	The city of an Address.	Eigenschaft
country	`country`	The name of the country of an Address.	Eigenschaft
iban	`iban`	The IBAN of a banking account.	Eigenschaft
image	`image`	A picture of the Product.	Eigenschaft
items	`shoppingCartItem`	The Shopping Cart Items currently in the Shopping Cart.	Eigenschaft
name	`name`	The first and last name of the addressee of an Address.	Eigenschaft
order date	`orderDate`	The date when the Order was placed.	Eigenschaft
order items	`orderItems`	The Order Items of an Order.	Eigenschaft
order status	`orderStatus`	The status of the Order. Either "Processing", "Delivered" or "Cancelled".	Eigenschaft
orders	`orders`	A reference to Orders in the Order History.	Eigenschaft
payment	`payment`	A reference to the Payment of an Order. The name of the account owner does not have to be indentical to the name in the chosen Billing Address.	Eigenschaft
payment options	`paymentOptions`	A reference to all Payment Options in a User Profile.	Eigenschaft
price	`price`	The price per item.	Eigenschaft
product	`product`	A reference to a Product from the catalog.	Eigenschaft

products	products	A reference to Products from the catalog.	Eigenschaft
product description	productDescription	A description of the Product's features and qualities.	Eigenschaft
product name	productName	The name of the Product.	Eigenschaft
quantity	quantity	The number of items ordered or to be ordered.	Eigenschaft
query string	queryString	A free-text search term that is used to search the catalog for Products. It is matched against the product name and the product description.	Eigenschaft
shipping address	shippingAddress	A reference to the Shipping Address of an Order.	Eigenschaft
shopping cart items	shoppingCartItems	References to Shopping Cart Items in the Shopping Cart.	Eigenschaft
street	street	The street name and house number of an Address.	Eigenschaft
total price	totalPrice	The total price of all items in the Shopping Cart. It is calculated by summing up the item's prices multiplied with their respective quantities.	Eigenschaft
total results	totalResults	The total number of results matching the Search Query.	Eigenschaft
zip code	zipCode	The ZIP code of an Address. There are no restrictions on the ZIP code's format.	Eigenschaft

Definitions- und Wertebereiche von Eigenschaften

Deskriptor	Definitionsbereich	Wertebereich	Kardinalität
accountOwner	Payment PaymentOption	String	1
addresses	UserProfile	Address	1-n
bic	Payment PaymentOption	String	1
billingAddress	NewOrder Order	BillingAddress	1
cancellationReason	Order OrderCancellation	String	1
city	Address BillingAddress ShippingAddress	String	1
country	Address BillingAddress ShippingAddress	String	1
iban	Payment PaymentOption	String	1
image	Product	URL	1
items	ShoppingCart	ShoppingCartItem	0-n
name	Address BillingAddress ShippingAddress	String	1
orderDate	Order	Date	1
orderItems	Order	OrderItem	1-n
orders	OrderHistory	Order	0-n
orderStatus	Order	String ("Processing", "Delivered", "Cancelled")	1

payment	NewOrder Order	Payment	1
paymentOptions	UserProfile	PaymentOption	1-n
price	Product OrderItem ShoppingCartItem	Decimal (min: 0.01)	1
product	AdditionToShoppingCart ShoppingCartItem	Product	1
products	SearchResults	Product	0-n
productDescription	Product OrderItem ShoppingCartItem	String	1
productName	Product OrderItem ShoppingCartItem	String	1
quantity	AdditionToShoppingCart OrderItem QuantityChange ShoppingCartItem	Integer (min: 1)	1
queryString	SearchQuery	String	1
shippingAddress	NewOrder Order	ShippingAddress	1
shoppingCartItems	NewOrder	ShoppingCartItem	1-n
street	Address BillingAddress ShippingAddress	String	1
totalPrice	ShoppingCart	Decimal (min: 0.01)	1
totalResults	SearchResults	Integer (min: 0)	1
zipCode	Address BillingAddress ShippingAddress	String	1

D Zustandsübergänge von hypershop

Beschreibung der Link-Relationstypen

Link-Relationstyp	Beschreibung	Sicher?
addToShoppingCart	Adds a Product as a new Shopping Cart Item to the Shopping Cart or increases the quantity of an existing Shopping Cart Item when the Product has already been added to the Shopping Cart.	Nein
addresses	A reference to all Addresses in a User Profile.	Ja
cancel	Cancels the processing of an order. The status will be set to "Cancelled".	Nein
changeQuantity	Change the quantity of a Shopping Cart Item.	Nein
items	The Shopping Cart Items currently in the Shopping Cart.	Ja
orders	A reference to Orders in the Order History.	Ja
orderHistory	A link to the Order History.	Ja
paymentOptions	A reference to all Payment Options in a User Profile.	Ja
placeOrder	Places a new Order based on the provided information. The Shopping Cart will be emptied after the Order has been successfully created.	Nein
product	A reference to a Product from the catalog.	Ja
products	A reference to Products from the catalog.	Ja
remove	Removes a Shopping Cart Item from the Shopping Cart.	Nein
searchCatalog	Searches the catalog for Products matching a Search Query. When no Search Query is provided, all Products will be returned.	Ja
shoppingCart	A link to the Shopping Cart.	Ja
userProfile	A link to the UserProfile.	Ja

Definitions- und Wertebereiche von Link-Relationstypen

Link-Relationstyp	Definitionsbereich	Wertebereich	Umleitung auf
addToShoppingCart	Product	ShoppingCartItems	ShoppingCart
addresses	UserProfile	Address	
cancel	Order	Order	Order
changeQuantity	ShoppingCartItem	ShoppingCartItem	ShoppingCart
items	ShoppingCart	ShoppingCartItem	
orderHistory	ApiRoot	OrderHistory	
orders	OrderHistory	Order	
paymentOptions	UserProfile	PaymentOption	
placeOrder	ShoppingCart	Orders	Order
product	ShoppingCartItem	Product	
products	SearchResults	Product	
remove	ShoppingCartItem	ShoppingCartItem	ShoppingCart
searchCatalog	ApiRoot	SearchResults	SearchResults
shoppingCart	ApiRoot	ShoppingCart	
userProfile	ApiRoot	UserProfile	

Vorbedingungen für Zustandsübergänge

Vorbedingung	Beschreibung	Anwendbar auf Link-Rel.
`isOrderCancellable`	An Order can only be cancelled if the order status is still in "Processing".	`cancel`
`isShoppingCartOrderable`	An Order can only be placed if there is at least one Shopping Cart Item in the Shopping Cart.	`placeOrder`

E hypercontract Vokabular

Klassen

Name	Beschreibung
`EntryPoint`	Instances of this class serve as entry points to the API. The URI of the instance is also the URL of the resource.
`Operation`	Either an unsafe state transition or a safe state transition that expects certain query params or has a returned type different from the rdfs:range value of the propery describing the state transition.
`Precondition`	A precondition that needs to be met before the Operation can be performed.
`Schema`	A schema that describes the serialization of a class instance or property value for a specified target format.
`StateTransition`	A link relation type that describes a state transition. Unsafe state transitions must also be defined as an Operation.

Eigenschaften

Name	Beschreibung	Definitionsbereich	Wertebereich
`constraint`	The constraints defined for the Operation.	`Operation`	`Precondition`
`expectedBody`	The type of representation expected in the request body of the Operation request.	`Operation`	`rdfs:Class`
`expectedQueryParams`	The type that describes the query parameters of the Operation request.	`Operation`	`rdfs:Class`
`instanceSchema`	The Schemas that describe the serialization of a class instance.	`rdfs:Class`	`Schema`
`method`	The HTTP method of the Operation request (e.g. POST).	`Operation`	`xsd:string`
`returnedType`	The type of representation returned as a result of the Operation. When the Operation redirects to another resource, this type may differ from the rdfs:range value of the property describing the state transition.	`Operation`	`rdfs:Class`
`schemaType`	The media type of the schema definition (e.g. application/schema+json).	`Schema`	`xsd:string`
`targetType`	The media type of the serialization that the Schema describes (e.g. application/json).	`Schema`	`xsd:string`
`valueSchema`	The Schemas that describe the serialization of a property value.	`rdfs:Property`	`Schema`

Eidesstattliche Versicherung

Ich versichere, die vorliegende Arbeit selbständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangabe kenntlich gemacht.

Ort, Datum Unterschrift