

REST Hypermedia Modelling & Visualisation

Bachelorarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2013 / 14

Autoren: Nicolas Karrer, Marco Sonderegger
Betreuer: Prof. Dr. Olaf Zimmermann
Projektpartner: Actifsource GmbH, Baden-Dättwil
Experte: Dr. Gerald Reif, ipt Zug
Gegenleser: Prof. Dr. Andreas Steffen

ABSTRACT

Modellgetriebene Ansätze zur Entwicklung von REST-Schnittstellen existieren nach dem aktuellen Stand der Technik nur vereinzelt. REST bedeutet „Representational State Transfer“ und ist ein Programmierparadigma für Webanwendungen. Eine REST-Schnittstelle ist dementsprechend eine Webapplikations-Schnittstelle die den REST-Prinzipien genügt. Die vorliegende Bachelorarbeit setzt einen solchen, modellgetriebenen Ansatz für Schnittstellen nach dem REST Paradigma um. Verschiedene Gründe sprechen für eine Modellierung solcher REST-Schnittstellen. Zum einen ist es ein Bedürfnis den Aufwand für die Entwicklung solcher Schnittstellen zu reduzieren. Zum anderen ist eine Modellierung von grösseren REST-Schnittstellen einfacher und weniger fehleranfällig als bisherige, nicht modellgetriebene Ansätze. Mit Hilfe der actifsource-Umgebung, welche auf eclipse basiert, wurden ein Meta-Model und der dazugehörige Meta-Code entwickelt. Das Meta-Modell beschreibt, wie die REST-Schnittstellen grafisch modelliert werden. Der Meta-Code ist die generische Vorlage für den spezifischen Code der REST-Schnittstellen. Dieser individuelle Code wird generiert und weist eine Grundstruktur auf, welche für alle REST-Schnittstellen gleich ist. Die erreichte, einheitliche Struktur des spezifischen Codes führt zu den beabsichtigten Zeitersparnissen in Neu- und Weiterentwicklung, sowie Wartung. Die Lösung erfüllt zudem die folgenden Voraussetzungen: Die Lösung darf nicht von der Zieltechnologie abhängig sein und muss die Entwicklung von REST-Schnittstellen für Architekten und Entwickler vereinfachen. Die Lösung ist für Entwickler durch die grafische Modellierungs- und Code- Generierungsmöglichkeit eine grosse Unterstützung in der Entwicklung von REST-Schnittstellen.

EIGENSTÄNDIGKEITSERKLÄRUNG

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben,
- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum:

Rapperswil, 20.12.2013

Name, Unterschrift:



Nicolas Karrer



Marco Sonderegger

INHALTSVERZEICHNIS

1	Management Summary	5
1.1	Ausgangslage	5
1.2	Vorgehen	6
1.3	Ergebnisse	7
1.4	Ausblick.....	8
2	Einführung in Representational State Transfer (REST)	9
2.1	Was ist REST.....	9
2.2	Begriff: Ressource.....	9
2.3	Ressourcen-Repräsentation und Hypermedia	9
2.4	Begriff: URI	10
2.4.1	URI-Templates	10
2.4.2	URI-Tunneling	10
2.4.3	Nice-URI	11
2.5	HTTP-Idiome.....	11
2.5.1	Status Codes.....	11
2.5.2	Methoden	11
2.5.3	Header	12
2.6	Links	12
2.7	Richardson Maturity Model.....	12
2.8	State-Transfer und Zustandslosigkeit	13
2.9	Benutzer-Identifikation und Authentifizierung.....	13
2.9.1	Möglichkeiten für Benutzer – Authentifizierung.....	13
2.9.2	OAuth und OpenId	14
2.10	RESTful und HTML	14
2.10.1	Verfügbare Methoden.....	14
2.10.2	Hypermedia.....	15
3	Anforderungsanalyse	16
3.1	Beschreibung	16
3.1.1	Modell Level Übersicht	16
3.1.2	Produkt Perspektive	17
3.1.3	Produkt Funktion.....	17
3.1.4	Benutzer Charakteristik	17
3.1.5	Einschränkungen	17

3.2	Spezifische Anforderungen (Specific Requirements)	18
3.2.1	Qualitätsmerkmale	18
3.3	Use Cases	22
3.3.1	Use Case Diagramm	22
3.3.2	Aktoren & Stakeholder	23
3.3.3	Beschreibungen (Brief)	23
3.3.4	Beschreibungen (Fully Dressed)	24
3.3.5	Anpassungen bezüglich den Use Cases	28
4	Architektur der erarbeiteten Modellierungssoftware	29
4.1	Kontext	29
4.1.1	Einleitung	29
4.1.2	Übersicht	29
4.1.3	Actifsource-Ebene	30
4.1.4	Meta-Ebene	31
4.1.5	Konkrete Ebene	31
4.2	REST-Meta-Model	31
4.2.1	Einleitung	31
4.2.2	Übersicht	32
4.2.3	Komponenten	33
4.2.4	Zusätzliche Komponente	37
4.3	Templates	38
4.3.1	Beschreibung	38
4.3.2	Bestandteile und Funktionsweise eines actifsource-Templates	39
4.4	Genereller Meta-Code	40
4.4.1	Beschreibung	40
4.4.2	Meta-Code für Ressourcen	41
4.4.3	Meta-Code für Repräsentationen	42
4.4.4	Meta-Code für Links und URI	42
4.4.5	REST-Exception	44
4.5	ApacheHTTP spezifischer Meta-Code	44
4.5.1	Beschreibung	44
4.5.2	Server	45
4.5.3	Service	45
4.5.4	Request-Handler-Mapper	45
4.5.5	URI-Handler	46
4.6	JAX-RS spezifischer Meta-Code	46

4.6.1	Beschreibung	46
4.6.2	JaxRsUriHandler	47
4.6.3	Webxml	48
4.7	Ergänzende Templates	49
4.7.1	Beschreibung	49
4.7.2	SQLite-Template	49
4.7.3	JUnit-Template	49
5	Anwendung der Konzepte auf Applikationsbeispiele	50
5.1	Beispiel 1: Arztpraxis	50
5.1.1	Einleitung	50
5.1.2	Ausgangslage	50
5.1.3	Erkenntnisse	51
5.1.4	REST-Schnittstelle DoctorService	55
5.2	Beispiel 2: RestBucks	58
5.2.1	Einleitung	58
5.2.2	Ausgangslage	58
5.2.3	Erkenntnisse	60
5.2.4	REST-Schnittstelle RestBucks	61
5.3	Beispiel 3: E-Commerce	65
5.3.1	Einleitung	65
5.3.2	Ausgangslage	65
5.3.3	Erkenntnisse	66
5.3.4	REST-Schnittstelle E-Commerce	68
5.4	Beispiel 4: Projekt Verrechnung	74
5.4.1	Einleitung	74
5.4.2	Ausgangslage	74
5.4.3	Erkenntnisse	77
5.4.4	REST-Schnittstelle Accounting	78
5.5	Beispiel 5: Monopoly	85
5.5.1	Einleitung	85
5.5.2	Ausgangslage	85
5.5.3	Erkenntnisse	87
5.6	Beispiel 6: Visualisierungssoftware	88
5.6.1	Einleitung	88
5.6.2	Übersicht	88
5.6.3	3 Tiers: Variante 1	88

5.6.4	2 Tiers: Variante 2.....	89
5.6.5	Variante 1 vs. Variante 2.....	90
5.6.6	Spezifikation der DSL.....	90
5.6.7	REST-Schnittstelle RestVisualisation.....	94
6	Zusammenfassung und Ausblick.....	96
6.1	Zusammenfassung.....	96
6.2	Ausblick.....	96
Anhang A	I
Abbildungsverzeichnis	I
Literatur	II
Glossary	IV
Anhang B	VI
Benutzeranleitung		
Metriken		

1 MANAGEMENT SUMMARY

1.1 Ausgangslage

Netzwerk-Schnittstellen auf der Applikationsschicht sollen auf dem „Representational State Transfer (REST)“ Paradigma basieren. Dieses ist ein fundamentaler Programmierstil zur Integration von Webanwendungen. Die Idee von REST ist, dass eine URL exakt einen Seiteninhalt als Ergebnis einer serverseitigen Aktion darstellt. Die Schnittstellen müssen die nachfolgenden REST-Prinzipien erfüllen:

- Die Netzwerk-Schnittstellen stellen Ressourcen im Netzwerk zur Verfügung und sorgen dafür, dass diese angesprochen werden können.
- Ressourcen unterstützen unterschiedliche Repräsentationen.
- Die Kommunikation zwischen Client und Server ist zustandslos.
- Sie ermöglichen mittels HTTP-Methoden den Zugriff auf diese Ressourcen.
- Die REST-Schnittstellen veranlassen, dass mit Hyperlinks die Aktionen, welche ein Client ausführen darf, diesem bekannt sind.

Der Client ist demnach vom Server entkoppelt. Dies erlaubt die separate Implementation der jeweiligen Business-Logik und dadurch eine optimale Trennung der Zuständigkeiten. Mit anderen Worten, jede Komponente der Architektur ist nur für eine einzige Aufgabe verantwortlich. Dieses Designprinzip wird im Englischen „Separation of Concerns“ genannt. Das Ziel der Bachelorarbeit ist es, solche REST-Schnittstellen grafisch zu modellieren. Es ist wünschenswert den Entwicklungsaufwand des Programmcodes zu reduzieren. Um dies zu erreichen, soll schnittstellen-spezifischer Code erzeugt werden. Dessen Struktur wird mittels Meta-Code und dessen Inhalt durch die grafische Modellierung der REST-Schnittstelle bestimmt. Der Applikationsentwickler muss anschliessend lediglich noch die Business-Logik in den generierten Code der Schnittstellen hinzufügen. Die Lösung basiert auf der actifsource-Umgebung, welche als eclipse-Plugin vorhanden ist.

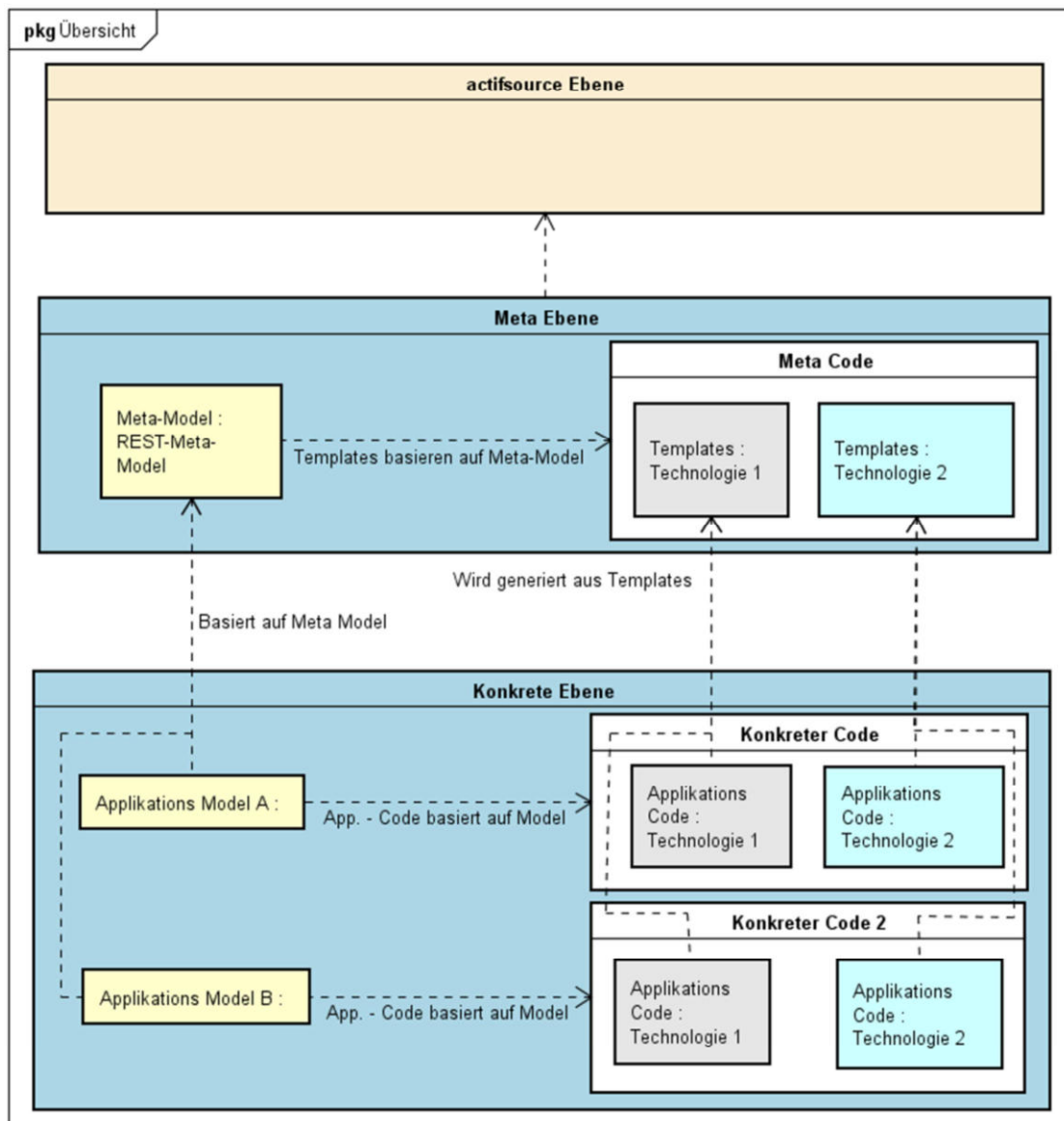


Abbildung 1: Übersicht der Architektur von REST Hypermedia Modelling & Visualisation

Die *Abbildung 1* zeigt die Architekturübersicht von REST Hypermedia Modelling & Visualisation auf. In der obersten Ebene befindet sich die actifsource-Umgebung. Die mittlere Ebene wird Meta Ebene genannt. Sie enthält das REST-Meta-Modell und den Meta-Code. Die unterste Ebene ist die konkrete Ebene. Sie beinhaltet die Business-Modelle und den spezifischen Code der effektiven Applikationen, auch REST-Schnittstellen genannt. Die Bedeutung der Ebenen wird im Bericht erklärt.

1.2 Vorgehen

Um das Ziel, REST-Schnittstellen grafisch zu modellieren und den Implementierungsaufwand des Entwicklers zu reduzieren, zu erreichen, wählten wir ein iteratives und agiles Vorgehensmodell ähnlich zu „Rational Unified Process (RUP)“. Wir teilten die Durchläufe in vier Schritte auf.

Im ersten Schritt analysierten wir die Problemstellung, sowie REST-Thematik und erstellten, als auch überarbeiteten die repräsentativen Beispiele. Im zweiten Schritt entwarfen und aktualisierten wir das Meta-Modell. Beim nächsten Schritt entwickelten und überarbeiteten

wir den Meta-Code für die Generierung des schnittstellen-spezifischen Codes. Im letzten Schritt der Iteration, implementierten und testeten wir das gewünschte Verhalten der REST-Schnittstellen. Zu dem verglichen wir die Applikationsbeispiele mit den Kriterien der Ausgangslage.

Dieser Vorgang wurde ausgeführt bis alle Kriterien für REST-Schnittstellen und somit das Ziel, REST-Schnittstellen grafisch zu modellieren und den Implementierungsaufwand des Entwicklers zu reduzieren, zu erreichen erfüllt waren.

1.3 Ergebnisse

Die beiden Hauptergebnisse der Arbeit sind ein REST-Meta-Modell und Meta-Code zur Generierung von spezifischem Code für REST-Schnittstellen. Mit Hilfe der Arbeitsergebnisse und des actifsource-Plugins für eclipse, ist es möglich REST-Schnittstellen grafisch zu modellieren und den dazugehörigen Code zu generieren.

Unserer Lösung fokussiert auf:

- Grafisches Modellieren der REST-Ressourcen und deren Verbindungen
- Konfigurieren der REST-Schnittstelle
- Generieren von Code

Für die Modellierung der REST-Schnittstellen führten wir einen neuen actifsource Diagramm-Typ „Service“ ein. Die Entwickler benutzen für die Modellierung der REST-Schnittstellen diesen Typ und damit unser Meta-Modell. Des Weiteren besteht die Möglichkeit erweiterte Diagramm-Typen und Meta-Modelle auf Basis des REST-Meta-Modelles zu erstellen. Diese gehen über unser Meta-Modell hinaus und decken individuelle Bedürfnisse der REST-Schnittstellen-Entwickler ab.

Die Konfiguration der REST-Schnittstelle bezieht sich auf die Definition der Medien-Typen, die Attribute und Adressierung der Ressourcen mittels flexibel konfigurierbarer URI.

Die Applikation generiert zwei verschiedene Arten von technologiespezifischem Code. Zum einen ist das ein eigenständiger Apache-HTTP-Server, zum anderen sind es Klassen, die mit dem REST-Framework JAX-RS und JERSEY von Oracle verwendet werden.

Zur Vervollständigung der Schnittstelle ist nur noch das manuelle Einbinden der eigentlichen Applikation nötig.

Der Programmcode aller REST-Schnittstellen ist einheitlich und übersichtlich. Das führt dazu, dass der Zeitaufwand für die Neu- und Weiterentwicklung, sowie Wartung der REST-Schnittstellen sinkt. Dies unterstreicht eine Analyse des Codes. Unsere Lösung generiert 80-90% des benötigten Applikationscodes der REST-Schnittstellen. Das bedeutet der Entwickler muss nur 10-20% des Codes selber schreiben.

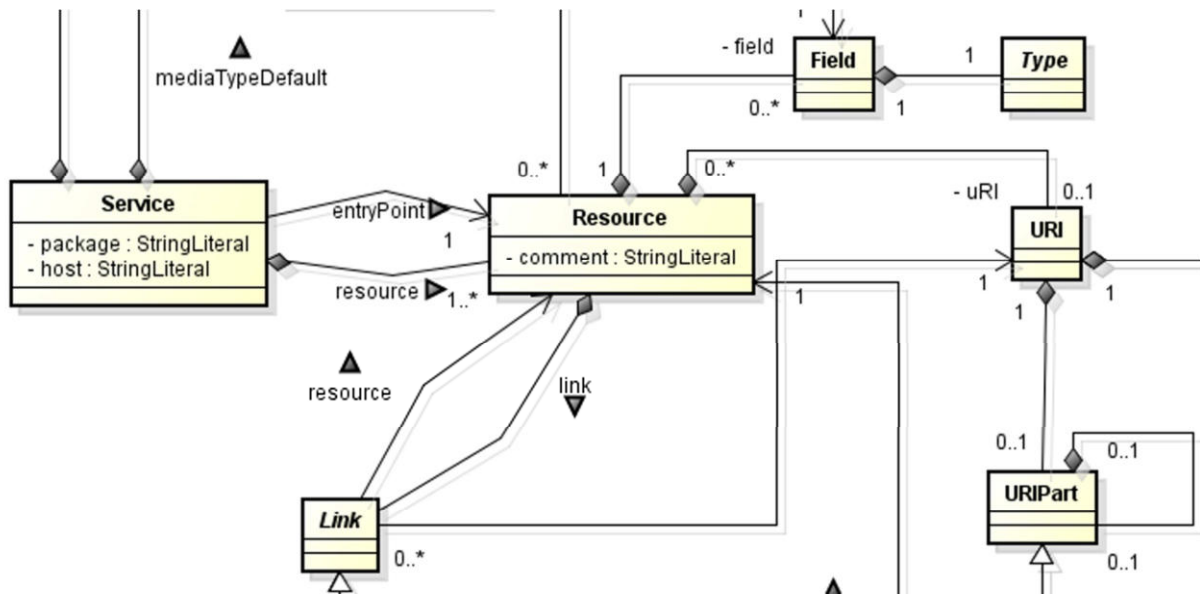


Abbildung 2: Ein Ausschnitt aus dem REST-Meta-Modell.

Die

Abbildung 2 stellt einen Ausschnitt aus dem erarbeiteten Meta-Modell von REST Hypermedia Modelling & Visualisation dar. Er zeigt die wichtigsten Elemente Service, Resource, Link sowie URI.

1.4 Ausblick

Erweiterungen der bestehenden Lösung sind auf verschiedene Arten denkbar. Einerseits ist es möglich, andere Ziel-Technologien zu verwenden. Das können andere Frameworks oder Programmiersprachen sein. In unserer Lösung verwenden wir die Java-Frameworks „Apache-HTTP-Server“ und „JAX-RS“.

Andererseits besteht die Möglichkeit das Meta-Modell um (Hilfs-) Funktionalität zu erweitern. Es liegen bereits Code-Vorlagen für SQLite und JUnit-Tests vor.

Unsere Lösung könnte in einer Folgearbeit von actifsource nach Eclipse Modeling Framework (EMF) oder Meta-Object Facility (MOF) der Object Management Group (OMG) in Enterprise Architect portiert werden. Möglich ist dies, da eclipse mit EMF, als auch Enterprise Architect mit MOF über Meta-Modellierungs-, sowie Code-Generierungsfunktionalität verfügt.

2 EINFÜHRUNG IN REPRESENTATIONAL STATE TRANSFER (REST)

2.1 Was ist REST

REST steht für „**RE**presentational **State Transfer**“ und ist eine Bezeichnung für ein Programmierparadigma. Der Ursprung von REST liegt in der Dissertation „*Architectural Styles and the Design of Network-based Software Architectures*“¹ von Roy Fielding aus dem Jahr 2000. Die Idee von REST ist, dass Ressourcen über eine bestimmte Bezeichnung, sogenannte URIs, im Netzwerk eindeutig identifizierbar sind. Zu REST gibt es keine Normen, trotzdem muss eine Schnittstelle gewisse Kriterien erfüllen um als RESTful bezeichnet werden zu können.

Als Protokoll für die Kommunikation von Client und Server wird das Hypertext Transfer Protocol (HTTP) verwendet.

Im Gegensatz zu anderen Technologien für Client-Server-Modelle, wie zum Beispiel *Remote Procedure Call* (RPC) findet die Interaktion des Clients mit dem Server nicht über Methoden statt. Bei REST wird der Status von Ressourcen ausgetauscht. Ein Vorteil davon ist, dass mit Hilfe von Links der Client vom Server entkoppelt wird. Das heisst der Client muss die möglichen Funktionen oder Methoden einer Ressource auf dem Server nicht kennen. Anstelle dessen wird er durch die vorhandenen Links in der Ressourcen-Repräsentation laufend auf die weiteren Möglichkeiten hingewiesen.

Die breiteste Verwendung von REST-Schnittstellen sind Webseiten und -applikationen. Der Fokus der Arbeit liegt nicht auf diesen zwei Arten sondern von Webservices. Es wird viel mehr auf die Verwendung von REST-Schnittstellen als Kommunikationsmöglichkeit in anderen verteilten Systemen beleuchtet.

2.2 Begriff: Ressource

Eine Ressource ist eine Informationsquelle. Die Definition einer Ressource ist nicht genauer spezifiziert. Es kann sich dabei sowohl um konkrete Konstrukte, wie zum Beispiel Bilder oder Dateien handeln, wie auch um abstrakte Konstrukte wie beispielsweise Operatoren². Ressourcen werden mittels einem *Uniform Resource Identifier* (URI) eindeutig identifiziert. Wobei folgendes gilt: Eine URI identifiziert genau eine Ressource, die Ressource kann aber durch über mehrere URIs erreichbar sein.

2.3 Ressourcen-Repräsentation und Hypermedia

Der Begriff Hypermedia setzt sich aus den Begriffen Hypertext und Multimedia zusammen. Hypertext Formate, wie zum Beispiel HTML und XML, bieten eine Möglichkeit Ressourcen zu repräsentieren. Eine andere Möglichkeit für solche Repräsentationen sind Medien-Technologien wie Grafiken, Videos und Ton-Formate.

Ressourcen-Repräsentationen bestimmen die Syntax oder das Format wie eine Ressource ausgeliefert wird. Eine Ressource kann dabei mehrere Repräsentationen haben. Zum Beispiel kann ein Bild sowohl als Bytecode, als auch über Meta-Informationen oder normalen

¹ [Field00]

² [RFC3986]

Text erreichbar sein. Identifiziert werden die Ressourcen über den so genannten Media Type.

2.4 Begriff: URI

URI bedeutet Unique Ressource Identifier und ist für die eindeutige Identifizierung von Ressourcen im Netzwerk zuständig.

Etwas bekannter ist der Begriff „Uniform Resource Location“ (URL). Wobei dieser ein Unter-Typ eines URI ist³.

Ein URI ist folgendermassen aufgebaut:

Schema://host[:port]/pfadteil/pfadteil2?parameter=wert#fragment.

Im Kontext dieser Arbeit, ist mit URI teilweise nur der Pfad-Teil des gesamten URIs gemeint.

2.4.1 URI-Templates

URI-Templates sind Vorlagen für URIs. Im Kontext dieser Arbeit gilt folgende Notation:

`/statischesElement/{dynamischesElement}`

Statische Elemente haben immer denselben Wert. Dynamische Elemente sind immer in geschweiften Klammern geschrieben und können verschiedene textbasierte Werte annehmen. Diese Notation wird auch vom Java-REST-Framework JAX-RS⁴ und dem PHP-Framework Silex⁵ für URI-Vorlagen verwendet.

2.4.2 URI-Tunneling

URI Tunneling bezeichnet den Vorgang, bei dem Ressourcen statt über eine eindeutige URI, über Parameter oder aus dem Inhalt des HTTP-Requests identifiziert werden.

Der Nachteil dieses Vorgangs ist, dass die HTTP-Paradigmen umgangen werden. Das kann sich negativ auf ein mögliches Caching auswirken. Als Beispiel dient ein folgender Beispiel-Link, wie er in einer beliebigen Web-Applikation vorkommen kann:

`index.php?action=createRessource&name=newRessource.`

Annahme: Der Server erstellt aus den Informationen aus den URI-Parametern action und name eine neue Ressource. Dieser Vorgang verstösst gegen das Prinzip der GET Methode, welches diese als rein lesender Zugriff definiert und keine Seiteneffekte haben darf.

Eine Weitere Möglichkeit ist, das Tunneling über POST. Dies ist ebenfalls ein Standard-Vorgang in einer Webaplikation. Es geschieht wenn Formulare abgesendet werden. Dabei bezieht der Server die Informationen nicht aus den URI – Angaben sondern aus dem Request-Payload.

³ [RESTIP]

⁴ [JAXRS]

⁵ [SILEX]

2 Einführung in Representational State Transfer (REST)

Dieses Vorgehen wird auch als REST – Anti Pattern bezeichnet⁶

2.4.3 Nice-URI

Nice-URIs bezeichnen für den Menschen einfach lesbare URIs. Zur Veranschaulichung können folgende Beispiele betrachtet werden:

- Ein Nice-URI: `/doctor/mjones/`
- Ein „Not-So-Nice-URI“: `/doctor?personalNumber=1354577890&action=read`

Folgende Punkte sind Design-Vorgaben für URIs⁷

- Nomen vor Verben bevorzugen
- URIs kurz halten
- Parameter sollen Positionsbezogen gesetzt werden
- URIs sollen nicht ändern

REST schreibt diese Praxis nicht vor⁸. Da die Modellierungsapplikation auf die Kommunikation zwischen Maschinen abzielt, sind diese Vorgaben auch nicht verbindlich. Für die Wartbarkeit der URI-Definitionen, sind sie jedoch auch in der Inter-Maschinen-Kommunikation zu bevorzugen. Eine Ausnahme bildet der vierte Punkt „URIs nicht ändern“. Durch die lose Kopplung die mit REST-Level 3 erreicht wird, verhindert dass das Ändern von URIs zu Problemen führt.

2.5 HTTP-Idiome

2.5.1 Status Codes

Status Codes sind zum Beispiel „404 not found“. Sie dienen als Hinweise ob ein Request erfolgreich war, oder nicht. Je nach Status-code sind sie spezifisch oder generell.

Einige wichtige und in der Arbeit verwendete Status-codes können der nachfolgenden Tabelle entnommen werden

Code	Bedeutung
200	Request war erfolgreich
404	Verlangte Ressource wurde nicht gefunden
405	Gewünschte HTTP-Methode ist nicht erlaubt
500	Auf dem Server wurde ein Fehler festgestellt
501	Gewünschte Funktionalität ist nicht implementiert

Quelle und eine Vollständige Liste der HTTP-Status-Codes sind im Standard rfc2616 zu finden⁹

2.5.2 Methoden

GET: Ermöglichen den lesenden Zugriff auf eine Ressource. Gemäss Spezifikation von HTTP, darf ein GET-Request keinen Inhalt (auch: *Content* oder *Payload*)¹⁰ haben. Zudem

⁶ [Paut1]

⁷ [Paut2]

⁸ [Paut2]

⁹ [RFC2616]

¹⁰ [RFC2616]

darf die Methode keinerlei Seiteneffekte haben. Das heisst eine GET-Anfrage erstellt, ändert oder löscht niemals eine Ressource. Eine Anfrage auf eine Ressource hat immer dasselbe Resultat, solange sich der interne Zustand einer Ressource nicht ändert.

PUT: Update einer Ressource. Nachteil dieser Methode ist, dass eine Ressource immer vollständig aktualisiert werden muss. Es sind keine partiellen Aktualisierungen von Ressourcen möglich.

POST: Erstellt eine Ressource. Darf aber auch für das partielle Aktualisieren einer Ressource verwendet werden. In gewissen Technologien wie HTML ist das sogar notwendig (Kapitel 2.10 *RESTful und HTML*).

DELETE: Löscht eine Ressource. Wie die GET-Methode darf diese keinerlei Inhalt haben.

2.5.3 Header

Es gibt verschiedene HTTP-Header. Einige können frei definiert werden, andere sind Bestandteile des HTTP-Protokolls.

Wichtige Header welche in der Arbeit verwendet werden können der nachfolgenden Tabelle entnommen werden:

Header	Bedeutung
Accept	Medien-Typen welche vom Client akzeptiert und interpretiert werden können.
Content-Type	Typ des im Protokoll übermittelten Inhaltes
Status-Code	Siehe Kapitel 2.5.1 <i>Status Codes</i>

2.6 Links

Links oder auch Hypermedia-Controls erlauben es, aus dem Inhalt einer Ressource auf eine andere zu schliessen. Dies sowohl lesend als auch schreibend. Ein Link benötigt Informationen zu der Ziel-Ressource wie zum Beispiel die HTTP-Methode und die Adresse um erfolgreich ausgeführt werden zu können. Zusätzlich können auch Informationen über die Darstellung hilfreich sein, vor allem wenn die Ressource in mehreren Repräsentationen zur Verfügung steht

2.7 Richardson Maturity Model

Das Richardson-Maturity-Model bezeichnet den Reifegrad einer REST-Schnittstelle. Dafür unterteilt es die REST-Schnittstellen in vier Stufen.

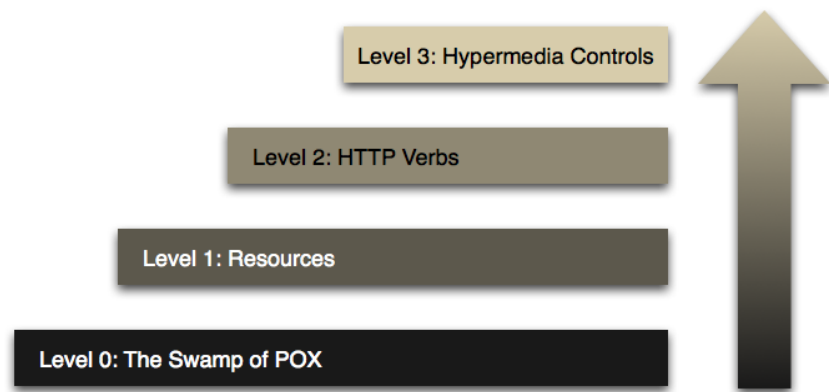


Abbildung 3: Graphik zum Richardson Maturity Model¹¹

Level 0: Zugriff auf eine Webapplikation oder Webservice über eine Ressource und eine HTTP-Methode.

Level 1: Zugriff auf Webapplikation über mehrere Ressourcen.

Level 2: Zugriff auf Webapplikation über mehrere Ressourcen und HTTP-Methoden

Level 3: Ressourcen verlinken auf andere Ressourcen. Hypermedia Unterstützung. Vorteil: Client muss nicht alle Ressourcen kennen sondern nur den Einstiegspunkt. Kontrollierte Workflows durch Verlinkung. Das Ziel ist eine lose Kopplung von Client und Server.

2.8 State-Transfer und Zustandslosigkeit

Die Kommunikation von Client und Server ist zustandslos. Das heisst der Server beantwortet Anfragen des Clients anhand von Informationen, welche er vom Client erhält ohne diese in einen Zeitabhängigen Kontext zu stellen¹².

Der „State-Transfer“ findet im Austausch von Ressourcen-Informationen statt. Der Server liefert dem Client den Status einer Ressource. Der Client wiederum darf diesen Status mittels http-Methoden manipulieren. Sofern entsprechende Funktionalität vom Server geboten wird.

Dieser Unterschied ist wichtig zu Verstehen bei der Verwendung oder Implementation von REST-Schnittstellen.

2.9 Benutzer-Identifikation und Authentifizierung

Die Identifikation und Authentifizierung von Clients durch REST-Schnittstellen ist mit der Prämisse der Zustandslosigkeit keine einfache Angelegenheit. Der Client muss bei jeder Anfrage Informationen zu seiner Identität mitgeben. Sofern solche Informationen vom Server verlangt werden.

2.9.1 Möglichkeiten für Benutzer – Authentifizierung

Es gibt Möglichkeiten Benutzer zu Authentifizieren, ohne dass die Prinzipien von REST umgangen werden oder zusätzlicher Aufwand nötig ist.

¹¹ [FowWeb]

¹² [Field00]

Eine Möglichkeit ist nach besuchen des Einstiegspunktes durch den Client, sämtliche weiterführenden Links benutzerabhängig zu gestalten. Dies kann mit Hilfe von Temporären Hash-Werten in der URI erreicht werden. Die Hash-Werte sind dabei der Identifikation einer Ressource der Schnittstellen und alle Anfragen werden über diese Ressource abgewickelt. Dabei darf diese Ressource nur nach erfolgreicher Identifikation des Clients, zum Beispiel durch Benutzername-Passwort Kombination, erstellt werden. Zusätzlich darf sie nur Temporär gültig sein um Missbrauch zu verhindern.

2.9.2 OAuth und OpenId

OAuth¹³ und OpenId¹⁴ sind Authentifizierungs-Services für Web-Applikationen. Der Vorteil dieser Lösungen ist, dass die Identifikation des Clients durch den Server an diese Services ausgegliedert werden kann. Zudem sind sie durch diverse Applikationen unterstützt was wiederum Vorteile für den Client hat.

Nachteil dieser Lösung ist, dass sie durch die externe Überprüfung die Antwortzeit des Servers verlängert.

2.10 RESTful und HTML

Die am häufigsten verwendeten Clients für Web-Applikationen sind die Web-Browser. Dieses Kapitel beschreibt einige Einschränkungen die beim Entwickeln von REST-Schnittstellen beachtet werden müssen, falls der Ziel-Client ein solcher Web-Browser sein soll.

2.10.1 Verfügbare Methoden

Seit jeher gibt es in HTML zwei Elemente welche für die Interaktivität von Webseiten zuständig sind. Der a-Tag (Anchor) und das form-Element (Formular).

Die unterstützten Methode eines a-Tags ist GET. Es können keine weiteren Methoden verwendet werden. Sie sind somit nur für lesende Zugriffe geeignet.

Der form-Tag wiederum unterstützt zwei Methoden. Das Attribut definiert für ein Formular entweder die GET oder die POST Methode. Dabei gilt es folgendes zu beachten: Erstens darf laut HTTP-Spezifikation GET keinen Inhalt haben. Was bei einem Formular aber immer, oder zumindest meistens der Fall ist. Zweitens ist eine vollständige Abbildung der Methoden nach REST in HTML nicht möglich, da PUT und DELETE nicht verwendet werden können.

Es gab Bestrebungen die Funktionalität in der HTML5 Spezifikation um PUT und DELETE zu erweitern. Im Endeffekt wurden sie aber nicht eingeführt, respektive wieder entfernt¹⁵.

Für die Entwickler gibt es Alternativen. Eine Möglichkeit ist es die Funktionalität mit Hilfe von JavaScript und AJAX abzubilden. Nachteil ist, dass das standardverhalten des Browsers damit umgangen wird. Eine Andere Möglichkeit zeigt das PHP-Framework Silex¹⁶. Dabei wird die Funktionalität mittels eines Formularfeldes abgebildet:

¹³ [OAUTH]

¹⁴ [OID]

¹⁵ [HTML5BUG]

¹⁶ [SILEX]

2 Einführung in Representational State Transfer (REST)

```
1. <form method="POST" />
2.   <input type="hidden" name="_method" id="_method" value="DELETE" />
3. </form>
```

Nachteil dieser Lösung ist, dass das eine programmiertechnische Lösung ist. Die Methode steht im Request-Payload und nicht als Header im HTTP-Protokoll.

2.10.2 Hypermedia

Wenn gewisse Ressourcen in HTML eingebunden werden, sendet der Browser eine neue Anfrage an den Webserver. Dies betrifft vor allem die Ressourcen-Typen: CSS-Dateien¹⁷, JavaScript-Dateien und Bilder. Mit dem HTTP-Accept-Header gibt der Browser an welchen Medien-Typ er für die Request-Antwort vom Server benötigt.

CSS-Dateien werden mit dem HTML-Tag `link` eingebunden:

```
<link rel="stylesheet" href="css/cssFile1.css" type="text/css" />
```

Der Browser sendet in der Anfrage den Accept-Header: `text/css, */*`. Der Typ `*/*` ist eine Wildcard und kann somit frei gewählt werden. Der Web-Server muss somit nur die Ressource CSS-Datei mit dem Medientyp `text/css` zur Verfügung stellen. Die Antwort muss mit dem http-Header `Content-Type=text/css` antworten. Ansonsten interpretiert der Browser die Style-Angaben nicht!

JavaScript-Dateien werden mit dem HTML-Tag `<script>` eingebunden:

```
<script type="text/javascript" src="javaScript/jsFile1.js"></script>
```

Der Browser sendet in der Anfrage nur den Accept-Header : `*/*`. Damit kann der Web-Server den gewünschten Content-Typen für die Antwort nicht eindeutig identifizieren. Einige Webserver senden dabei, wenn nichts anderes angegeben ist den richtigen Content-Typen. Für den Schnittstellen-Entwickler bedeutet das, dass er bei der Ressource nur einen Content-Typen erlauben kann. Wobei der Browser die Antwort auch mit dem Content-Type `text/html` richtig interpretiert und das JavaScript richtig ausführt.

¹⁷ Cascading Style Sheet (CSS)

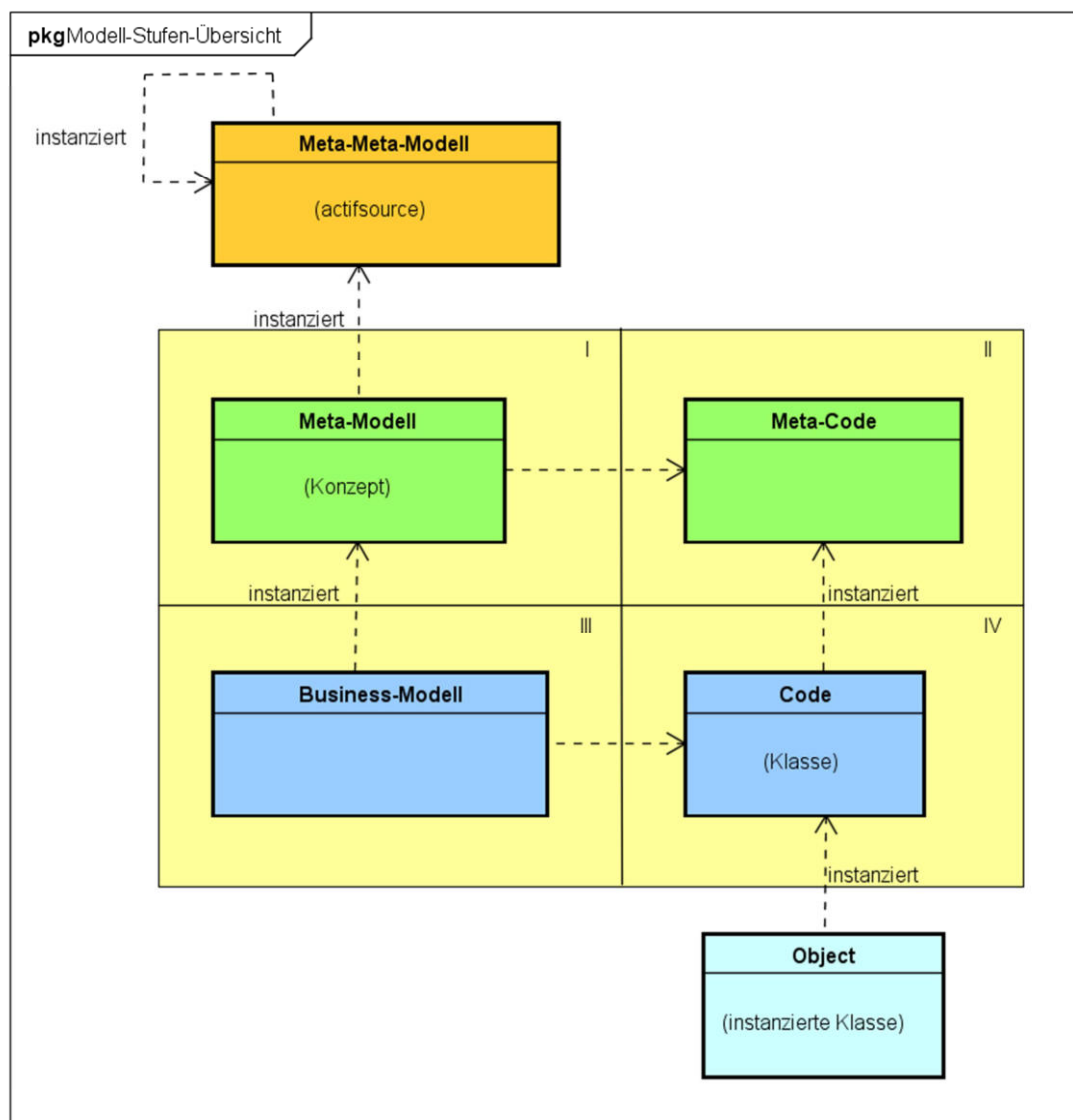
3 ANFORDERUNGSANALYSE

3.1 Beschreibung

Es soll ein Meta-Modell entwickelt werden. Dieses soll dem Benutzer ermöglichen RESTful (Representational State Transfer) Services zu erstellen. Er soll seine Beispielapplikationen modellieren können, welche auf dem übergeordneten Meta-Modell basieren. Zudem soll der Benutzer Client- / Server-Code generieren können.

Des Weiteren soll ein Tool entwickelt werden, welches von der actifsource Software Schnittstelle die DSL (Domain Specific Language) eines Models (XML-basiert) abfragt. Anschliessend werden die DSL als UML-Diagramme im Browser visualisiert.

3.1.1 Modell Level Übersicht



powered by Astah

Abbildung 4: Übersicht der verschiedenen Modell- und Code-Stufen

In der obigen Grafik ist grob der Zusammenhang der verschiedenen Modell - und Code - Stufen dargestellt. Die Begriffe in diesem Bild werden im nachfolgenden Dokument, wie auch in anderen Projekt Dokumenten, immer wieder vorkommen.

3.1.2 Produkt Perspektive

Das zu entwickelnde Meta-Model soll in Zukunft die Entwicklung von RESTful Services auf Maturitätsstufe 3 erleichtern.

Da es noch kein ähnliches Produkt auf dem Markt beziehungsweise als Freeware im Internet zu finden ist, kann dieses Meta-Model zur Generierung von Business-Models in Maturitätsstufe 3 grossen Anklang finden, sowohl bei Web-Entwicklern, als auch bei Kunden der actifsource GmbH. Da das Generieren von Client-Proxys und Server-Stubs einem Bedürfnis entspricht, könnte dieses Meta-Model, als Diagramm-Typ, in der actifsource Software eingebunden und / oder als eigenständiges eclipse-Plugin angeboten werden.

Das Visualisierungstool könnte man so erweitern, dass nicht nur actifsource DSL's im Browser visualisiert werden können, sondern auch eine beliebige andere DSL.

3.1.3 Produkt Funktion

Man kann mit Hilfe des zu entwickelnden Meta-Models einfach Services modellieren. Mittels des dazugehörigen Meta-Codes können dann auch Client Proxy's und Server-Stubs der entwickelten Services generiert werden. Dies erleichtert die Implementation der Businesslogik, weil das Gerüst bereits vorhanden ist.

Das Visualisierungstool fragt die DSL des gewünschten Modells ab. Diese XML-basierte DSL wird dann vom Tool geparkt. Anschliessend wird die DSL als UML-Diagramm auf einer Webseite dargestellt.

3.1.4 Benutzer Charakteristik

- Nutzer der actifsource-Software
- Web-Entwickler, welche RESTful Services mit Maturitätsstufe 3 erstellen wollen.

3.1.5 Einschränkungen

Business-Modelle werden nicht generiert. Diese müssen von Hand modelliert werden.

Die generierten Client-Proxys und Server-Stubs enthalten nur Request und Response Methoden, welche entweder nichts machen oder nur „null“-Werte, leere Strings oder 0-Repräsentationen von Zahlentypen zurückgeben. Der generierte Code muss nur kompilierbar sein. Es wird dabei keine Logik erzeugt. Diese muss von Hand geschrieben werden. Die Generierung von Proxys und Stubs soll sich ähnlich wie bei eclipse WTP¹⁸ verhalten.

Das Visualisierungstool kann nur actifsource DSL's parsen und ein UML-Diagramm davon im Browser darstellen. Das dargestellte UML-Diagramm ist nicht interaktiv.

¹⁸ [ECLIPSEWTP]

3.2 Spezifische Anforderungen (Specific Requirements)

3.2.1 Qualitätsmerkmale

3.2.1.1 Funktionale Anforderungen

- **Priorität A**
 - Meta-Modell ist komplett entwickelt und eingepflegt.
 - Meta-Code zum Meta-Modell ist implementiert
 - Das Meta-Modell ist als Diagramm-Typ in actifsource definiert.
 - Business-Modelle von den Beispielapplikationen können mit dem Diagramm-Typ erstellt werden.
 - Client-Proxys & Server-Stubs werden aus Code-Vorlagen generiert.
 - actifsource DSL (XML-basiert) kann mit dem Visualisierungstool geparst werden.
 - actifsource DSL (XML-basiert) kann mit dem Visualisierungstool als UML-Diagramm dargestellt werden.

- **Priorität B**
 - eclipse-Plugin mit dem Diagramm-Typ erstellt.
 - Services von den Beispielapplikationen können mit dem Plugin erstellt werden.
 - Client-Proxys & Server-Stubs können mit dem Plugin generiert werden.
 - Meta-Code der JUnit Tests zu den generierten Client-Proxys und Server-Stubs sind implementiert
 - JUnit Test zu den generierten Client-Proxys und Server-Stubs können generiert werden.

Die unter Priorität A definierten funktionalen Anforderungen müssen sicher entwickelt werden. Wenn noch Zeit vorhanden ist, sollen die Punkte von Priorität B auch erfüllt werden.

Mindestziel der BA ist, dass die Modellierung von Business-Modellen mittels actifsource-Plugin in eclipse funktioniert.

Wünschenswert wäre eine Weiterentwicklung des Meta-Models, so dass man nicht auf das actifsource Plugin angewiesen ist, um die Business-Modelle zu entwickeln.

3.2.1.2 Nicht-Funktionale Anforderungen

Benutzbarkeit

Das Erstellen einer REST-Schnittstelle soll logisch und intuitiv sein. Die Benutzung muss nicht speziell durch Tutorials beigebracht werden. Einzig die Bedienung des Built-in / Plugins bzw Diagramm-Typs kann mittels eines Tutorials erklärt werden. Bei Bedarf kann zu den, vom BA-Team gebrauchten, actifsource Tutorials verwiesen werden.

3 Anforderungsanalyse

Das Visualisierungstool soll einfach zu bedienen sein. Es soll somit intuitiv und selbstdokumentierend sein.

Die Test-Webseite des Visualisierungstools soll übersichtlich sein und ein schlichtes Design haben. D.h. es sollen nur ca. 2-7 Bedienelemente (Buttons, Links, etc.) vorhanden sein. Es soll auch nur wenige, ca. 2-5 Textpassagen haben. Die Anzahl Seiten der Test-Webseite soll nicht mehr als 2-5 Seiten umfassen. Das Design sollte deswegen schlicht gehalten werden, da es sich nur um eine Test-Webseite handelt. Diese hat nur zum Zweck die Funktionen des Visualisierungstools zu demonstrieren, was auch im Sinne des Benutzers ist. Durch eine übersichtliche, unkomplizierte und schlichte Gestaltung der Webseite und einfache Bedienung, wird dieses Ziel hervorragend erreicht. Die Benutzerfreundlichkeit ist zudem auch Erreicht

Überall, wo das Visualisierungstool Berechnungen anstellen muss, soll ein Arbeitsfortschritt oder eine Signalisation, das noch gearbeitet wird, angezeigt werden.

Die Rechtschreibung im Visualisierungstool, wie auch im Built-in / Plugin bzw. Diagramm-Typ soll grammatikalisch korrekt sein.

Zuverlässigkeit

Das, den Business-Modellen, zu Grunde liegende Meta-Modell soll nach bestem Wissen und Gewissen fehlerfrei sein.

Das Meta-Modell sollte eine hohe Abdeckung der Thematik mit repräsentativen Beispielen aus verschiedenen Quellen, aus Literatur und Industrie, gewährleisten. Es besteht jedoch kein Anspruch auf Vollständigkeit.

Der generierte Java Code soll syntaktisch korrekt sein.

Wenn der optionale Punkt „JUnit Test Cases generieren“ auch erfüllt wird, sollen die generierten JUnit Test Cases der Proxys und Stubs ebenfalls syntaktisch korrekt, das heißt sie müssen kompilieren, und ausführbar sein.

Wenn ein Built-in / Plugin aus dem Diagramm-Typ, welcher sonst nur innerhalb von actifsource gebraucht werden kann, erstellt wird, soll dieser einwandfrei funktionieren. Es soll keine vom Built-in / Plugin verschuldeten Abstürze von eclipse geben. Abstürze, welche von actifsource selbst herrühren, können vorkommen und haben nichts mit dem Built-in / Plugin zu tun.

Das Visualisierungstool soll fehlertolerant gegenüber Eingaben beim Parser und bei Aufrufen der Visualisierungsfunktion sein. Es sollen entsprechende Meldungen ausgegeben werden.

Effizienz

Man kann mit dem Built-in / Plugin bzw. Diagramm-Typ ein bis zwei einfache Beispielapplikationen, wie in der Dokumentation beschrieben, in vernünftiger Zeit erstellen. Für dies sollte ein mit REST-vertrauter und modellierungserfahrener Benutzer maximal zwischen 1 – 2 Stunden brauchen.

Beim Modellieren mit dem Built-in / Plugin bzw. der Diagramm-Typ soll die Reaktion schnell sein. D.h. sollen keine sichtbaren Verzögerungen beim Modellieren der REST-Schnittstellen auftreten.

Mit dem Built-in / Plugin bzw. Diagramm-Typ in actifsource soll man schneller eine lauffähige REST-Applikation in Maturitätsstufe 3 erstellen können, als wenn man diese komplett von Hand schreiben würde.

Das Parsen von einer actifsource DSL (DSL einer REST-Schnittstelle) soll zwischen 0.5 bis maximal 2 Minuten dauern.

Das Visualisierungstool soll zwischen 0.5 bis maximal 2 Minuten haben, um die geparsete DSL im Browser darzustellen.

Wartbarkeit

Der generierte Code soll mittels generierten oder von Hand geschriebenen JUnit Tests überprüft werden.

Das ganze Modellierungssystem soll mittels Systemtest getestet werden. Dabei soll ein Testprotokoll erstellt werden, welches die Testfälle abdeckt.

Bei beiden Testarten sollen alle definierten Tests bestanden werden.

Der Code des Server Teils des Visualisierungstools soll JUnit Tests überprüft werden.

Die Bedienung und die Darstellung der DSL sollen mittels Systemtest überprüft werden. Dazu wird ein Testprotokoll erstellt, welches die Testfälle abdeckt.

Bei beiden Testarten sollen alle definierten Tests bestanden werden.

Beim Visualisierungstool, als auch beim Modellierungsteil sollen die Namen im Code bzw. Komponenten möglichst kurz und prägnant sein.

Des Weiteren soll der Code gut dokumentiert sein und strukturiert sein, damit er relativ einfach analysier- und modifizierbar ist.

Übertragbarkeit

Beim Modellierungsteil wird Java SE, eclipse und das actifsource Plugin benötigt.

3 Anforderungsanalyse

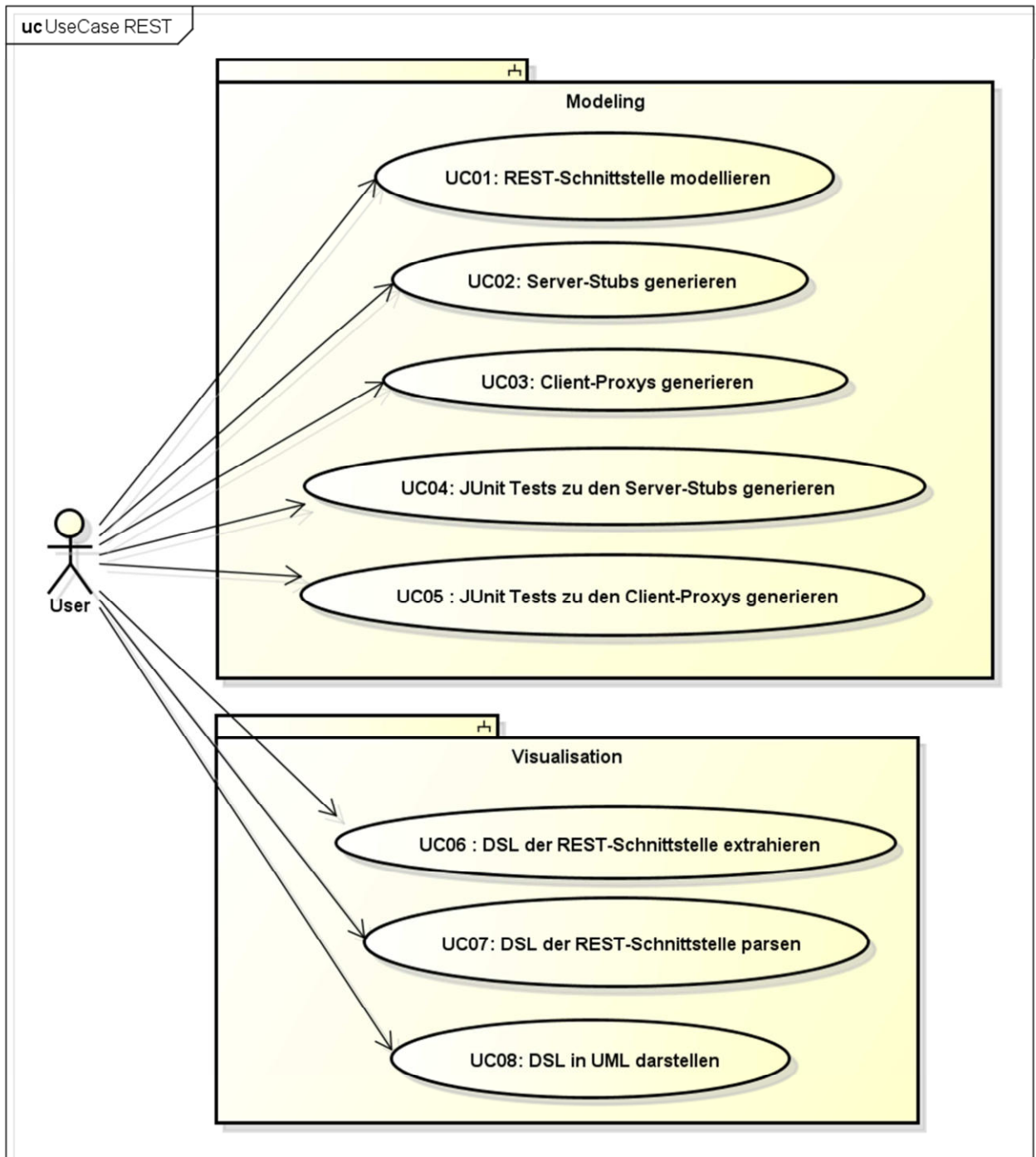
Das Visualisierungstool soll die Browser Google Chrome (Version 30.0), Mozilla Firefox (Version 24.0) und Microsoft Internet Explorer (Version 11.0) unterstützen.

Der Server benötigt für das Visualisierungstool den Apache Tomcat 7.

Das Visualisierungstool, als auch der Modellierungsteil sind auf Windows 7 lauffähig.

3.3 Use Cases

3.3.1 Use Case Diagramm



powered by Astah

Abbildung 5: Use Case Diagramm

3.3.2 Aktoren & Stakeholder

Aktor & Stakeholder	Beschreibung
REST-Modellierer / Web-Integrations-Programmierer	<p>Der Benutzer von dem Meta-Modell-Diagramm beziehungsweise Plugin und der Visualisierungsapplikation ist sowohl Aktor, als auch Stakeholder.</p> <p>Er kann REST-Schnittstellen modellieren, als auch Client-Proxys und Server-Stubs generieren. Eventuell kann der Benutzer die JUnit Tests zu den generierten Client-Proxys und Server-Stubs generieren (Tiefere Priorität, darum als eventuell bezeichnet). Dazu kommt, dass er DSL von einer REST-Schnittstelle exportieren kann. Der Benutzer kann eine DSL von einer REST-Schnittstelle im Visualisierungstool parsen und dann diese darstellen.</p>

3.3.3 Beschreibungen (Brief)

Nr.	Use Case	Beschreibung
UC01	REST-Schnittstelle modellieren	Der REST-Modellierer / Web-Integrations-Programmierer will eine REST-Schnittstelle bis hin zur Maturitätsstufe 3 ¹⁹ modellieren.
UC02	Server-Stubs generieren	Der REST-Modellierer / Web-Integrations-Programmierer will Server-Stubs von der REST-Schnittstelle generieren lassen.
UC03	Client-Proxys generieren	Der REST-Modellierer / Web-Integrations-Programmierer will Client-Proxys von der REST-Schnittstelle generieren lassen.
UC04	JUnit Tests zu den Server-Stubs generieren	(Eventuell) Der REST-Modellierer / Web-Integrations-Programmierer will die JUnit Tests zu den Server-Stubs generieren lassen.
UC05	JUnit Tests zu den Client-Proxys generieren	(Eventuell) Der REST-Modellierer / Web-Integrations-Programmierer will die JUnit Tests zu den Client-Proxys generieren lassen.
UC06	DSL der REST-Schnittstelle extrahieren	Der REST-Modellierer / Web-Integrations-Programmierer will die DSL einer REST-Schnittstelle extrahieren.
UC07	DSL der REST-Schnittstelle parsen	Der REST-Modellierer / Web-Integrations-Programmierer wählt eine actifsource DSL einer REST-Schnittstelle aus und lässt sie holen und parsen.
UC08	DSL in UML darstellen	Der REST-Modellierer / Web-Integrations-Programmierer will die geparste DSL in UML (Unified Modeling Language) darstellen.

¹⁹ Siehe Glossar

3.3.4 Beschreibungen (Fully Dressed)

3.3.4.1 UC01: REST-Schnittstelle modellieren²⁰

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der REST-Modellierer / Web-Integrations-Programmierer kann eine REST-Schnittstelle in der Maturitätsstufe 3 modellieren.
Stakeholder and Interests	Benutzer: Das Modellieren soll einfach und intuitiv sein. Alle benötigten Elemente für das Modellieren einer REST-Schnittstelle sollen vorhanden sein.
Preconditions	Das Meta-Modell ist vollständig.
Postconditions	Die REST-Schnittstelle wurde erfolgreich gespeichert.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der REST-Modellierer / Web-Integrations-Programmierer erstellt ein neues Diagramm. 2. Der REST-Modellierer / Web-Integrations-Programmierer modelliert die REST-Schnittstelle. 3. Der REST-Modellierer / Web-Integrations-Programmierer gibt einen Namen für die REST-Schnittstelle ein 4. Der REST-Modellierer / Web-Integrations-Programmierer speichert das Modell.
Extensions	-
Special Requirements	-
Frequency	Oft

3.3.4.2 UC02: Server-Stubs generieren

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der REST-Modellierer / Web-Integrations-Programmierer kann Server-Stubs von der REST-Schnittstelle generieren lassen.
Stakeholder and Interests	Benutzer: Generierter Code enthält keine Syntaxfehler. Code ist kompilierbar.
Preconditions	-
Postconditions	Die Server-Stubs wurden erfolgreich generiert.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der REST-Modellierer / Web-Integrations-Programmierer führt UC01 aus. 2. Die Server-Stubs werden vom System laufend generiert.
Extensions	-
Special Requirements	<ul style="list-style-type: none"> • Server-Stub haben Response beziehungsweise Request Methoden. • Die zu Generierenden Methoden oder Attribute werden vom Benutzer modelliert. • Diese Methoden sind entweder leer oder enthalten return-Werte, gemäss sprachspezifischen Typensystem. • Attribute werden gemäss dem sprachspezifischen Typensystem initialisiert.
Frequency	Oft

²⁰ Gemäss Meta-Model, siehe Kapitel: 4 Architektur der erarbeiteten Modellierungssoftware

3.3.4.3 UC03: Client-Proxys generieren

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der REST-Modellierer / Web-Integrations-Programmierer kann Client-Proxys von der REST-Schnittstelle generieren lassen.
Stakeholder and Interests	REST-Modellierer / Web-Integrations-Programmierer: Generierter Code enthält keine Syntaxfehler. Code ist kompilierbar.
Preconditions	-
Postconditions	Die Client-Proxys wurden erfolgreich generiert.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der REST-Modellierer / Web-Integrations-Programmierer führt UC01 aus. 2. Die Client-Proxys werden vom System laufend generiert.
Extensions	-
Special Requirements	<ul style="list-style-type: none"> • Client-Proxys haben Response beziehungsweise Request Methoden. • Die zu Generierenden Methoden oder Attribute werden vom Benutzer modelliert. • Diese Methoden sind entweder leer oder enthalten return-Werte, gemäss sprachspezifischen Typensystem. • Attribute werden gemäss dem sprachspezifischen Typensystem initialisiert.
Frequency	Oft

3.3.4.4 UC04: JUnit Tests zu den Server-Stubs generieren

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	(Eventuell) Der REST-Modellierer / Web-Integrations-Programmierer kann die JUnit Tests zu den Server-Stubs generieren lassen.
Stakeholder and Interests	REST-Modellierer / Web-Integrations-Programmierer: Generierter Code enthält keine Syntaxfehler. Code ist lauffähig. Die JUnit Test Cases umfassen alle Server-Stubs und deren Methoden.
Preconditions	-
Postconditions	Die JUnit-Test Cases für die Server-Stubs wurden erfolgreich generiert.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der REST-Modellierer / Web-Integrations-Programmierer führt UC01 aus. 2. Die JUnit-Test Cases für die Server-Stubs werden laufend generiert.
Extensions	-
Special Requirements	Es werden nur JUnit-Test Cases zu den vom REST-Modellierer / Web-Integrations-Programmierer modellierten und generierten Server-Stub Methoden generiert. Siehe UC02.
Frequency	Oft

3.3.4.5 UC05: JUnit Tests zu den Client-Proxys generieren

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	(Eventuell) Der REST-Modellierer / Web-Integrations-Programmierer will die JUnit Tests zu den Client-Proxys generieren lassen.
Stakeholder and Interests	REST-Modellierer / Web-Integrations-Programmierer: Generierter Code enthält keine Syntaxfehler. Code ist lauffähig. Die JUnit Test Cases umfassen alle Client-Proxys und deren Methoden.
Preconditions	-
Postconditions	Die JUnit-Test Cases für die Client-Proxys wurden erfolgreich generiert.
Main Success Scenario	<ol style="list-style-type: none"> 3. Der REST-Modellierer / Web-Integrations-Programmierer führt UC01 aus. 4. Die JUnit-Test Cases für die Client-Proxys werden laufend generiert.
Extensions	-
Special Requirements	Es werden nur JUnit-Test Cases zu den vom REST-Modellierer / Web-Integrations-Programmierer modellierten und generierten Client-Proxy Methoden generiert. Siehe UC03.
Frequency	Oft

3.3.4.6 UC06: DSL von der REST-Schnittstelle exportieren

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der REST-Modellierer / Web-Integrations-Programmierer will die DSL von einer REST-Schnittstelle exportieren.
Stakeholder and Interests	Benutzer: DSL's der REST-Schnittstellen sind einfach zu exportieren.
Preconditions	UC01 wurde ausgeführt.
Postconditions	Exportierte DSL der REST-Schnittstelle entspricht dem Model.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der REST-Modellierer / Web-Integrations-Programmierer wählt das zu exportierende REST-Schnittstellen-Diagramm aus. 2. Der REST-Modellierer / Web-Integrations-Programmierer wählt im Kontext-Menü von eclipse „Export“ aus. 3. Unter REST (oder actifsource?) wählt er „DSL Export (XML)“ aus.
Extensions	-
Special Requirements	-
Frequency	Oft

3.3.4.7 UC07: DSL von der REST-Schnittstelle parsen

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der Benutzer wählt eine actifsource DSL einer REST-Schnittstelle aus und lässt sie importieren und parsen.
Stakeholder and Interests	Benutzer: Das Auswählen der DSL Datei entspricht dem Standard. Das Analysieren der DSL funktioniert in angemessener Zeit. Es wird ein Arbeitsstatus angezeigt.
Preconditions	Die DSL Datei ist bereits vorhanden. <ul style="list-style-type: none"> • z.B. UC06 wurde ausgeführt.
Postconditions	Die DSL wurde erfolgreich geparkt.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der Benutzer wählt die DSL Datei aus. 2. Er drückt auf den Parse-Button.
Extensions	<p>1a: Der Benutzer wählt die REST-Schnittstelle aus.</p> <ul style="list-style-type: none"> • Der Benutzer muss nicht mehr das Parsen der DSL auslösen. Er kann zu UC08 weitergehen.
Special Requirements	-
Frequency	Oft

3.3.4.8 UC08: DSL in UML darstellen

Primary Actor	REST-Modellierer / Web-Integrations-Programmierer
Overview	Der Benutzer will die geparkte DSL in UML darstellen.
Stakeholder and Interests	Benutzer: Die DSL wird korrekt in UML dargestellt. Das UML-Diagramm ist übersichtlich und vollständig.
Preconditions	UC07 wurde ausgeführt.
Postconditions	Das UML-Diagramm wird korrekt im Browser dargestellt.
Main Success Scenario	<ol style="list-style-type: none"> 1. Der Benutzer wählt die in UC07 geparkte DSL aus. 2. Er lässt die DSL visualisieren
Extensions	1a: Der Benutzer wählt eine andere, früher geparkte DSL aus
Special Requirements	Alle bereits früher geparkte DSL bleiben zur Auswahl für das Visualisieren vorhanden, jedoch nur solange das Visualisierungstool nicht neu gestartet wird.
Frequency	Oft

3.3.5 Anpassungen bezüglich den Use Cases

Nr.	Use Case	Anpassungen
UC01	REST-Schnittstelle modellieren	<ul style="list-style-type: none"> Keine Anpassungen
UC02	Server-Stubs generieren	<ul style="list-style-type: none"> Keine Anpassungen
UC03	Client-Proxys generieren	<ul style="list-style-type: none">
UC04	JUnit Tests zu den Server-Stubs generieren	<ul style="list-style-type: none"> Keine Anpassungen
UC05	JUnit Tests zu den Client-Proxys generieren	<ul style="list-style-type: none">
UC06	DSL der REST-Schnittstelle extrahieren	<ul style="list-style-type: none"> Es gibt keine Möglichkeit die DSL einer REST-Schnittstelle zu exportieren. Es muss eine XML-Datei von Hand erstellt werden, welche die REST-Schnittstelle beschreibt. Siehe im Applikationsbeispiele Kapitel²¹ für eine genaue Beschreibung der Struktur und des Aufbaus einer solchen XML-Datei.
UC07	DSL der REST-Schnittstelle parsen	<ul style="list-style-type: none"> Keine Anpassungen
UC08	DSL in UML darstellen	<ul style="list-style-type: none"> DSL wird nicht als UML dargestellt. DSL wird im Browser als SVG dargestellt.

3.4 Nicht umgesetzte Anforderungen

Da die Entwicklung der Serverseite anspruchsvoller war als ursprünglich gedacht, konnten die Templates für die Client-Stubs nicht mehr implementiert werden.

Es wurde ebenfalls aus zeitlichen Gründen keine automatisierten Tests geschrieben werden. Das Testen der Applikation erfolgte von Hand. Dabei wurden bereits vorhandene REST-Clients oder ein Browser verwendet.

Die Unterstützung des Browser Internet-Explorer wird aus Mangel an Kompatibilität von der Visualisierungsapplikation nicht unterstützt.

²¹Kapitel 5 Anwendung der Konzepte auf Applikationsbeispiele

4 ARCHITEKTUR DER ERARBEITETEN MODELLIERUNGS SOFTWARE

4.1 Kontext

4.1.1 Einleitung

Dieses Kapitel beschreibt den Kontext der Modellierungssoftware.

4.1.2 Übersicht

Die Grafik zeigt eine Übersicht des Kontextes. Die einzelnen Bereiche und Komponenten werden in den nachfolgenden Unterkapiteln beschrieben. Zur Veranschaulichung des Kontextes dient *Abbildung 6*.

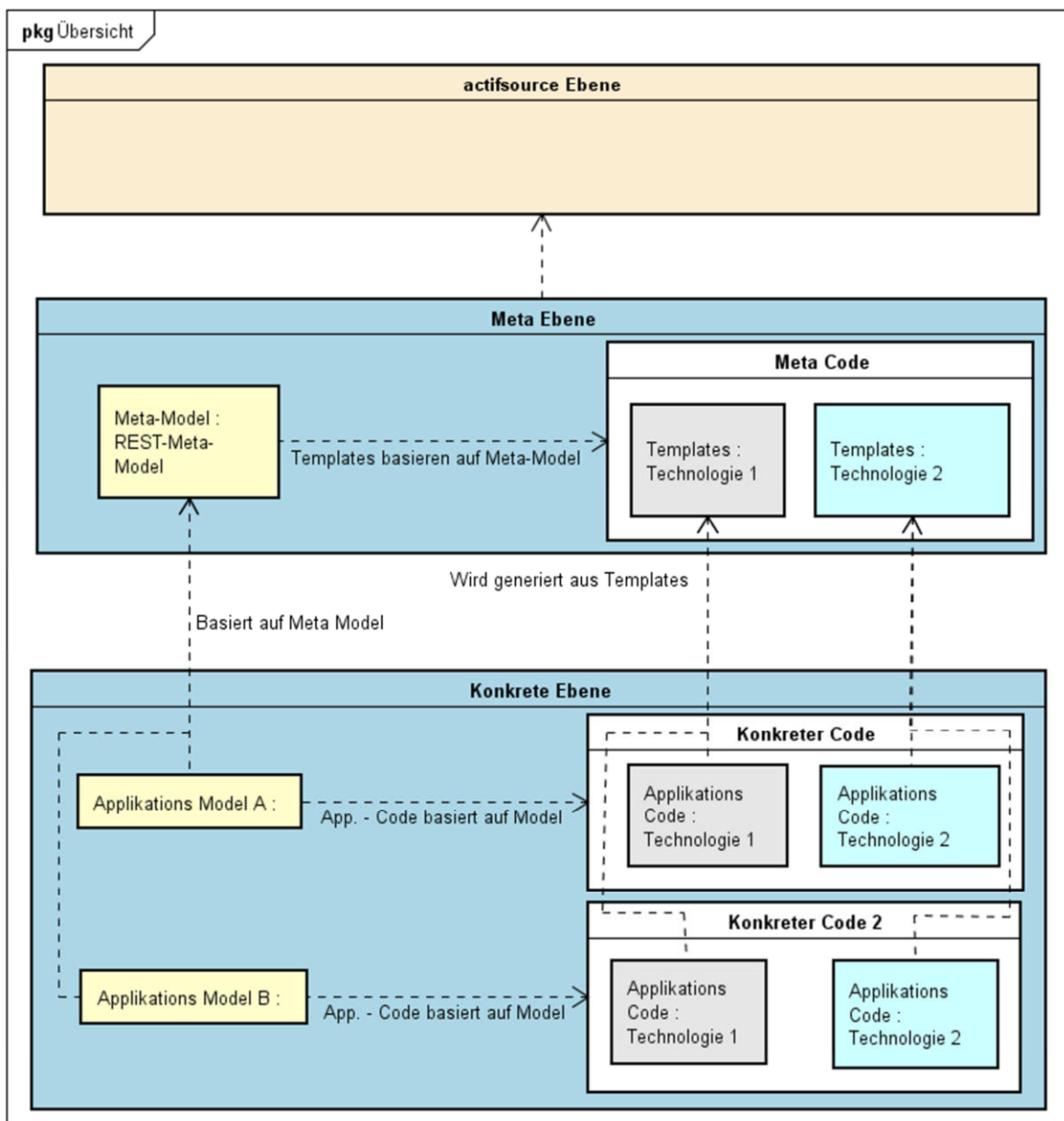


Abbildung 6: Übersicht Kontext

Die Meta-Ebene umfasst das entwickelte Meta-Model und die dazugehörigen Code-Templates. Im Bereich „Konkrete Ebene“ sind die Applikations-Modelle und der dazugehörige Code in zwei verschiedenen, nicht näher definierten Technologien ersichtlich. Eine detailliertere Übersicht bietet die *Abbildung 7*. Als Beispieltechnologien für die Code-Umsetzung werden JAX-RS, ApacheHttpServer, PHP und C# / .Net verwendet. Im Zuge dieser Arbeit sind die Templates für Java implementiert. C# / .Net und PHP wären als Zieltechnologien auch möglich. Sie sind wegen ihrer weiten Verbreitung als Alternativ-Technologien aufgeführt.

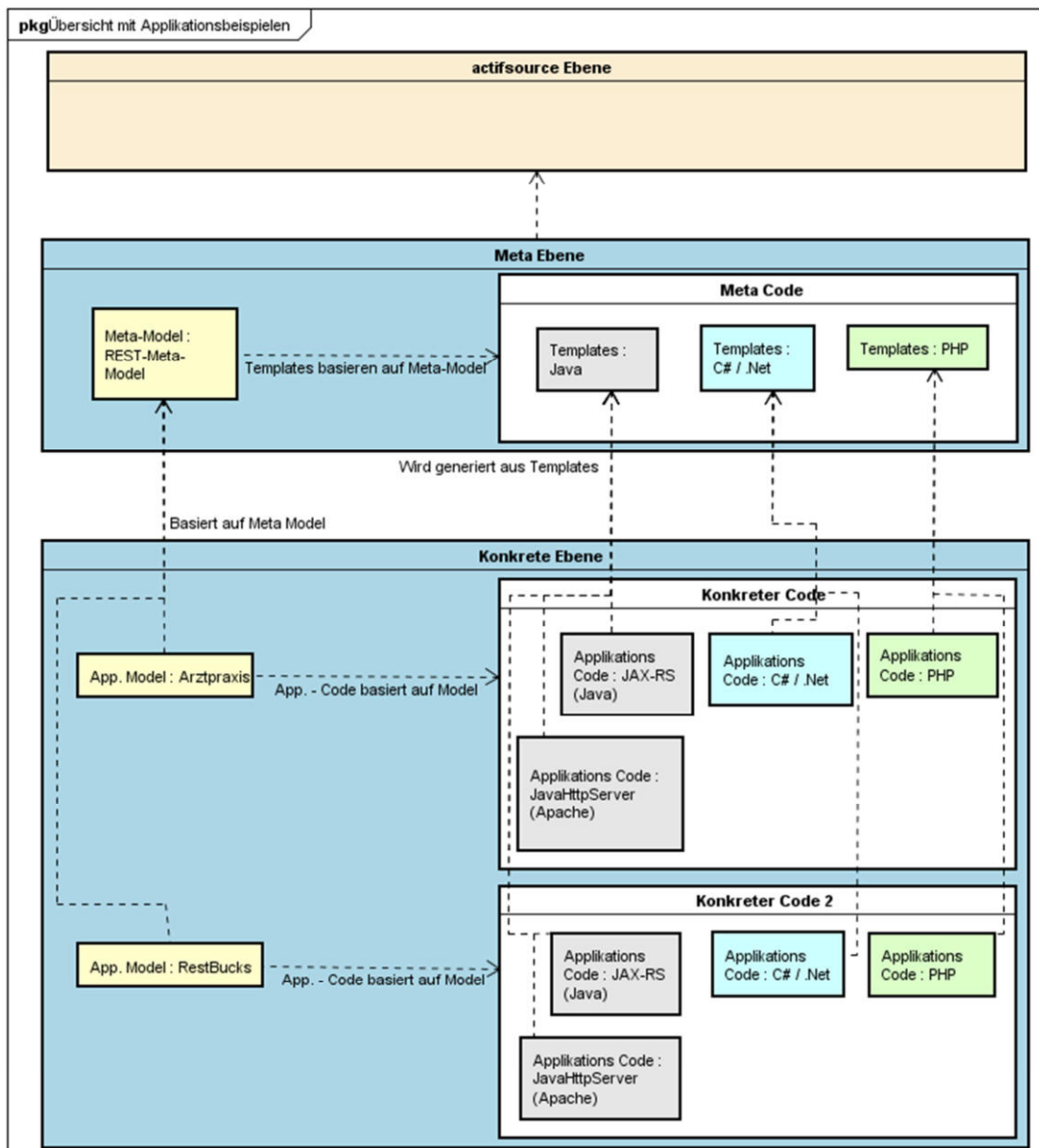


Abbildung 7: Übersicht Kontext mit Applikationsbeispielen

4.1.3 Actifsource-Ebene

Diese Ebene beinhaltet die actifsource-Umgebung.

4.1.4 Meta-Ebene

Auf dieser Ebene befindet sich das REST-Meta-Model. Dieses ist in der actifsource-Software eingepflegt. Zudem befinden sich die Code-Templates auf dieser Ebene.

4.1.4.1 Meta-Model

Dies ist das REST-Meta-Model. Dieses wird im Kapitel 4.2 *REST-Meta-Model* genauer erläutert.

4.1.4.2 Meta-Code

Der Meta-Code wird in Form von Templates in der Applikation implementiert. Diese Vorlagen werden basierend auf den Komponenten des Meta-Models erstellt. Es gibt pro Technologie einen Satz an Vorlagen. Wie diese aufgebaut sind und funktionieren wird im Kapitel 4.3 *Templates* näher beschrieben.

4.1.5 Konkrete Ebene

Die konkrete Ebene beherbergt die Business-Modelle, wie zum Beispiel „Arztpraxis“ oder „RestBucks“, welche in *Abbildung 7: Übersicht Kontext mit Applikationsbeispielen* ersichtlich sind. Ebenfalls befinden sich die konkreten Implementationen, dieser Modelle auf dieser Ebene.

4.1.5.1 REST-Schnittstellen-Modell

Schnittstellen-Modelle (*App. Model* in *Abbildung 7*) werden mit den Komponenten, welche im Meta-Model ausgearbeitet und eingepflegt wurden, erstellt.

4.1.5.2 Konkreter Code

Der konkrete Code basiert auf den Templates. Er wird automatisch generiert und aktualisiert, sobald das konkrete Modell der Applikation erstellt oder aktualisiert wird. Der Entwickler muss im konkreten Code anschliessend nur noch die Business-Logik hinzufügen, respektive den eigentliche Applikations-Code einbinden.

4.2 REST-Meta-Model

4.2.1 Einleitung

Dieses Kapitel beschreibt die Architektur des REST-Meta-Modelles sowie deren einzelnen Komponenten.

4.2.2 Übersicht

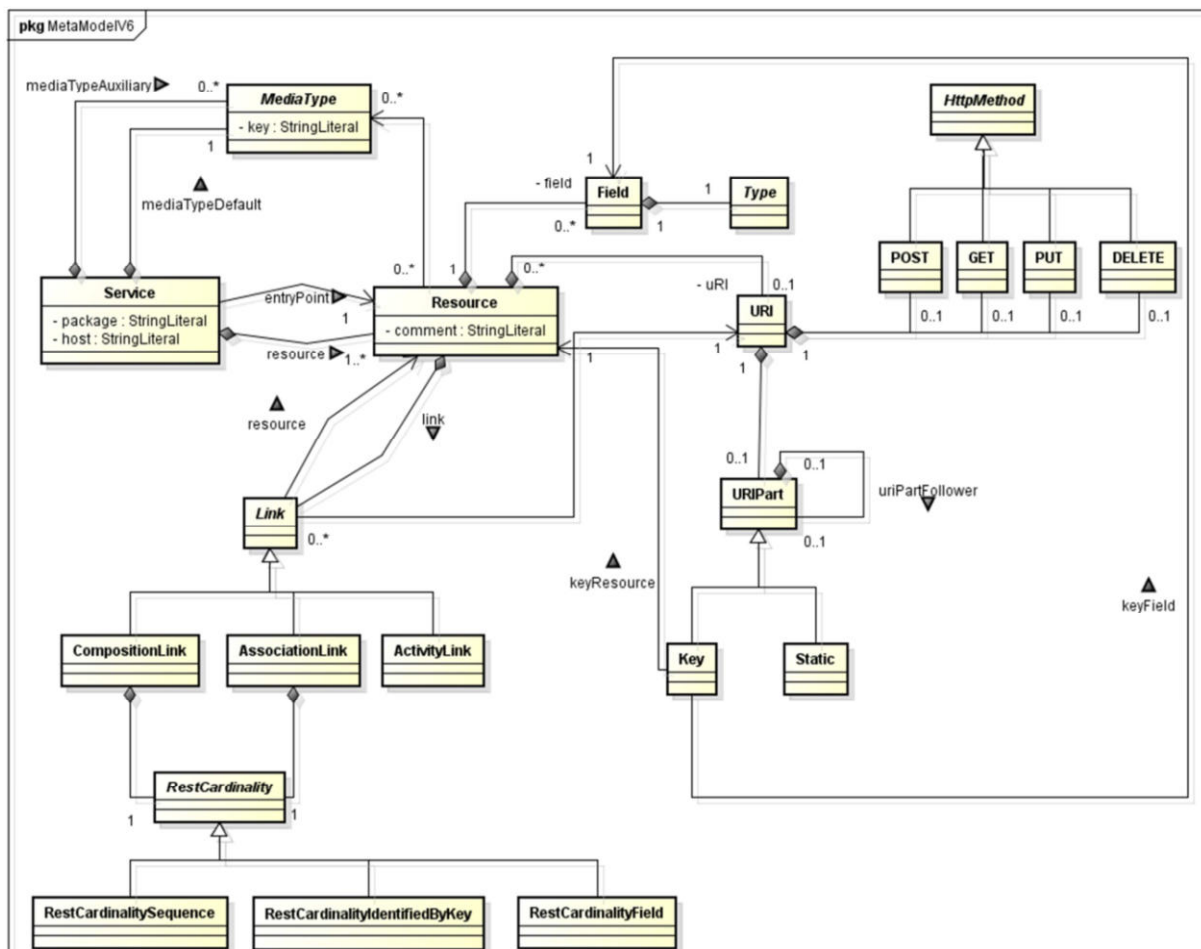


Abbildung 8: REST-Meta-Model

Die Grafik zeigt das REST-Meta-Model. Auf Basis dieses Modelles können REST-Schnittstellen modelliert werden.

Die beiden zentralen Punkte sind der Service und dessen Ressourcen (Resource). Die Ressourcen unterstützen verschiedene Medientypen. Die Felder (Field) nehmen die Rolle von Attributen der Ressourcen ein. Ein Feld hat immer ein Datentyp (Type). URI-Objekte (URI) identifizieren die Ressourcen im Netzwerk. Das URI-Objekt setzt sich aus zwei Arten von URI-Teilen (URIPart) zusammen. Statische (Static) Elemente sind reiner Text. Die Teile vom Typ Key referenzieren immer auf eine Resource und nutzen deren Informationen.

Die Links (Link) sind die Beziehungen zwischen Ressourcen. Es gibt verschiedene Arten von Links, welche jeweils eine unterschiedliche Semantik aufweisen. Sie sind im Kapitel 4.2.3.5 *Link* beschrieben.

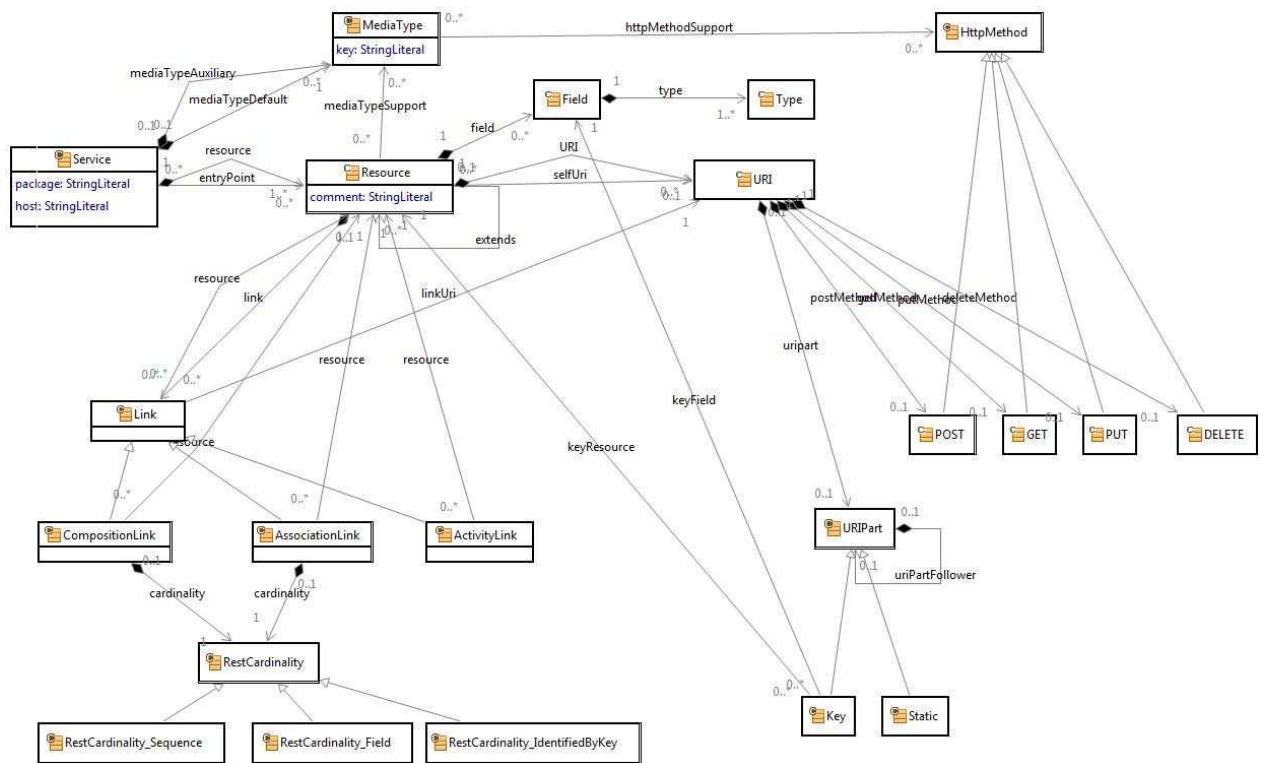


Abbildung 9: REST-Meta-Model in der actifsource Software

Die Abbildung 9 stellt dasselbe Meta-Model wie Abbildung 8 dar. Das, hier aber in der actifsource-Umgebung.

4.2.3 Komponenten

In diesem Abschnitt werden die einzelnen Komponenten des REST-Meta-Models erklärt. Jede der genannten Komponente stellt eine Ressource innerhalb der actifsource-Umgebung dar.

4.2.3.1 Service

typeOf	ch.hsr.rest.generic.Service
name	RestBucks
package	ch.hsr
host	localhost
mediaTypeDefault	XML : XML
mediaTypeAuxiliary	
entryPoint	ch.hsr.restbucks.service.RestBucks.RestBucks
resource[1]	Menu : Resource
resource[2]	OrderItem : Resource
resource[3]	Order : Resource
resource[4]	PaymentExpectedOrder : Resource
resource[5]	PayedOrder : Resource
resource[6]	TakenOrder : Resource
resource[7]	CancelledOrder : Resource
resource[8]	Coffee : Resource
resource[9]	Cookie : Resource
resource[10]	MenuItem : Resource
resource[11]	RestBucks : Resource
resource[12]	OrderConfirmation : Resource
resource[13]	Receipt : Resource

Abbildung 10: Die Konfigurationsmaske der Service-Komponente in der actifsource-Umgebung

Ein Service stellt eine REST-Schnittstelle dar. Sie unterstützt die Erfassung wichtiger Konfigurations-Attribute wie das (Java-) Package des Services, die Host-Information und den Einstiegspunkt des Services. Der Service besteht aus Ressourcen. Zudem können der Standard-Medientyp (`mediaTypeDefault`) und zusätzliche, von der Schnittstelle unterstützte, Medientypen (`mediaTypeAuxiliary`) definiert werden. Die Erfassung der Medientypen ist Voraussetzung dafür, dass eine Ressource einen Medientypen unterstützen kann. Der Service ist zudem die Basiskomponente für den Diagrammtypen.

4.2.3.2 Ressource

Die Ressourcen-Komponente stellt eine REST-Ressource innerhalb einer REST-Schnittstelle dar. Sie sind zusammen mit den Links grafisch erfassbar. Die Ressource orientiert sich stark am Konzept einer Klasse. Sie können Attribute (Field, Kapitel 4.2.3.8 *Field*) und Beziehungen zu anderen Ressourcen (Link, Kapitel 4.2.3.5 *Link*) haben, sowie von anderen Ressourcen erben.

Ein weiterer wichtiger Aspekt einer Ressource sind die URIs. Sie sind notwendig um eine Ressource richtig zu adressieren. Wie URIs aufgebaut sind und konfiguriert werden beschreibt Kapitel 4.2.3.3 *URI*. Für jede Ressource können mehrere URIs konfiguriert werden (1 in der *Abbildung 11*).

Wie bei (2) in der *Abbildung 11* gezeigt werden URIs für das Attribut `selfUri` in der Ressource verwendet. Wenn dieses konfiguriert wird, werden für die Repräsentation die Links auf die aktuelle Ressource ebenfalls angezeigt. Das Auswahlfeld für URIs ist so konfiguriert, dass nur URIs gewählt werden können, welche der Entwickler auf der aktuellen Ressource erfasst hat. Dieses Attribut ist optional.

Als Drittes verwenden die Links (3) die Ressourcen-URIs. Das Kapitel 4.2.3.5 *Link* beschreibt die Verwendung näher.

resource[1]	typeOf	ch.hsr.rest.generic.Resource
	name	Menu
	comment	Menu of Restbucks
	mediaTypeSupport	
	extends	
	field	
	selfUri 2	ch.hsr.restbucks.service.RestBucks.Menu.Menu
1	URI	Menu : URI
	link[1]	coffees : CompositionLink
	link[2]	cookies : CompositionLink 3
	link[3]	backToRestBucks : ActivityLink

Abbildung 11: Formular zum erfassen einer Ressource in actifsource

4.2.3.3 URI

Die URI-Klasse definiert URIs einer Ressource. Das URI-Objekt ist aus Teil-URIs (URIPart) zusammengesetzt. Dargestellt bei der Nummer (1) in der *Abbildung 12*. Diese Teile können statisch (Static), also reine Textelemente oder variabel (Key) sein. Variable URI-Teile haben eine Schlüssel-Ressource. Dafür zuständig ist das Attribut `keyResource` (3). Aus dieser Ressource wird das Attribut `keyField` definiert (4). Die beiden Teile, sind dazu da bei der Generierung der URIs für die Links die korrekten Felder, respektive deren Inhalt, anzusprechen.

Bei der Erfassung von URIs wird berücksichtigt, dass diese eine strikte Reihenfolge haben. Dazu ist im URIPart-Objekt das `uriPartFollower`-Attribut (2) zuständig. Für den Anwender bedeutet das, dass er, um ein URI-Objekt zu erfassen, genau einen `uripart` festlegt und weiter Teile über das `uriPartFollower` definieren muss.

Die Auswahl der Ressourcen für das `keyResource`-Attribut ist eingeschränkt. Für die auszuwählenden Resource-Objekte gilt mindestens eine der folgenden Voraussetzungen:

- Die Ressource ist diejenige, für welche das URI-Objekt definiert wird.
- Die Ressource erbt von der Ressource aus Punkt a.
- Die Ressource ist Ziel eines Links der Ressource
- Die Ressource erbt von der Ressource aus Punkt c.

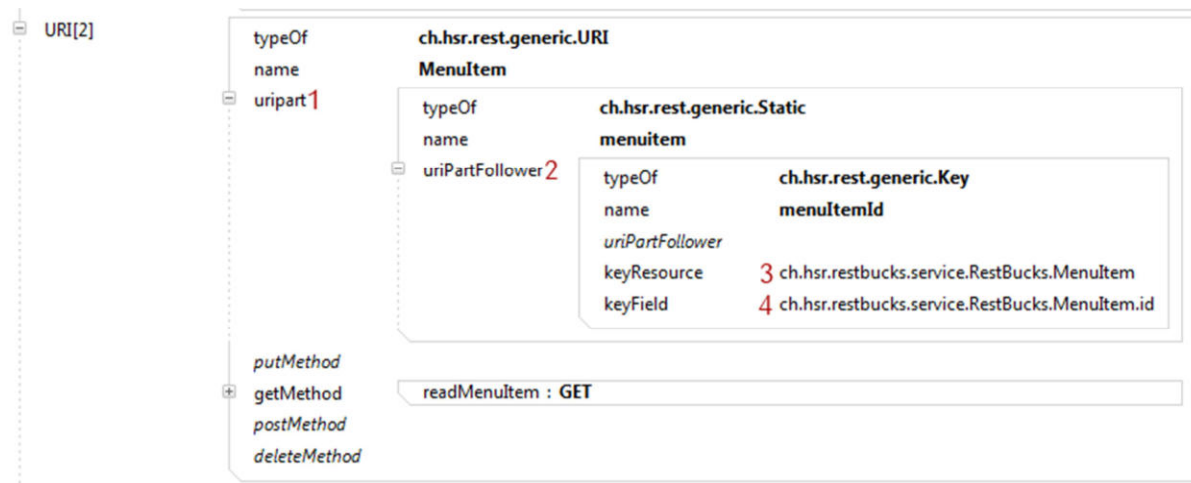


Abbildung 12: Formular zum erfassen eines URI in actifsource

Die in der *Abbildung 12* gezeigte Definition der URI „MenuItem“ hat zwei Varianten. Einerseits benutzt das `RESTUriParser-Template`²² die Definition als URI-Vorlage. Ein URI-Template sieht folgendermassen aus: „/menuItem/{menuItemId}“. Diese Variante wird ebenfalls vom `ApacheHttpUriHandler`²³ verwendet, um eingehende Requests den entsprechenden Handler zuzuordnen. Andererseits verwendet der `RestLink`²⁴ die gepaarte, konkrete Variante „/menuItem/1“.

4.2.3.4 HTTPMethod

Die `HTTPMethod` Objekte definieren welche Operationen auf einer Ressource ausgeführt werden können. Jedes URI-Objekt kann bis zu vier Objekte vom Typ `HTTPMethod` definieren. Dies wird erreicht in dem für die Attribute `putMethod`, `getMethod`, `postMethod` und `deleteMethod` eine Methode im Formular erfasst wird. Zu jeder definierten `HTTPMethod` wird in der Applikation eine eigene Controller-Methode generiert.

Der Grund weshalb die HTTP-Methoden als Attribut der URI-Objekte modelliert sind, lässt sich am besten an einen Beispiel erklären: Eine Ressource `person` hat die URI `person/{personId}`. Die `GET`-Methode erlaubt das Lesen der Ressource. Des Weiteren soll der Client eine Liste von Personen abfragen können. Die Ressource kann die zusätzliche `GET` Methode nicht implementiert, da sie bereits mit der `GET`-Methode der einzelnen Person besetzt ist. Die Lösung ist, die Methode für jeder URI zu implementieren

- `GET /person` = Liste von Personen
- `GET /person/{personId}` = Person mit entsprechender Id

²² Kapitel 4.4.4 Meta-Code für Links und URI

²³ Kapitel 4.5.4 Request-Handler-Mapper

²⁴ Kapitel 4.4.4 Meta-Code für Links und URI

Es wäre möglich eine Ressource für die Liste von Personen einzuführen. Das ist aber aufwändiger. Eine weitere Alternative bietet RAML²⁵, welches die Methode für jeden URIPart definiert. Diese Lösung ist etwas eleganter, da sie die doppelte Erfassungen von URIPart-Objekten verhindert, jedoch ist die Verwendung im Meta-Code komplizierter. Was den Entwicklungsaufwand erhöht hätte.

4.2.3.5 Link

Ein Link stellt eine Verbindung zwischen Ressourcen dar. Sie können im Diagramm des Services erfasst werden. Es gibt drei Varianten von Links, ActivityLinks, AssociationLink und CompositionLink.

Ein ActivityLink wird benötigt, um einen einfachen Übergang zwischen zwei Ressourcen zu modellieren. Dabei besitzt die Quell-Ressource eines solchen Links keinerlei Informationen zur Ziel-Ressource und dementsprechend auch keine Kardinalität. Ein gutes Beispiel für die Anwendung eines ActivityLinks sind Links zum Erstellen von Ressourcen.

Ein AssociationLink stellt eine Verbindung zwischen Ressourcen dar. Ressourcen können bei diesem Link-Typ unabhängig voneinander bestehen. Die Quellressource verfügt dabei über die Informationen der Zielressource.

Im Unterschied zum AssociationLink können bei einem CompositionLink die Ziel-Ressourcen nicht unabhängig voneinander existieren. Zum Beispiel: Wird die Quell-Ressource gelöscht, muss die Zielressource ebenfalls gelöscht werden.

Damit die Link-Informationen generiert werden können, benötigen sie eine URI-Information. Diese wird mit dem Attribut linkUri gesetzt. Die fehlerlose Generierung der Link-Objekte, ist nur möglich, wenn ein URI-Objekt ausgewählt wird, welches folgende Voraussetzung erfüllt:

1. Das URI-Objekt besitzt keinen URIPart vom Typ Key
2. Wenn das URI-Objekt eine keyResource definiert hat, muss...
 - a. ...die Quelle einen Link auf die Ressource besitzen, die dieses Objekt definiert
 - i. Der Link darf nicht vom Typ ActivityLink sein.
 - b. ...das Objekt auf der Quelle es Links definiert sein.
 - c. ...das Objekt auf dem Ziel des Links definiert sein.
3. Das URI-Objekt ist auf dem Ziel oder einer Eltern-Ressource des Ziels definiert.

Damit ist der Anwender reduziert im Design von möglichen URI-Objekten für eine Ressource. Dies ist jedoch nötig um eine fehlerlose Generierung des Codes zu ermöglichen. In einer alternativen Variante müsste der Entwickler die Code-Stücke für die Zuordnung von Link und URI selber schreiben. Konkret heisst das, er müsste für jeden verwendeten Link

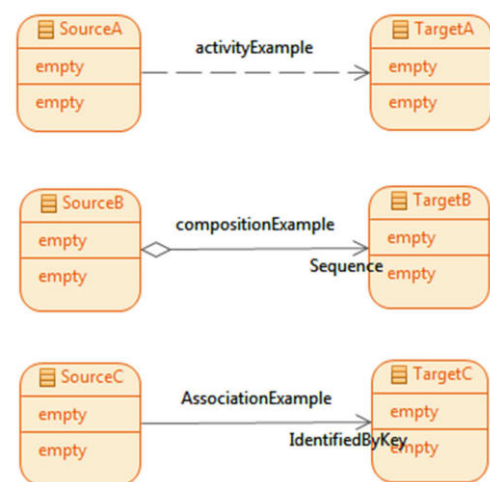


Abbildung 13: Beispiel für Link-Typen

²⁵ [RAML]

eine Code-Zeile selber schreiben und dies an unübersichtlichen Code-Stellen. Das entspricht nicht dem Ziel des Projektes.

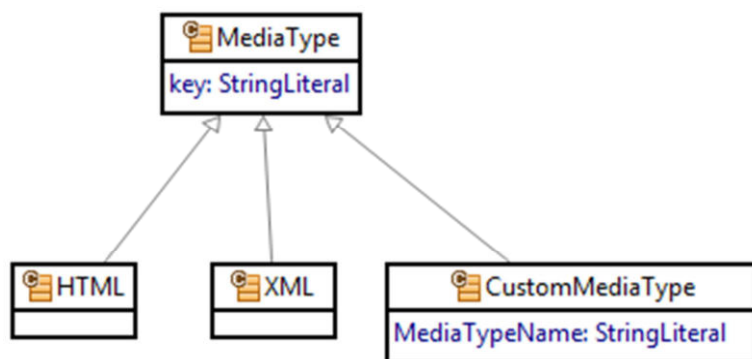
4.2.3.6 *RESTCardinality*

Es gibt drei Arten von Kardinalitäten. Diese sind in den folgenden Abschnitten beschrieben.

Bei der Kardinalität „Field“ definiert die Quellressource eine Zielressource als Attribut. Das entspricht einer 0..1 Beziehung.

Die Kardinalitäten „IdentifiedByKey“ und „Sequence“ entsprechen beide einer 0..n Beziehung. Der Unterschied ist die jeweilige Implementation in den Code-Templates. In Java übersetzt haben sie folgende Bedeutung: IdentifiedByKey entspricht einer Java-HashMap und Sequence einer Java-ArrayList. Die Benennung ist, im Hinblick auf die mögliche Unterstützung weiterer Technologien, bewusst technologie-neutral gewählt.

4.2.3.7 *MediaType*



Media Type auch MIME-Type oder Content-Type genannt ist eine Bezeichnung für die Art des Inhalts, welcher mittels HTTP übermittelt wird. Diese actifsource-Ressource und deren Untertypen werden benötigt, um die verschiedenen Medientypen, welche von Ressourcen unterstützt werden können, zu definieren.

Abbildung 14: Die Ressource MediaType und deren Untertypen

Es gibt zwei konkrete Typen, HTML und XML. Diese sind vordefiniert und werden in einem separaten Template definiert. Der CustomMediaType dient dazu, dem Entwickler die Möglichkeit zu geben, eigene Medien-Typen zu erstellen.

4.2.3.8 *Field*

Felder besitzen einen Namen und einen Typ. Eine andere Bezeichnung für Feld ist Attribut. Jedes Feld hat einen Datentyp. Dieser ist frei definierbar. Beispielsweise können so die Java-Datentypen Integer und String als Name des Typs verwendet werden.

4.2.4 *Zusätzliche Komponente*

Dieses Kapitel beschreibt die zusätzlichen Komponenten, welche von actifsource zur Verfügung gestellt werden. Diese sind nützlich für die Modellierungsapplikation.

Der Diagrammtyp ermöglicht das Erstellen von grafischen Modellen in der actifsource-Umgebung. Die Modellierungssoftware stellt diese Funktionalität mit dem Diagrammtyp Service zur Verfügung. Auf Basis der Service-Ressource ist es damit möglich Ressourcen und Links grafisch zu modellieren und darzustellen.

Die Functionspace-Objekte erleichtern die Entwicklung des Meta-Codes in dem sie komplexe oder lange Selektoren zusammenfassen. Dies ist bei häufiger Verwendung solcher

Selektoren nützlich. Die Selektoren werden im Kapitel 4.3.2 *Bestandteile und Funktionsweise eines actifsource-Templates* erklärt.

4.3 Templates

4.3.1 Beschreibung

Ein Template ist ein Werkzeug in der actifsource-Umgebung. Die Templates ermöglichen das Schreiben von Meta-Code. Sie werden in BuildConfig-Objekten erfasst. Die actifsource-Software generiert den konkreten Code anhand der Informationen in der BuildConfig. Die BuildConfig-Elemente können beliebig verschachtelt werden. Dies kommt bei den Templates, welche für eine bestimmte Zieltechnologie zuständig sind zum tragen. *Abbildung 15* zeigt die Verschachtelung solcher Konfigurationen.

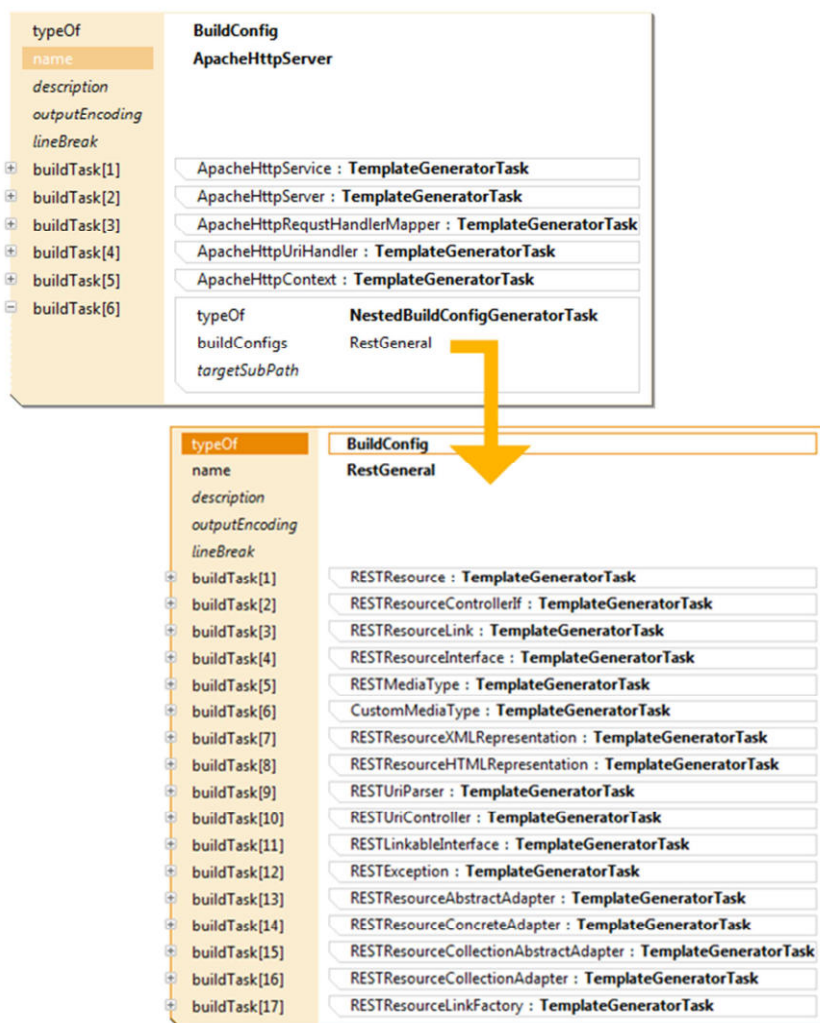


Abbildung 15: Beispiel einer actifsource-BuildConfig: Die BuildConfig für ApacheHttp bindet die Generelle BuildConfig ein

4.3.2 Bestandteile und Funktionsweise eines actifsource-Templates

Dieses Kapitel beschreibt die Bestandteile und Funktionsweise der actifsource-Templates. Folgende Konzepte sind alle von der actifsource-Umgebung definiert und finden in der Modellierungssoftware Verwendung.

Templates sind Code-Vorlagen und erlauben das Erstellen von Meta-Code. Die actifsource-Software generiert aus jedem Template eine oder mehrere Code-Dateien und deren Inhalt. Die Anzahl der erzeugten Dateien hängt von der Anzahl Resultate ihrer Kontext-Selektoren ab.

```
Selector 1 (Resource.link:AssociationLink.resource union Resource.link:CompositionLink.resource union Resource.extends) intersect Service.resource  
  
Service.baseFilePath/resources/Resource.name.toAllLower@BuiltIn/Resource.name.java 2  
Java 3  
1 package Resource.thisPackage.Resource.name.toAllLower@BuiltIn;  
2  
3 import java.util.ArrayList;  
4 import java.util.HashMap;  
5  
6 import Resource.thisPackage.Resource.name.toAllLower@BuiltIn.Resource.name;  
7  
8  
9  
10 /**  
11  * Resource Resource.name  
12  * Supported Media Types:  
13  * - MediaType.simpleName@BuiltIn 5  
14  */  
15 public class Resource.name extends Resource.name {  
16
```

Abbildung 16: Aufbau eines actifsource-Templates: Dieses Beispiel zeigt das Resource Template

Die in der *Abbildung 16* verwendeten Nummern zeigen Interessante und wichtige Aspekte eines actifsource-Templates.

Bei (1) befindet sich ein sogenannter Selektor. Die Funktion dieser Selektoren ist, den globalen Kontext des Templates, respektive die Teil-Kontexte innerhalb des Templates auf eine bestimmte actifsource-Ressource zu limitieren. Als Basis dient die actifsource-Ressource Build. Beispielsweise definiert der Selektor Build.allServices, dass der Kontext für alle Service-Objekte innerhalb des entworfenen Modelles gilt. allServices ist eine Funktion (Functionspace) der actifsource-Ressource Build. Die Resultate eines Selektors sind sowohl einzelne Ressourcen, als auch Listen von Ressourcen. Falls das Ergebnis eine Liste ist, wird der ganze Bereich des Kontextes iteriert.

Bei (2) ist der Dateiname der resultierenden Code-Datei angegeben. Wenn eine Dateiendung angegeben ist, wird bei (3) automatisch die Zielsprache eingefügt. Dafür muss die Sprache in der actifsource-Umgebung definiert sein. Sie erlaubt unter anderem das Syntax-Highlighting innerhalb des Templates. Bei (4) ist ein sogenannter LineContext zu sehen. Diese können im Template beliebig eingefügt werden und mittels Selektoren wird die zu benutzenden actifsource-Ressourcen gewählt. Alternativ zu zeilenbasiert können sie auch spaltenbasiert (ColumnContext) sein. Dies ist bei (5) in der *Abbildung 16: Aufbau eines actifsource-Templates* ersichtlich.

Es ist möglich in den generierten Code-Dateien, nicht-generierten Code zu schreiben. Dazu wird im Template ein „Protected Context“, wie bei (6), definiert. Die Verwendung solcher geschützten Bereiche ist notwendig, da bei Änderungen am Model der Code jeweils neu generiert wird und manuell ergänzter Code sonst wieder gelöscht wird. Folgende

Eigenschaften sind wichtig zu verstehen bei Verwendung von geschützten Kontexten, weil sie ein manuelles Eingreifen des Entwicklers erfordern:

- Ein Refactoring innerhalb des Modelles, wie zum Beispiel die Umbenennung einer Ressource, wird bei entsprechender Verwendung innerhalb des Bereichs nicht angewandt.
- Wenn ein geschützter Kontext im Template gelöscht wird, bleibt der Kontext am Ende der erzeugten Datei bestehen und muss manuell entfernt werden.

In den nachfolgenden drei Kapiteln werden die, für die Applikation geschriebenen Templates, genauer behandelt.

4.4 Genereller Meta-Code

4.4.1 Beschreibung

Dieser Teil des Meta-Codes wird sowohl für den Apache spezifischen (Kapitel: 4.5 ApacheHTTP spezifischer Meta-Code) als auch für den JAX-RS spezifischen Teil verwendet (Kapitel 4.6 JAX-RS spezifischer Meta-Code). Die *Abbildung 17* zeigt den Aufbau den Aufbau und die Abhängigkeiten der Templates.

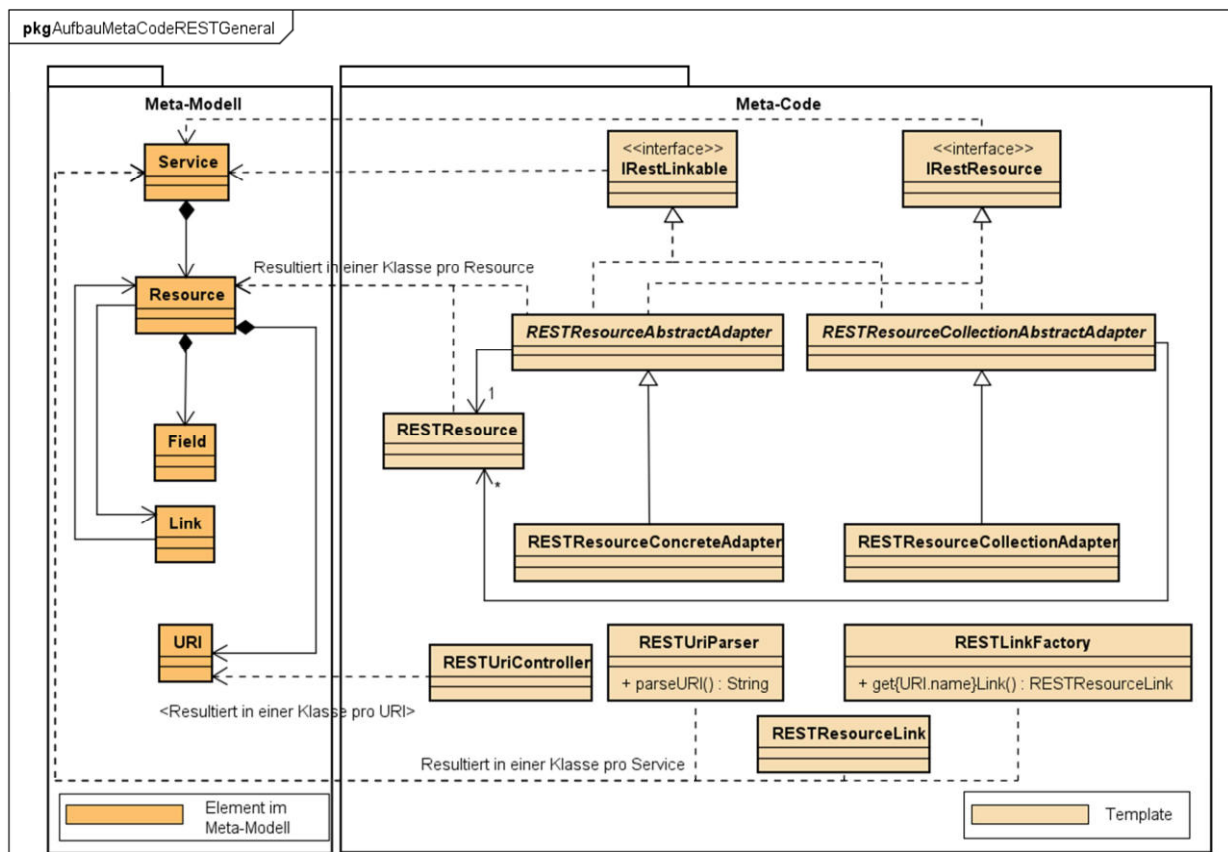


Abbildung 17: Aufbau der Templates für den Generellen Meta-Code.

Im Bereich Meta-Modell der *Abbildung 17* befinden sich die Komponenten des Meta-Modelles. Im Bereich Meta-Code befinden sich die Templates. Die gestrichelten Pfeile zwischen einem Template und einer actifsource-Ressource zeigen an, für welche Instanz-

Typen der Code generiert wird. Ein Beispiel ist das Template `RESTUriController`. Es generiert für jede im Modell definierte URI-Ressource eine URI-Controller-Code-Datei.

4.4.2 Meta-Code für Ressourcen

Das Template `RESTResource` enthält den Meta-Code für REST-Ressourcen. Das heisst, es bildet die Attribute (*Abbildung 18*) und Beziehungen der Ressource ab. Für jedes auf der REST-Ressource definierten Attributes, generiert das Template eine `get` und eine `set`-Methode innerhalb der Klasse.

```
16
17     protected Field.type.name Field.name;
50
51
52     /**
53      * Getter for Field Field.name
54      */
55     public Field.type.name getField.name.toFirstUpper@BuiltIn() {
56         return Field.name;
57     }
58
59
60
61     /**
62      * Setter for Field Field.name
63      */
64     public void setField.name.toFirstUpper@BuiltIn(Field.type.name Field.name) {
65         this.Field.name = Field.name;
66     }
```

Abbildung 18: Definition eines Feldes in einer Ressource mit `get` und `set` Methode

Actifsource erstellt für jede im Modell definierte REST-Ressource ein Klasse mit dem konkreten Code. Der Klassenname ist jeweils der Name der Ressource.

Das Template `RESTResourceInterface` definiert ein Java-Interface für einen Ressourcen-Adapter. Es legt Methoden für das Zusammenfassen aller möglichen Links und Felder einer Ressource fest. Diese können dann später einheitlich in der Repräsentation verwendet werden.

Das `RESTResourceAbstractAdapter` Template definiert einen abstrakten Adapter für jede im Modell festgelegte REST-Ressource. Das heisst, im generierten Code gibt es für jede Ressource eine Adapter-Klasse. Diese Adapter implementieren das Ressourcen Interface.

Die im Interface definierten Methoden, `getFields` und `getLinks` sind nach dem Template-Method-Pattern²⁶ implementiert, was *Abbildung 19* veranschaulicht. Für jeden vorhandenen Link einer Ressource im Model ist eine abstrakte Methode definiert. Zusätzlich ein sogenannter „Self-Link“ für die Verlinkung einer Ressource auf sich selber. Folgendes Code-beispiel zeigt die Implementation der `getLinks` – Template Methode einer Ressource.

²⁶ [WIKSCHABMETH]

```

42     /**
43      * Template method Collect Requested Links in Concrete Adapter
44      */
45     @Override
46     public HashMap<String, ArrayList<IRestLinkable>> getLinks() {
47         HashMap<String, ArrayList<IRestLinkable>> links = new HashMap<String, ArrayList<IRestLinkable>>();
48         links.put("link.name", getLink.name.toFirstUpper@BuiltInList());
49         links.put("self", getSelfLinkList());
50         return links;
51     }

```

```

    /**
    * Template method Collect Requested Links in Concrete Adapter
    */
    @Override
    public HashMap<String, ArrayList<IRestLinkable>> getLinks() {
        HashMap<String, ArrayList<IRestLinkable>> links = new HashMap<String, ArrayList<IRestLinkable>>();
        links.put("coffees", getCoffeesList());
        links.put("cookies", getCookiesList());
        links.put("backToRestBucks", getBackToRestBucksList());
        links.put("self", getSelfLinkList());
        return links;
    }

```

Abbildung 19: : Auszug aus Template und generiertem Code des `RESTResourceAbstractAdapter`

Das `RESTResourceAdapter` Template legt die konkrete Implementation eines Ressourcen-Adapters fest. Dazu implementiert es die im abstrakten Adapter definierten Template-Methoden.

Der abstrakte Adapter einer Ressource diente, in einer frühen Version der Modellierungssoftware, dazu eine zusätzliche Abstraktionsstufe einzuführen. Der Entwickler musste die URI eines Links selber setzen. Mit der Generierung der Links wurde dieses Konstrukt unnötig. Aus Zeitgründen blieb die zusätzliche Stufe bestehen.

4.4.3 Meta-Code für Repräsentationen

Für den Code der Repräsentationen sind die Templates `RESTResourceMediaType`, `RESTResourceXMLRepresentation`, `RESTResourceHTMLRepresentation` und `RESTResourceCustomRepresentation` verantwortlich. `RESTResourceMediaType` definiert ein Interface, welches von den generierten Klassen der anderen Templates implementiert wird.

Repräsentationen wandeln REST-Ressourcen in die gewünschte Medien-Typ-Notation um. Die Applikation generiert jedoch nur XML-Repräsentationen vollständig. Alle anderen Medien-Typen müssen vom Entwickler definiert und ausprogrammiert werden. Für die HTML-Repräsentation steht zudem ein eigenes Template zu Verfügung.

Grundsätzlich ist es möglich für alle Arten von Medien-Typen den Code zu automatisch zu generieren. Die Applikation erzeugt aber aus zwei Hauptgründen nur die XML-Repräsentation. Erstens ist der Benutzer in der Gestaltung der Repräsentation eingeschränkt, wenn der Code generiert wird. Dies kommt vor allem bei HTML-Repräsentationen zur Geltung. Zweitens ist die Generierung solcher Repräsentationen zeitaufwändig, wenn sie sinnvoll sein soll.

4.4.4 Meta-Code für Links und URI

`RESTResourceControllerIF` beschreibt das Interface für die URI-Controller. Dieses Interface definiert die get- und set-Methoden für Parameter. In diesem Kontext sind mit Parameter die Variablen Teile eines URIs gemeint. Je nach Technologie werden sie vom `URIHandlerMapper` (ApacheHTTP) oder mittels Annotation (JAX-RS) aus dem URI-Objekt geparkt.

RESTUriController definieren die Controller-Klassen der Schnittstelle. Sie sind für jeden festgelegten URI generiert. Ihre Methoden sind die konfigurierten HTTP-Methoden einer URI-Instanz. Es gibt zwei Typen solcher HTTP-Methoden im Template. Einerseits diejenigen, welche einen Content-Bereich besitzen (POST und PUT) und solche welche keinen aufweisen (DELETE und GET). Die Methoden der RESTUriController sind so konzipiert, dass sie das Einbinden von Applikationscode unter Zuhilfenahme von geschützten Kontexten in die REST-Schnittstelle ermöglichen. Der Rückgabewert einer solchen Methode ist ein IRestResource Interface, welches im Template RESTResourceInterface definiert wird.



```

13 public static Service.ResourceLinkClassName@RESTResourceLink getURI.nameLink(Key.keyResource.name Key.keyObject@RESTUriParser, ) {
14     String uri = UriParser.parseURI.name.toFirstUpper@BuiltIn(Key.keyObject@RESTUriParser, );
15     HashMap<String, String> methods = new HashMap<String, String>();
16
17     methods.put("GET.name", "GET.typeOf.name");
18     methods.put("PUT.name", "PUT.typeOf.name");
19     methods.put("POST.name", "POST.typeOf.name");
20     methods.put("DELETE.name", "DELETE.typeOf.name");
21
22     return new Service.ResourceLinkClassName@RESTResourceLink("URI.-URI.name.toAllLower@BuiltIn", uri, methods);
23 }
    
```

```

/**
 *
 */
public static RESTLink getreservedSlotOfDoctorLink(Doctor doctor, ReservedSlot reservedSlot) {
    String uri = UriParser.parseReservedSlotOfDoctor(doctor, reservedSlot);
    HashMap<String, String> methods = new HashMap<String, String>();

    methods.put("readSlotOfDoctor", "GET");
    methods.put("modifySlotOfDoctor", "PUT");
    methods.put("removeSlotOfDoctor", "DELETE");

    return new RESTLink("reservedslot", uri, methods);
}
    
```

Abbildung 20: Auszug aus Template und generiertem Code der RESTLinkFactory

Die RESTLinkFactory ist zuständig für das erstellen von RESTLink-Instanzen. Der Aufbau der RESTLink-Klasse zeigt die *Abbildung 21*. Für jeden im Modell definierten URI wird in der Factory eine Methode generiert. Wie diese Methoden im Template und im generierten Code aussehen, kann der *Abbildung 20* entnommen werden. Die Parameter einer Methode in der RESTLinkFactory sind die KeyResources der, in der URI-Konfiguration verwendeten URIParts vom Typ Key.

Beim Erstellen der Repräsentation, wandelt die dafür zuständige Klasse die RESTLink-Objekte in die entsprechende Link-Notation des Medien-Typs um. Als Beispiel für eine solche Notation der XML-Repräsentation zeigt der folgende Code Ausschnitt:

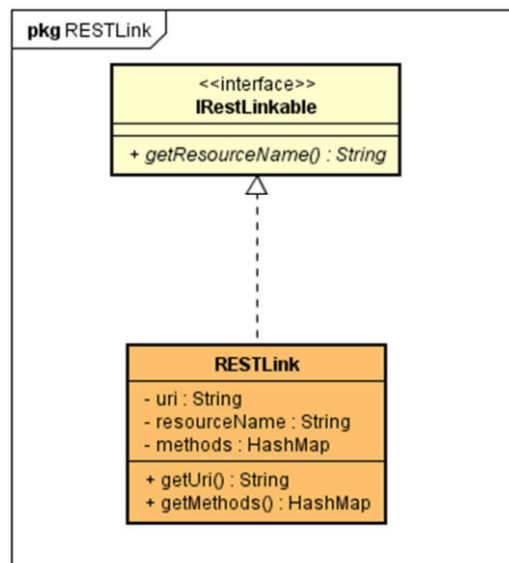


Abbildung 21: Model der RESTLink Klasse

```

1. <reservedslot>
2. <link href="/doctor/mjones/reservedSlot/1" method="PUT" rel="modifySlotOfDoctor"/>
3. <link href="/doctor/mjones/reservedSlot/1" method="DELETE" rel="removeSlotOfDoctor" />
4. <link href="/doctor/mjones/reservedSlot/1" method="GET" rel="readSlotOfDoctor"/>
5. </reservedslot>
    
```

Für das korrekte Erstellen der URIs ist der RESTUriParser zuständig. Wie die RESTLinkFactory, wird er für jeden definierten Service erstellt. Zudem besitzt er für jeden konfigurierten URI

eine Methode. Ein Auszug aus dem Template und dem generierten Code wird in der *Abbildung 22* dargestellt.

```

12     /**
13     * Parse URI: URI.name
14     */
15     public static String parseURI.name.toFirstUpper@BuiltIn(Key.keyField.-field.name Key.keyObject, ) {
16         String uri = "URI.buildURITemplate@ApacheHttpUriHandler";
17         uri = uri.replace("{Key.name}", Key.keyObject.getKey.keyField.name.toFirstUpper@BuiltIn());
18         return uri;
19     }

```

```

/**
 * Parse URI: reservedSlotOfDoctor
 */
public static String parseReservedSlotOfDoctor(Person doctor, Slot reservedSlot) {
    String uri = "/doctor/{doctorId}/reservedSlot/{reservedSlotId}";
    uri = uri.replace("{doctorId}", doctor.getName());
    uri = uri.replace("{reservedSlotId}", reservedSlot.getId());
    return uri;
}

```

Abbildung 22: Auszug aus Template und generiertem Code des RESTUriParsers

4.4.5 REST-Exception

Das Template RESTException definiert die gleichnamige Klasse. Damit können Exceptions innerhalb der Applikation, mit passenden HTTP-Status-Codes und Mitteilungen, abgefangen und geworfen werden.

4.5 ApacheHTTP spezifischer Meta-Code

4.5.1 Beschreibung

ApacheHTTP ist die primäre Ziel-Technologie der Arbeit. Die Bibliothek erfordert dass der Web-Server vollständig selber erstellt wird. Die Bibliothek ist deswegen besonders gut geeignet um festzustellen, ob das erstellte Meta-Model vollständig ist. Zudem stellt die Bibliothek sowohl Klassen für die Client-Entwicklung, als auch für die Server-Entwicklung zur Verfügung. Für den produktiven Einsatz als Server-Technologie ist sie jedoch nicht geeignet. Denn es muss zu viel Funktionalität eines Web-Servers manuell umgesetzt werden.

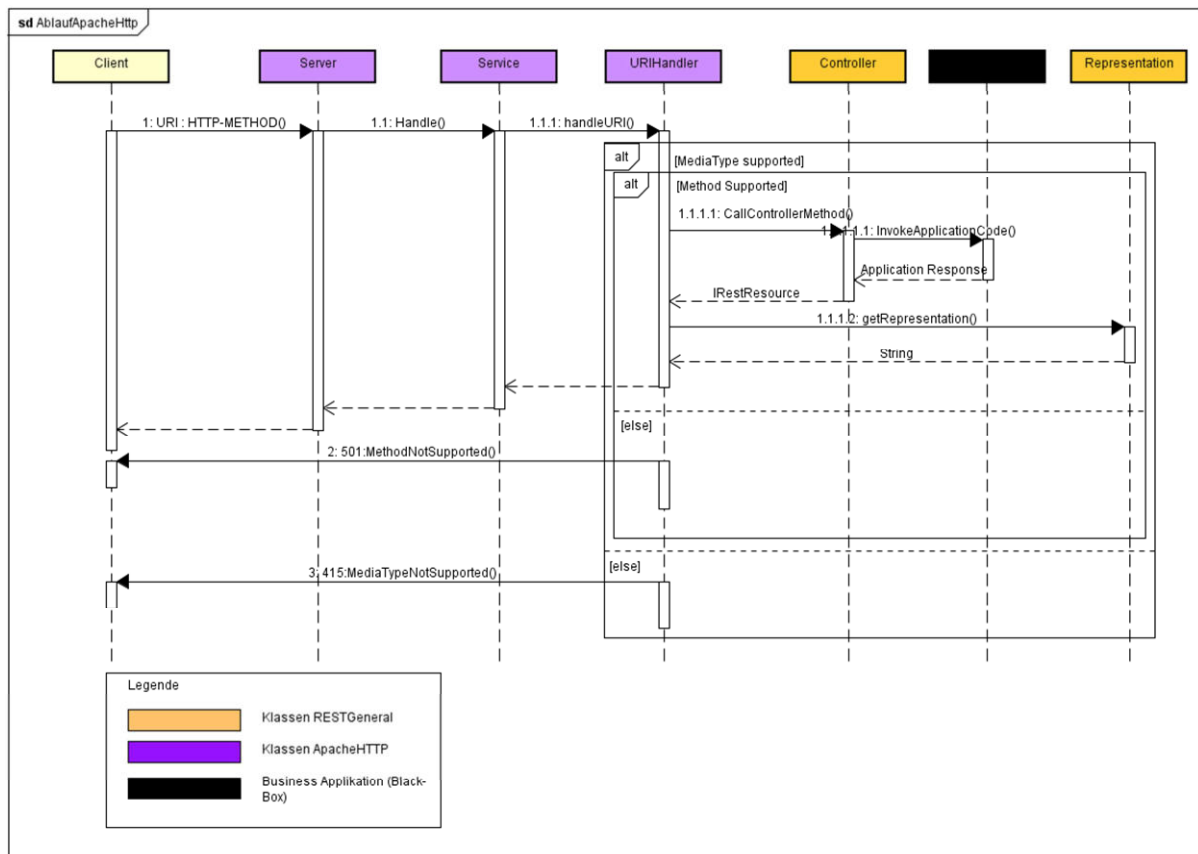


Abbildung 23: Serverseitiger Ablauf der ApacheHTTP Variante als UML-Sequenz-Diagramm

4.5.2 Server

Der Server hört den HTTP-Server-Socket ab und startet einen neuen Thread, sobald eine Anfrage eines Clients eingeht. Pro Service wird nur eine Server-Klasse generiert.

4.5.3 Service

Der Service ist ein Container für alle Informationen die eine REST-Schnittstelle braucht. Er verwaltet die Informationen zu den Medien-Typen und Schnittstellen.

Pro Service existiert nur eine Service-Klasse.

4.5.4 Request-Handler-Mapper

Pro Service ist genau eine Request-Handler-Mapper-Klasse vorhanden. Die ApacheHTTP-Bibliothek definiert einen eigenen URI Handler. Dieser ist jedoch in seiner Funktionalität eingeschränkt. Er beschränkt sich bloss auf statische Komponenten von URIs. Die variablen Teile können allein mittels Wildcard (*-Notation) erkannt werden, sofern sie zu Beginn oder am Ende eines URIs stehen. Zum Beispiel: /ressource/* würde erfolgreich mappen, /ressource*/ressource2 jedoch nicht. Deshalb stellt die Modellierungsapplikation einen eigenen URI-Handler zur Verfügung, der dem ausgearbeiteten URI-Konzept mächtig ist.

4.5.5 URI-Handler

Für jeden im Modell definierten URI existiert eine URI-Handler-Klasse. Diese beinhaltet eine handle-Methode, wie es das ApacheHTTP vorgegebene Interface `HttpRequestHandler` verlangt. Diese Methode prüft die Gültigkeit der Client-Anfrage und leitet sie an die entsprechende Methode innerhalb der Klasse weiter. Zudem bereitet sie die Antwort an den Client vor. Die handle-Methode prüft die Gültigkeit der Anfrage anhand von zwei Kriterien. Erstens muss der vom Client verlangte Medien-Typ von der REST-Schnittstelle allgemein, oder von der Ressource im spezifischen unterstützt sein. Zweitens muss die vom Client verwendete HTTP-Methode implementiert sein. Im Fehlerfall antwortet die handle-Methode mit dem HTTP-Status-Code 415 für einen nicht unterstützten Medientyp oder, 501 für eine nicht unterstützte Methode.

Das Template definiert eine Methode für jede auf dem URI konfigurierte HTTP-Methode. Diese kümmern sich um die Weiterleitung der Anfrage auf die Controller.

4.6 JAX-RS spezifischer Meta-Code

4.6.1 Beschreibung

JAX-RS ist eine JAVA-API für REST-Schnittstellen²⁷. Es gibt Implementationen für verschiedene Java-basierte Server-Plattformen, wie zum Beispiel GlassFish oder Apache Tomcat. JAX-RS nutzt Java-Annotationen²⁸, um die REST-Komponenten in Java zu identifizieren.

Die Modellierungssoftware benutzt die Jersey-Implementation²⁹ in der Version 1.18 als Bibliothek für die Definition der URI-Handler (Kapitel: *4.6.2 JaxRsUriHandler*) und des Web-XMLs.

Die Jersey-Implementation ist bereits in der Version 2.4.1 erhältlich. Diese Version ist für die Server-Technologie GlassFish optimiert. Der Zielsever der Modellierungsapplikation ist jedoch der Apache Tomcat.

Die JAX-RS eignet besser für den produktiven Einsatz, als die ApacheHTTP-Bibliothek. Dies ist unter anderem der Fall, da für diese Technologie lediglich noch der Web-Applikationscode nötig ist. Als Webserver kann ein bereits vorhandener Tomcat-Server verwendet werden. JAX-RS arbeitet mit der JEE-Servlet-Technologie.

²⁷ [JAXRS]

²⁸ [WIKIJAVANNOT]

²⁹ [JERSEY]

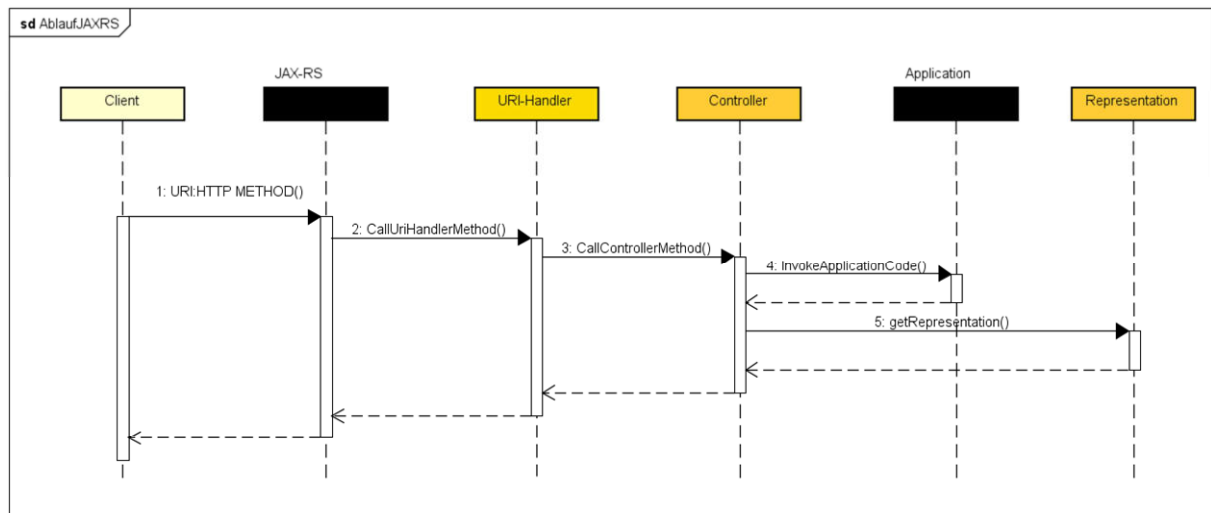


Abbildung 24: Serverseitiger Ablauf der JAX-RS Variante als UML-Sequenz-Diagramm

Im Vergleich mit dem Ablauf der ApacheHTTP Variante in *Abbildung 23* zeigt sich in der *Abbildung 24*, dass für JAX-RS weniger Klassen notwendig sind. Es sind lediglich zwei technologiespezifische Templates nötig. Des Weiteren sind in der *Abbildung 23* zwei Black-Box Bereiche ersichtlich. JAX-RS leitet die Anfrage des Clients anhand von Java-Annotationen an die Handler weiter. Der Bereich Applikation ist wiederum der REST-Schnittstelle unterliegender Business-Applikations-Code. Die Verwendung des spezifischen Codes der REST-Schnittstellen-Webanwendung ist die Selbe wie bei ApacheHTTP.

4.6.2 JaxRsUriHandler

Dieses Template ist die Vorlage für die JAX-RS spezifischen Code-Teil. Für jeden URI wird eine Klasse erzeugt. Da JAX-RS überwiegend mit Java-Annotationen arbeitet, werden diese in die Klasse generiert.

Das Template verwendet folgende Annotationen:

`@Path`: Bestimmt welcher URI für die Klasse zuständig ist. In den Template wird die Annotation einzig auf Ebene der Klasse verwendet. Es ist ausserdem möglich sie auf Methoden-Ebene zu verwenden. Dabei ist zu beachten, dass diese ergänzend zu den Klassen-Annotationen wirken. Beispielsweise wenn in der Klasse der Pfad `/ressource/` und in der Methode `/ressource/1/` verwendet wird, ruft JAX-RS die Methode lediglich auf, wenn vom Client die Anfrage `/ressource/ressource/1` lautet.



```

@Path("URI.buildURITemplate@ApacheHttpUriHandler")
public class URI.thisClassName {

    @HttpMethod.typeOf.name
    @Produces("MediaType.key")
    public String HttpMethod.nameMediaType.typeOf.name(@PathParam("Key.name")Key.keyField.type.name Key.name, @Context ServletContext ctx) {
        StringBuffer responseContent = new StringBuffer();
        CustomMediaType.MediaTypeNameRepresentation mediaType = new CustomMediaType.MediaTypeNameRepresentation();
        MediaType.name@ExportPackageTemplate mediaType = new MediaType.name@ExportPackageTemplate();

        HashMap<String, String> parameter = new HashMap<String, String>();
        HashMap<String, Object> context = new HashMap<String, Object>();
    }
}

@Path("/doctor/{doctorId}")
public class Doctor {

    @GET
    @Produces("vnd.hsr.doctorservice+xml")
    public String readDoctorXML(@PathParam("doctorId")String doctorId, @Context ServletContext ctx) {
        StringBuffer responseContent = new StringBuffer();
        DoctorServiceJAXRS_XML mediaType = new DoctorServiceJAXRS_XML();

        HashMap<String, String> parameter = new HashMap<String, String>();
        HashMap<String, Object> context = new HashMap<String, Object>();
    }
}

```

Abbildung 25: Template und generierter Code eines URI-Handlers in JAX-RS

@GET, @DELETE, @POST, @PUT: Gibt die Art der Methode an, respektive für welches HTTP-Idiom die URI-Handler-Methode aufgerufen wird.

@Produces: Zeigt an, welchen Medientyp die Methode produziert. Anders als in den HTTP-Handlern, generiert das Template für jede im Modell definierte URI-Handler-Methode und jeden festgelegten Medientypen eine eigene Methode. Die Anzahl Handler-Methoden ist demnach doppelt so hoch als beim Gegenstück von ApacheHTTP.

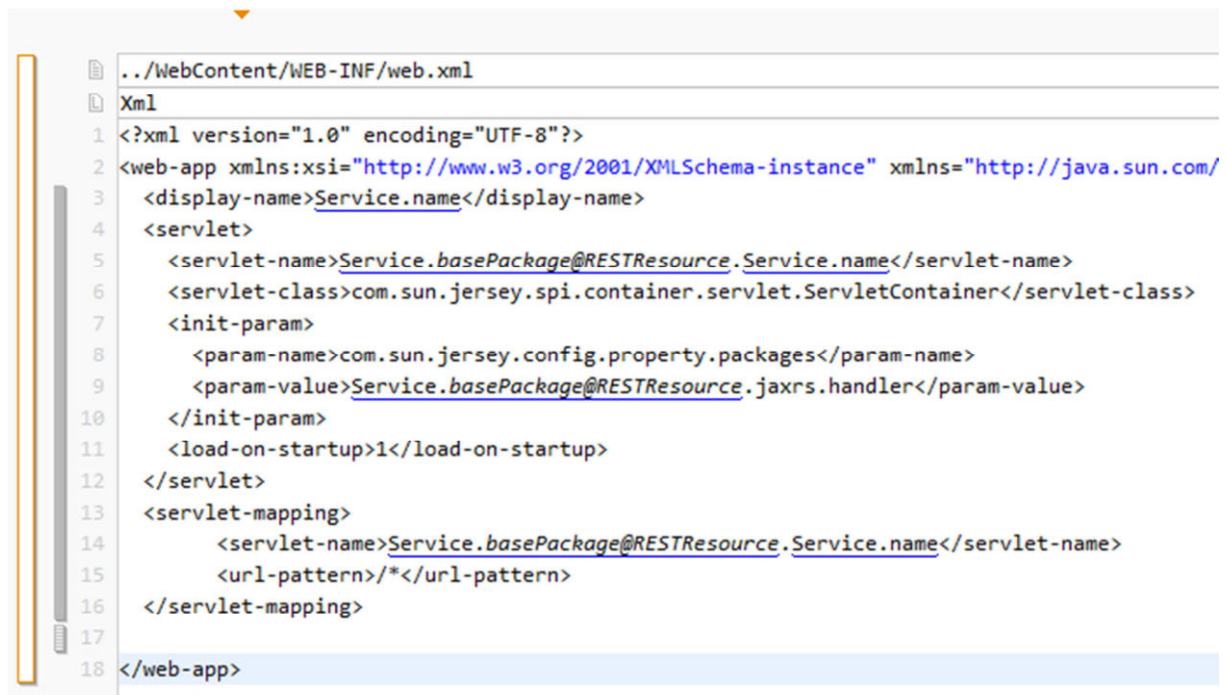
@Consumes: Bestimmt, welchen Medientyp der Inhalt einer Client-Anfrage aufweisen muss damit die Methode aufgerufen wird. Dies gilt nur für PUT und POST-Idiome, denn die Anderen dürfen keinen Request-Body besitzen.

@PathParam: Extrahiert variable Teile der URI und setzt die Parameter-Variable mit dessen Wert. JAX-RS identifiziert die variablen Teile des URI anhand von geschweiften Klammern. Wie in der *Abbildung 25* ersichtlich wird der URI-Teil {doctorId} der Variable doctorId zugewiesen.

@Context: Diese Annotation verhilft der Methode zum Zugriff auf den ServletContext in Form eines Methodenparameters. Diese Variable erlaubt den Zugriff auf Daten in der Servlet-Umgebung.

4.6.3 Webxml

Das web.xml konfiguriert den Web-Server. Generiert wird nur das Minimum. Die Datei ist mittels eines geschützten Bereichs erweiterbar.



```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/
3   <display-name>Service.name</display-name>
4   <servlet>
5     <servlet-name>Service.basePackage@RESTResource.Service.name</servlet-name>
6     <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>
7     <init-param>
8       <param-name>com.sun.jersey.config.property.packages</param-name>
9       <param-value>Service.basePackage@RESTResource.jaxrs.handler</param-value>
10    </init-param>
11    <load-on-startup>1</load-on-startup>
12  </servlet>
13  <servlet-mapping>
14    <servlet-name>Service.basePackage@RESTResource.Service.name</servlet-name>
15    <url-pattern>/*</url-pattern>
16  </servlet-mapping>
17
18 </web-app>
```

Abbildung 26: Das webxml-Template

4.7 Ergänzende Templates

4.7.1 Beschreibung

Die in diesem Kapitel beschriebenen Templates dienen als Helfer für die REST-Schnittstelle. Sie sind als separate actifsource-BuildConfig im Projekt vorhanden. Die ergänzenden können nach Belieben in den Projekten verwendet werden.

4.7.2 SQLite-Template

Das SQLite Template generierte pro Service einen Datenbank-Adapter für die REST-Schnittstelle. Für jede REST-Ressource wird eine Datenbank-Tabelle erstellt. Zusätzlich generiert das Template für jede Ressource eine Zugriffsmethode. Einige Methoden müssen für die Verwendung ausprogrammiert werden. Dafür stehen geschützte Bereiche innerhalb der Templates zur Verfügung

4.7.3 JUnit-Template

Das JUnit Template generiert die Testklassen für die Controller (Kapitel: 4.4.4 *Meta-Code für Links und URI*). Das heisst für jede Controller-Klasse wird eine Test-Klasse und für jede Methode innerhalb der Controller eine Testmethode definiert. Die Tests basieren auf dem JUnit-Framework von Java. Für die korrekte Verwendung müssen die erzeugten Klassen ausprogrammiert werden. Dafür stehen geschützte Bereiche zur Verfügung.

5 ANWENDUNG DER KONZEPTE AUF APPLIKATIONSBEISPIELE

5.1 Beispiel 1: Arztpraxis

5.1.1 Einleitung

Dieses Beispiel stammt aus einem Blogbeitrag von Martin Fowler zum Richardson Maturity Model³⁰. Es beschreibt einen RESTful Service, welcher es für Patienten möglich macht, einen Arzttermin zu reservieren.

5.1.2 Ausgangslage

5.1.2.1 Domain Model

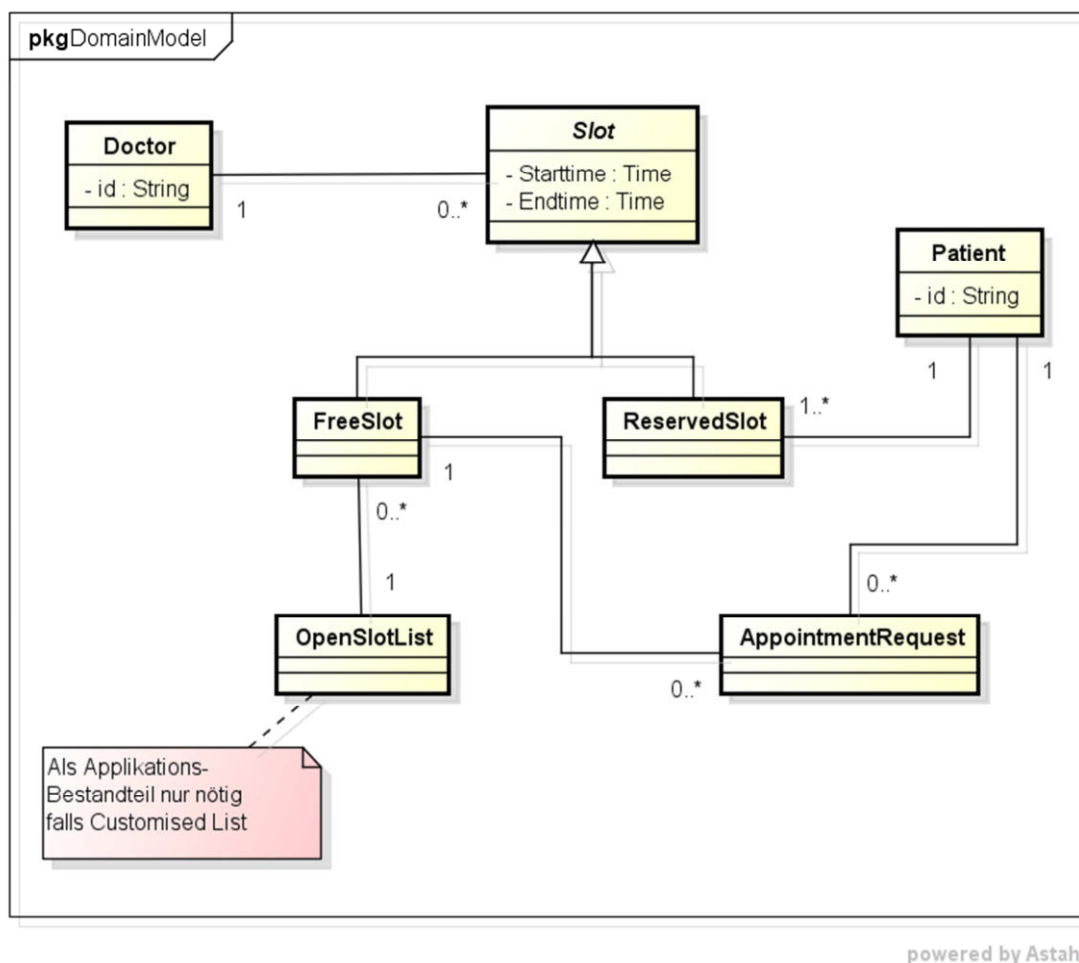


Abbildung 27: Domainmodel für die Terminreservierung

³⁰ [FowWeb]

5.1.3 Erkenntnisse

5.1.3.1 REST-Komponenten

Ressourcen

Ressource	Bemerkungen
Doctor	Über diese Ressource können Slots für Termine abgefragt werden.
Slot	Slots sind entweder frei oder belegt. Über diese Ressource können Termin-Anfragen getätigt werden
Patient	Über diese Ressource können die Kontaktdaten des Patienten abgefragt werden.
Appointment	Subressource eines Slots. (Ist das immer noch eine Ressource?)
ContactInfo	Subressource eines Patienten.

HTTP-Methoden

Verb	Beschreibung
GET	Alle lesenden Anfragen auf eine Ressource.
POST	Neue Terminanfragen senden.

5.1.3.2 Medientypen³¹

Media-Type	Beschreibung
application/xml; text/xml	Ober-Typ für XML-Content

³¹ [IANA]

5.1.3.3 HATEOAS / REST LEVEL 3

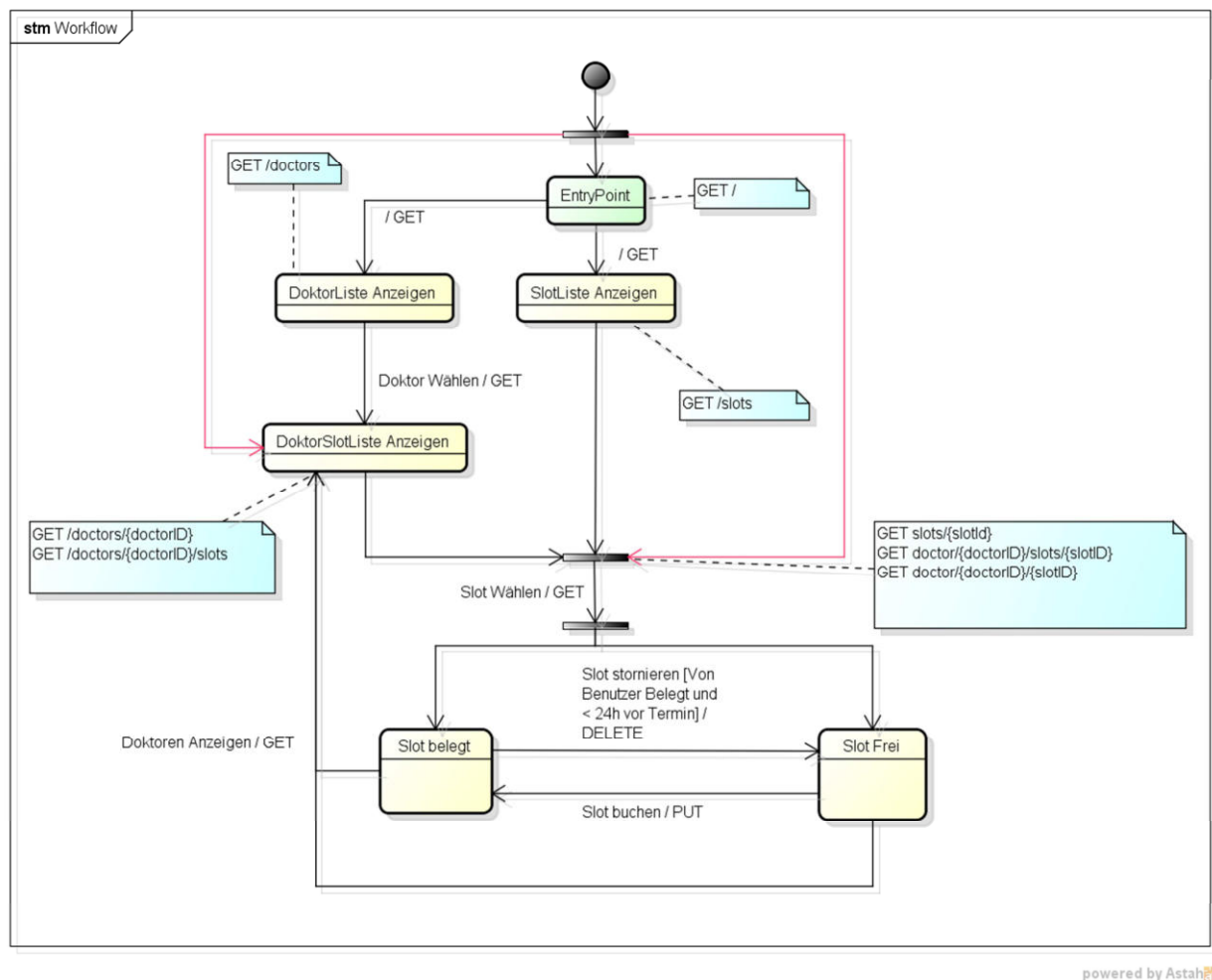


Abbildung 28: Workflow / Erreichbarkeit der Ressourcen

Im obigen Diagramm zeigt sich der Ablauf der Schnittstelle. Dem Client wird zu Beginn nur der „EntryPoint“ bekannt gemacht. Alle schwarzen Pfeile sind ausgehende Links einer Ressource innerhalb des Workflows und müssen dem Client bekannt gemacht werden.

Es gibt zwei verschiedene mögliche Workflows. Über einen bestimmten Doktor wird einer seiner Slots gewählt oder ein Slot wird unabhängig vom Arzt gewählt. Um in der Applikation nur einen der beiden Abläufe zu erlauben, kann der „EntryPoint“ entweder durch die Ressource „DoktorListe Anzeigen“ oder die Ressource „SlotListe Anzeigen“ ersetzt werden.

Rote Pfeile zeigen, dass die Ressourcen auch direkt aufgerufen werden könnten.

Für eine optimale Benutzerführung gilt: $\text{Anzahl Links} = \text{Ausgehender Pfeil} * \text{Medientyp der Zielressource}$. Für die roten, direkten Links gilt: Falls nichts angegeben wird der Standard (Default) Medientyp verwendet. Dies ist wichtig, da ein Browser nicht unbedingt den Content-Type mitschickt, sondern mit dem „Accept“ Header³² arbeitet. Dieser kann von spezifizierten Medientypen bis zu einer Wildcard reichen (*/*)

³² [W3org]

5.1.3.4 Kommunikation

Ablauf von Entry-Point nach Slot

Ein Beispiel für XML – Kommunikation, gemäss Ablauf in Abbildung 28: Workflow / Erreichbarkeit der Ressourcen.

Anfrage: GET /

Antwort: Status Code 200; Status Message: „OK“

```
1. <application>
2.   <link method="GET" rel="doctors" href="http://localhost:8000/doctor/" />
3.   <link method="GET" rel="slot" href="http://localhost:8000/slot/" />
4. </application>
```

Anfrage: GET /doctor/

Antwort: Status Code: 404; Status Message: “Your requested doctor does not exist”

```
1. <doctors>
2.   <msg>
3.     Your requested doctor does not exist Available Doctors are:
4.   </msg>
5.   <link method="GET" rel="self" href="http://localhost:8000/doctor/jsmith" />
6.   <link method="GET" rel="self" href="http://localhost:8000/doctor/mjones" />
7. </doctors>
```

Anfrage: GET /doctor/jsmith

Antwort: Status Code 200; Status Message: „OK“

```
1. <doctor>
2.   <name>jsmith</name>
3.   <freeSlots>
4.     <slot>
5.       <link method="GET" rel="slot/5" href="http://localhost:8000/slot/5" />
6.       <link method="PUT" rel="slot/reserve" href="http://localhost:8000/slot/5" />
7.     </slot>
8.     <slot>
9.       <link method="GET" rel="slot/4" href="http://localhost:8000/slot/4" />
10.      <link method="PUT" rel="slot/reserve" href="http://localhost:8000/slot/4" />
11.    </slot>
12.  </freeSlots>
13.  <reservedSlots/>
14.  <link method="GET" rel="self" href="http://localhost:8000/doctor/jsmith" />
15. </doctor>
```

Anfrage: GET /slot/4

Antwort: Status Code 200; Status Message: „OK“

```
1. <slot>
2.   <id>4</id>
3.   <start>Thu Jan 01 01:00:01 CET 1970</start>
4.   <end>Thu Jan 01 01:00:01 CET 1970</end>
5.   <status>Free</status>
6.   <link method="GET" rel="self" href="http://localhost:8000/slot/4" />
7.   <link method="PUT" rel="slot/reserve" href="http://localhost:8000/slot/4" />
8.   <doctor>
9.     <link method="GET" rel="doctor/jsmith" href="http://localhost:8000/doctor/jsmith" />
```

```
10.     </doctor>
11. </slot>
```

Slot Reservieren

Anfrage: PUT /slot/4

```
1. <patient>
2.   <name>myName</name>
3. </patient>
```

Antwort: Status Code: 200; Status Message: „OK“

```
1. <slot>
2.   <msg>Slot 4 booked for patient: myName</msg>
3.   <link method="GET" rel="self" href="http://localhost:8000/slot/4" />
4.   <link method="DELETE" rel="slot/cancel" href="http://localhost:8000/slot/4" />
5. </slot>
```

Slot canceln

Anfrage: DELETE /slot/4

Antwort: Status Code: 200; Status Message „OK“

```
1. <slot>
2.   <msg>Slot 4 cancelled</msg>
3.   <link method="GET" rel="self" href="http://localhost:8000/slot/4" />
4.   <link method="PUT" rel="slot/reserve" href="http://localhost:8000/slot/4" />
5. </slot>
```

Fehlerfall I: Von der Ressource nicht unterstützte Methode

Anfrage: POST /doctor/jsmith

Antwort: Status Code 405; Status Message „Method not allowed“

```
1. <doctor>
2.   <msg>Method not allowed: POST</msg>
3. </doctor>
```

Fehlerfall II: Von der Ressource nicht unterstützter Medientyp

Anfrage: GET /doctor/jsmith (mit Content-Type: application/json)

Antwort: Status Code 415; Status Message „unsupported Media Type“

```
1. <application>
2.   <msg>Unsupported Media Type</msg>
3. </application>
```

5.1.3.5 Bemerkungen

In diesem Beispiel gibt es einen Link, der mit URI-Tunneling arbeitet (Zeile 8). Zudem wird häufig mit Subressourcen gearbeitet. Zum Beispiel: slots/1234/appointment/test.

```
1. <link rel = "/linkrels/appointment/cancel"
2.   uri = "/slots/1234/appointment"/>
3. <link rel = "/linkrels/appointment/addTest"
```



```

4.     uri = "/slots/1234/appointment/tests"/>
5. <link rel = "self"
6.     uri = "/slots/1234/appointment"/>
7. <link rel = "/linkrels/appointment/changeTime"
8.     uri = "/doctors/mjones/slots?date=20100104@status=open"/>
9. <link rel = "/linkrels/appointment/updateContactInfo"
10.    uri = "/patients/jsmith/contactInfo"/>
11. <link rel = "/linkrels/help"
12.    uri = "/help/appointment"/>

```

5.1.4 REST-Schnittstelle DoctorService

Dieser Abschnitt zeigt die Umsetzung von *Beispiel 1: Arztpraxis* in actifsource auf. Es werden das Business-Modell und dessen Konfiguration dargestellt und erklärt. Zudem wird der interessante Aspekt dieser REST-Schnittstelle erläutert.

5.1.4.1 REST-Schnittstelle in actifsource

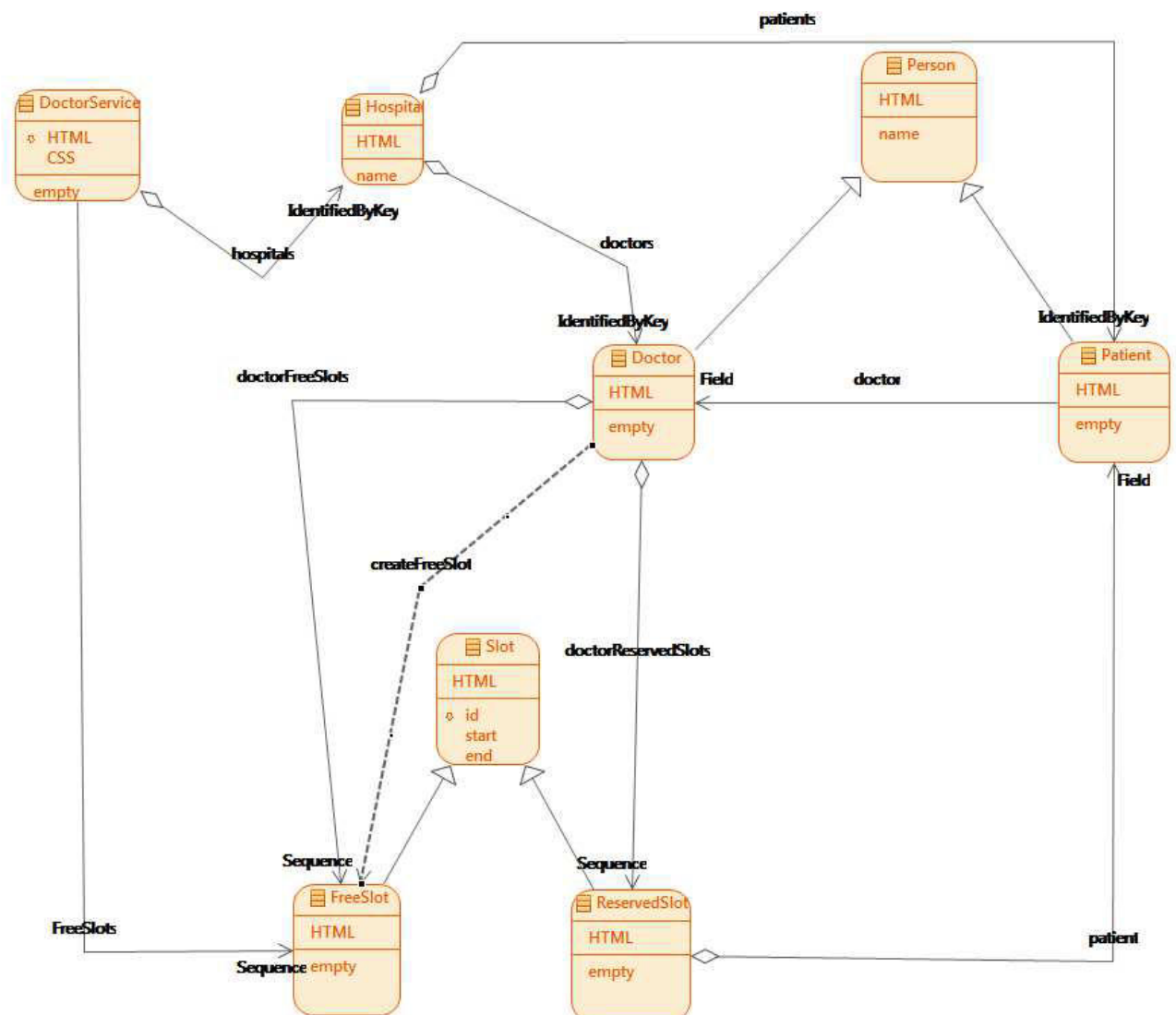


Abbildung 29: Business-Modell der REST-Schnittstelle DoctorService

Die *Abbildung 29* ist die Umsetzung vom *Beispiel 1: Arztpraxis* als REST-Schnittstelle. Die Ressourcen „DoctorService“ und „Hospital“ wurden zusätzlich hinzugefügt und sind in der Beschreibung des Beispiels nicht vorhanden.

Die Ressource „DoctorService“ ist der Einstiegspunkt der Beispielapplikation. „Hospital“ wurde definiert, um nicht direkt vom Applikationseinstieg zu den Doktor- und Patient-Ressourcen zu gelangen. Diese wurden deswegen in einer weiteren Ressource gekapselt.

Wie im Business-Modell ersichtlich, kann ein Anwender mit dem URI-Konzept durch die Ressourcen navigieren. Auf den Ressourcen werden nur URI-Objekte, welche auf ihr mit entsprechenden Links konfiguriert und somit im Modell dargestellt werden, angezeigt. In dieser REST-Schnittstelle sind bis auf „createFreeSlot“, welche ein POST-Request ist, alles GET- oder auch PUT-Requests.

5.1.4.2 Konfiguration der REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service
name	DoctorService
package	ch.hsr
host	localhost
mediaTypeDefault	HTML : HTML
mediaTypeAuxiliary[1]	XML : XML
mediaTypeAuxiliary[2]	CSS : CustomMediaType
entryPoint	ch.hsr.rest.doctorservice.service.DoctorService.DoctorService
resource[1]	DoctorService : Resource
resource[2]	Hospital : Resource
resource[3]	Person : Resource
resource[4]	Patient : Resource
resource[5]	Doctor : Resource
resource[6]	Slot : Resource
resource[7]	FreeSlot : Resource
resource[8]	ReservedSlot : Resource

Abbildung 30: Konfiguration des Business-Modelles der REST-Schnittstelle DoctorService

In der obigen Darstellung der Konfiguration des „DoctorServices“ sind alle relevanten Informationen der REST-Schnittstelle aufgelistet. Speziell an dieser Schnittstellen-Konfiguration ist, dass neben dem Standard-Medientypen „HTML“ noch zwei zusätzliche Medientypen, „XML“ und „CSS“, auf dem Service definiert sind.

5.1.4.3 Interessanter Aspekte zur REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service						
name	DoctorService						
package	ch.hsr						
host	localhost						
mediaTypeDefault	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.HTML</td> </tr> <tr> <td>key</td> <td>text/html</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.HTML	key	text/html		
typeOf	ch.hsr.rest.generic.mediatype.HTML						
key	text/html						
mediaTypeAuxiliary[1]	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.XML</td> </tr> <tr> <td>key</td> <td>application/xml</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.XML	key	application/xml		
typeOf	ch.hsr.rest.generic.mediatype.XML						
key	application/xml						
mediaTypeAuxiliary[2]	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.CustomMediaType</td> </tr> <tr> <td>key</td> <td>text/css</td> </tr> <tr> <td>MediaTypeName</td> <td>CSS</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.CustomMediaType	key	text/css	MediaTypeName	CSS
typeOf	ch.hsr.rest.generic.mediatype.CustomMediaType						
key	text/css						
MediaTypeName	CSS						
entryPoint	ch.hsr.rest.doctorservice.service.DoctorService.DoctorService						

Abbildung 31: Konfiguration der Medientypen vom DoctorService

In der *Abbildung 31* ist ersichtlich, wie die Medientypen für die REST-Schnittstelle DoctorService angegeben werden.

Ein Standard-Medientyp (mediaTypeDefault) muss immer vorhanden sein. Bei key wird der entsprechende Medientyp gemäss IANA³³ angegeben. In diesem Fall ist bei key der Medientyp von „HTML“ (text/html) angegeben.

Des Weiteren können beliebig viele zusätzliche Medientypen hinzugefügt werden. Im konkreten Beispiel vom DoctorService sind dies die Medientypen „XML“ und „CSS“. Diese sind als mediaTypeAuxiliary anzugeben.

Speziell zu erwähnen ist, dass nur HTML und XML als Medientyp vorhanden sind. Wenn ein zusätzlicher Medientyp benötigt wird, muss er auf dieselbe Weise wie der Typ „CSS“ erfasst werden. Der zu wählende Medientyp ist „CustomMediaType“. Bei key wird, gleich wie bei „HTML“ oder „XML“ der Medientyp nach IANA angegeben. Zusätzlich muss nun unter „MediaTypeName“ der gewünschte Name des Medientypen hinzugefügt werden. In diesem Fall ist dies „CSS“.

Die beschriebene Vorgehensweise um die Medientypen einer REST-Schnittstelle zu konfigurieren ist für alle weiteren Beispiele dieselbe.

Eine Liste von Medientypen ist auf der Seite von IANA³⁴ zu finden.

³³ [IANA]

³⁴ [IANA]

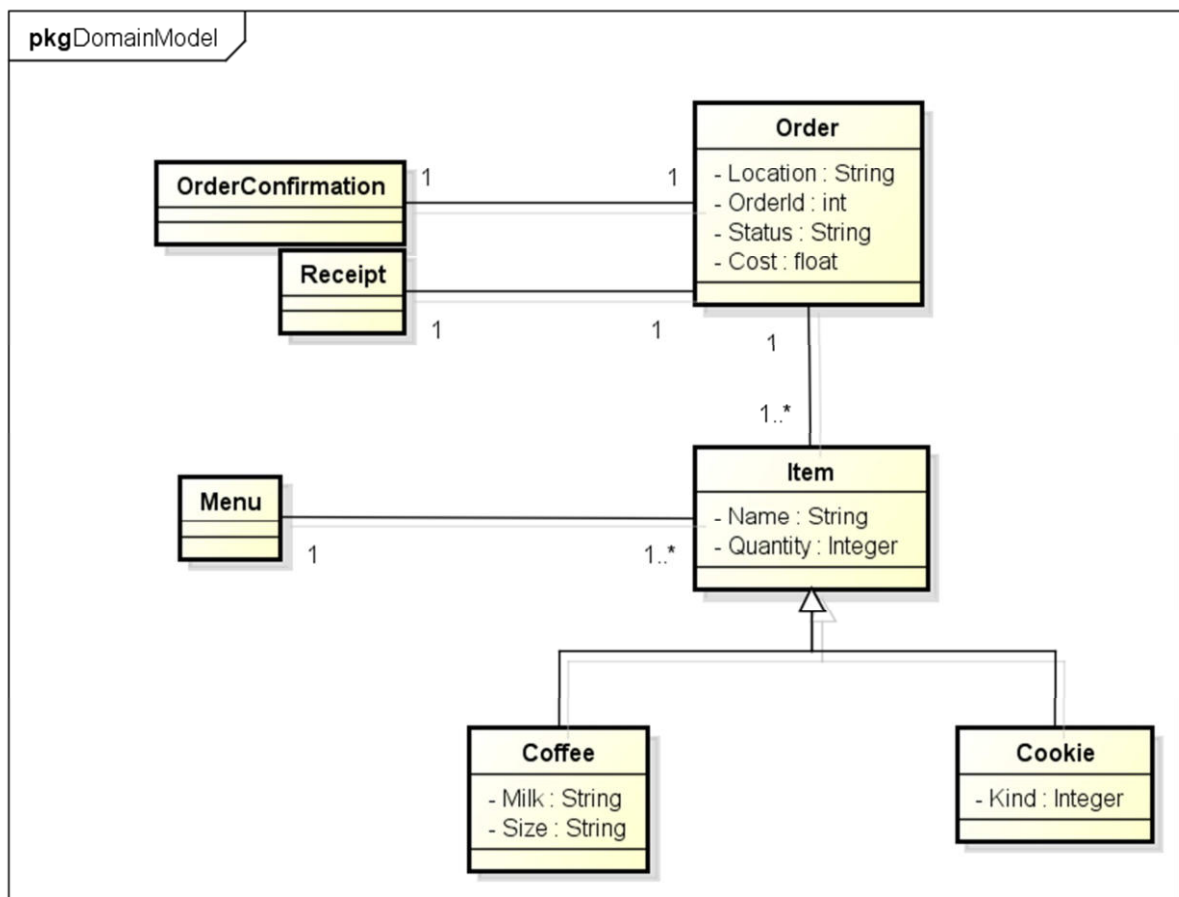
5.2 Beispiel 2: RestBucks

5.2.1 Einleitung

Dieses Beispiel stammt aus dem Buch „REST in Practice“³⁵. Es beschreibt einen Kaffee-Kauf-Prozess. Der Kunde kommt in den Laden und bestellt am Tresen einen Kaffee. Kaffees gibt es in verschiedenen Grössen und mit oder ohne Milch. Neben Kaffee können noch andere Produkte erworben werden, aber nur solche, die auch auf der Karte (Menu) stehen. Die Bestellung kann solange geändert werden, wie die Zahlung nicht erfolgt ist. Nach dem Bezahlvorgang, wird die Bestellung bearbeitet. Sobald der Zubereitungsprozess beendet ist, kann der Kunde seinen Kaffee abholen und bekommt eine Quittung.

5.2.2 Ausgangslage

5.2.2.1 Domain Modell

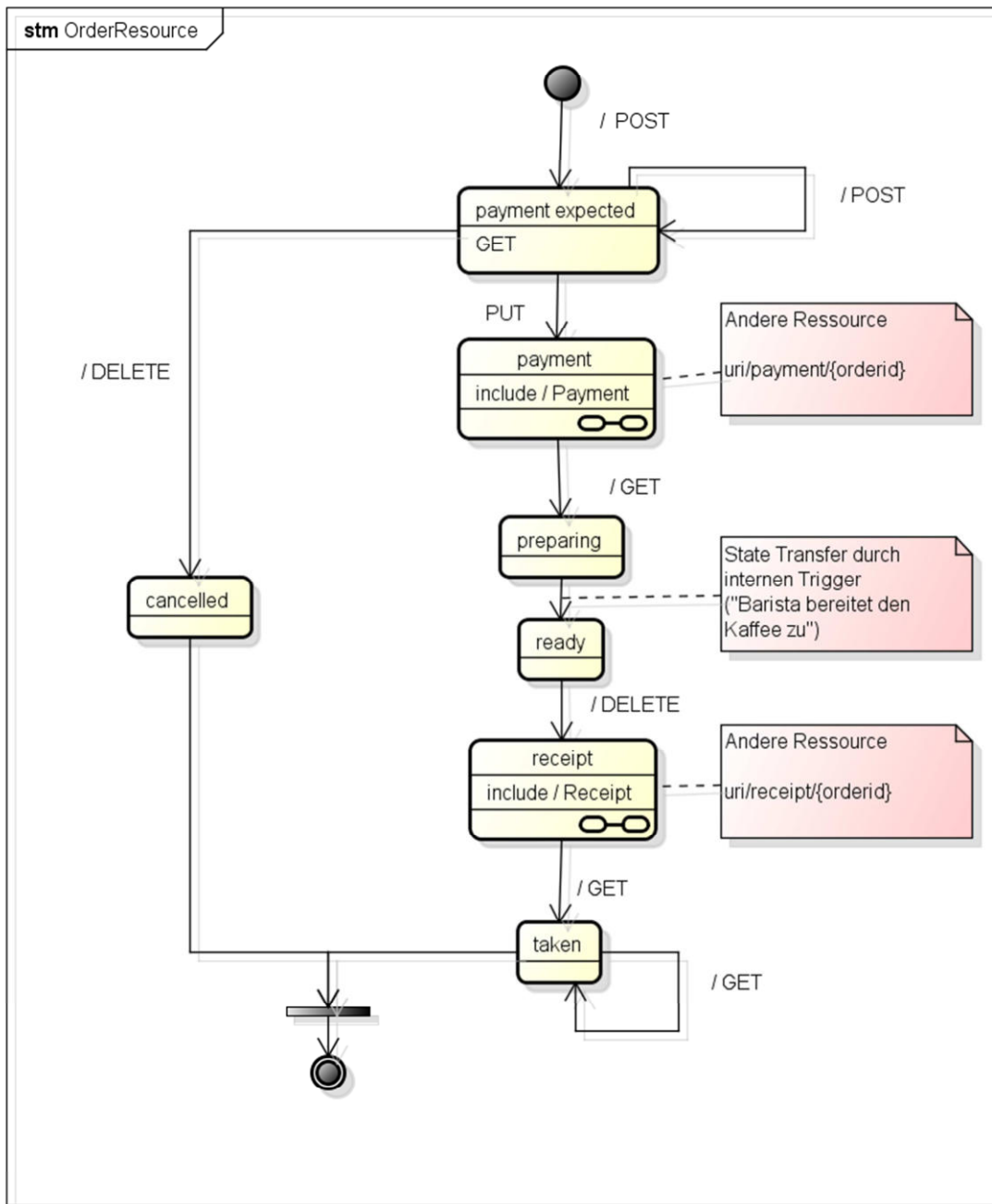


powered by Astah

Abbildung 32: UML-Domain Model RestBucks

³⁵ [RESTIP]

5.2.2.2 State Diagramm



powered by Astah

Abbildung 33: UML-State-Diagramm der Bestellung "Order"

5.2.3 Erkenntnisse

5.2.3.1 REST-Komponenten

Ressourcen

Ressource	Bemerkungen
Order	Interner Zustand der Ressource kann über POST/PUT Aufrufe auf andere Ressourcen geändert werden.
Menu	Reine GET Ressource
Payment	Ressource mit eigenem Status
Receipt	Nur über DELETE aufrufbar.

HTTP-Methoden

Verb	Beschreibung
GET	Alle lesenden Anfragen auf eine Ressource.
POST	Erstellen einer neuen Zahlung, Erstellen einer neuen Quittung
PUT	Alle schreibenden Anfragen auf eine bestehende Ressource. Als Beispiel: „Order“, um Item erweitern / reduzieren.
DELETE	DELETE wird in diesem Beispiel zum Beenden eines Bestellvorgangs verwendet (Sowohl beim erfolgreichen Vorgang, als auch als Beendigung vor dem Zahlvorgang.).

Medientypen³⁶

Media-Type	Beschreibung
application/xml; text/xml	Ober-Typ für XML-Content
Application/vnd.restbucks+xml	Eigener Medientyp der RestBucks Services basierend auf XML
Application/json	Ober-Typ für JSON-Content
Application/vnd.restbucks+json	Eigener Medientyp der RestBucks Services basierend auf JSON

URI / Adressen

- <http://restbucks.com/order/{orderId}>
- <http://restbucks.com/payment/{orderId}>
- <http://restbucks.com/receipt/{orderId}>

5.2.3.2 HATEOAS / REST LEVEL 3

Die Führung des Clients durch die Applikation zeigt sich im State-Diagramm (Kapitel 5.2.2.2 *State Diagramm*). Alle Ausgehenden Pfeile einer Ressource sind dem Client als Links bekannt zu machen.

5.2.3.3 Bemerkungen

Zugriff auf die Ressourcen „Payment“ und „Receipt“ ändern den internen Zustand der Ressource „Order“.

Es fehlt noch der optionale Punkt der „Coffee Card“. Es handelt sich dabei um eine für den Client optionale Erweiterung. Deshalb befindet sich diese nicht in der Ausgangslage.

³⁶ [IANA]

Speziell ist, dass mit einem DELETE Request auf ein Receipt der Bestellvorgang beendet wird. Die Ressource „Receipt“ wird demnach entweder nie erstellt, oder wenn, dann in der Applikationslogik.

5.2.4 REST-Schnittstelle RestBucks

Im folgenden Abschnitt wird das *Beispiel 2: RestBucks*, wie es als REST-Schnittstelle in actifsource eingepflegt ist, beschrieben. Des Weiteren wird die Konfiguration der REST-Schnittstelle kurz erklärt und der interessante Aspekt von der REST-Schnittstelle RestBucks aufgezeigt.

5.2.4.1 REST-Schnittstelle in actifsource

Nachfolgend in der *Abbildung 34* ist die REST-Schnittstelle von RestBucks ersichtlich, wie es in actifsource modelliert ist. An sich entspricht dieses Modell, dem Domain Modell der Ausgangslage vom *Beispiel 2: RestBucks*. Zusätzlich kommen die Ressource-Subtypen der Ressource Order hinzu. Diese stellen den Fortschritt der Bestellung als Ressourcen dar.

Es ist zu erwähnen, dass alle gestrichelten Links (ActivityLinks) mit Namen „backToRestbucks“, GET-Requests auf den statischen URI der Ressource „RestBucks“ sind. Die anderen ActivityLinks sind POST-Requests, um die entsprechende Ressource zu erstellen.

Die vorhandenen, durchgezogenen Links, sind entweder CompositionLinks oder AssociationLinks. Auf diesen sind GET- und PUT-Requests definiert. Diese Links bilden sich auf eine spezifische Ressource ab.

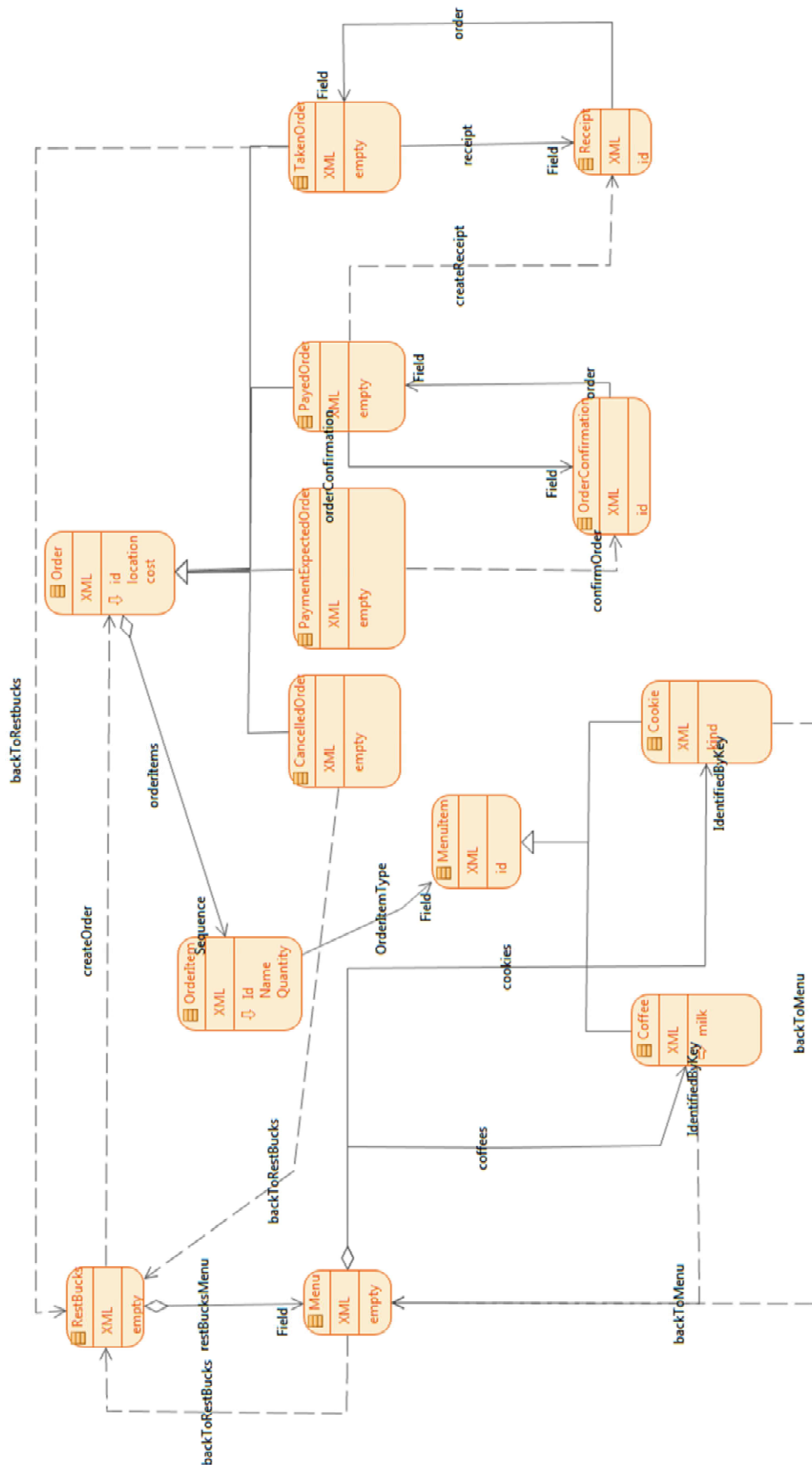


Abbildung 34: Business-Modell der REST-Schnittstelle RestBucks in der activesource-Umgebung

5.2.4.2 Konfiguration der REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service
name	RestBucks
package	ch.hsr
host	localhost
mediaTypeDefault	XML : XML
mediaTypeAuxiliary	
entryPoint	ch.hsr.restbucks.service.RestBucks.RestBucks
resource[1]	Menu : Resource
resource[2]	OrderItem : Resource
resource[3]	Order : Resource
resource[4]	PaymentExpectedOrder : Resource
resource[5]	PayedOrder : Resource
resource[6]	TakenOrder : Resource
resource[7]	CancelledOrder : Resource
resource[8]	Coffee : Resource
resource[9]	Cookie : Resource
resource[10]	MenuItem : Resource
resource[11]	RestBucks : Resource
resource[12]	OrderConfirmation : Resource
resource[13]	Receipt : Resource

Abbildung 35: Konfiguration der REST-Schnittstelle RestBucks in der actifsource-Umgebung

In der *Abbildung 35* ist die komplette Konfiguration des Services „RestBucks“ abgebildet. Die REST-Schnittstelle besitzt nur einen Standard-Medientyp. Dieser ist im Minimum erforderlich für eine korrekte Einstellung der REST-Schnittstellen.

5.2.4.3 Interessanter Aspekt der REST-Schnittstelle

Die *Abbildung 36* zeigt, wie mit den Ressourcen in einer REST-Schnittstelle dafür gesorgt werden kann, dass ein bestimmter Ablauf eingehalten werden muss.

In diesem Beispiel wird, sobald eine neue „Order“ erstellt werden soll, eine „PaymentExpectedOrder“ erzeugt. Die Ressource „Order“ selbst kann nicht angelegt werden.

Der Anwender bestätigt die Bestellung, indem er durch einen POST-Request eine neue „OrderConfirmation“ Ressource anlegt. Dazu benutzt er den einzigen, ihm sichtbaren Link. Daraufhin erstellt der „Controller“ der Ressource „OrderConfirmation“ eine „PayedOrder“. Von diesem Zeitpunkt an ist es nicht mehr möglich zu der Ressource „PaymentExpectedOrder“ zu gelangen. Dieses Schema wird bis zur letzten Bestell-Ressource „TakenOrder“ fortgesetzt.

Bei dieser Ressource angelangt, gibt es nur noch die folgenden Optionen. Entweder die Ressource „Receipt“ nochmals anzuzeigen oder zurück zu kehren zur Start-Ressource „RestBucks“.

Das Konzept des zuvor beschriebenen Aspektes, ist in den folgenden Beispielapplikationen wieder zu finden.

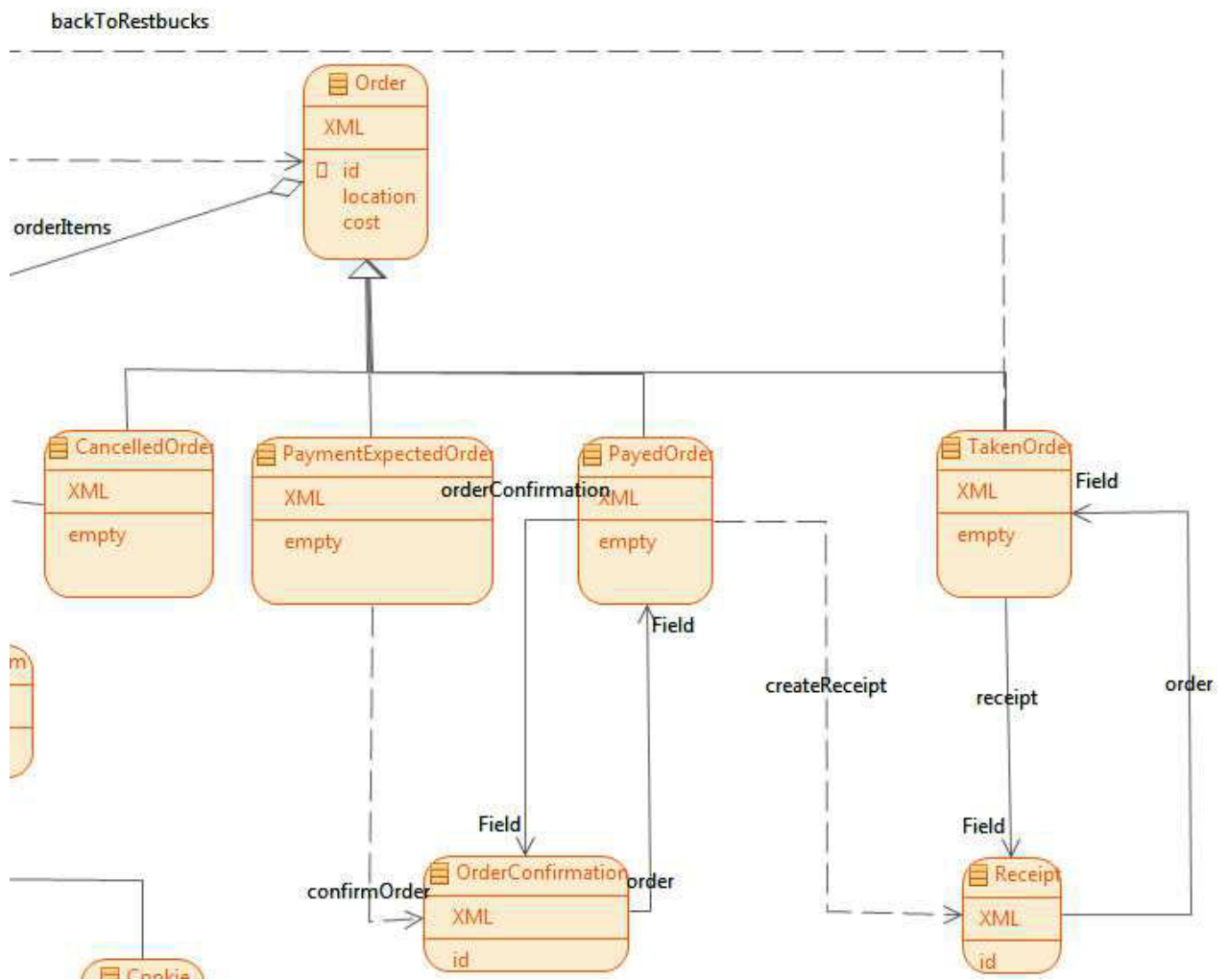


Abbildung 36: Ressource Order deren Subressourcen (Ausschnitt aus *Abbildung 34*)

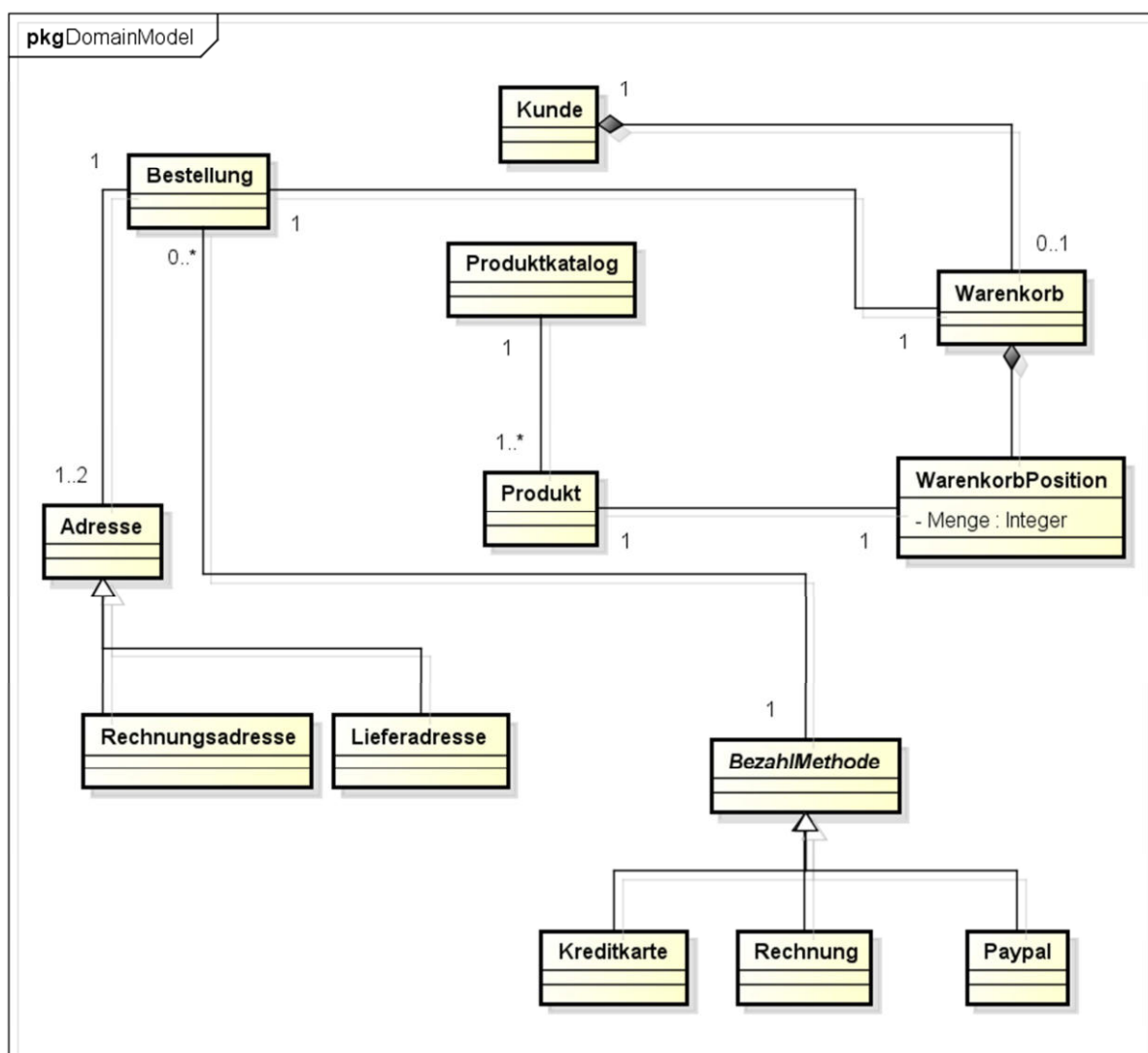
5.3 Beispiel 3: E-Commerce

5.3.1 Einleitung

Dieses Beispiel beschreibt einen Standard-Web-Shop. Ein Kunde kann Produkte, aus einem Produkte-Katalog, in einen Warenkorb legen. Wenn er alle seine gewünschten Produkte in den Warenkorb gelegt hat, kann er diese Produkte kaufen. Dabei kann er die Liefer- und, wenn sie abweicht, die Rechnungs-Adresse festlegen. Zusätzlich kann er aus mehreren unterschiedlichen Bezahl-Methoden wählen.

5.3.2 Ausgangslage

5.3.2.1 Domain Modell



powered by Astah

Abbildung 37: Domain-Model der E-Commerce Anwendung

5.3.3 Erkenntnisse

5.3.3.1 REST-Komponenten

Ressourcen

Ressource	Bemerkungen
Kunde	
Warenkorb	GET Abfrage für den aktuellen Zustand. POST zum Anlegen eines neuen Warenkorbes. PUT zum Hinzufügen neuer Positionen. DELETE für Check-Out
WarenkorbPosition	POST zum Erstellen, DELETE zum Entfernen, PUT zum Ändern der Anzahl
Produktekatalog	GET für Infos. Immutable für den Client
Produkt	GET für Infos. Immutable für den Client
Bestellung	Kein eigenständiges Erstellen der Ressource, sondern über Checkoutfunktionalität des Warenkorbes
Adresse	Allenfalls Unterressource des Kunden.
Bezahlmethode	Hier sind allenfalls folgende Links notwendig: Links auf externe Anbieter mit einem Callback.

HTTP-Methoden

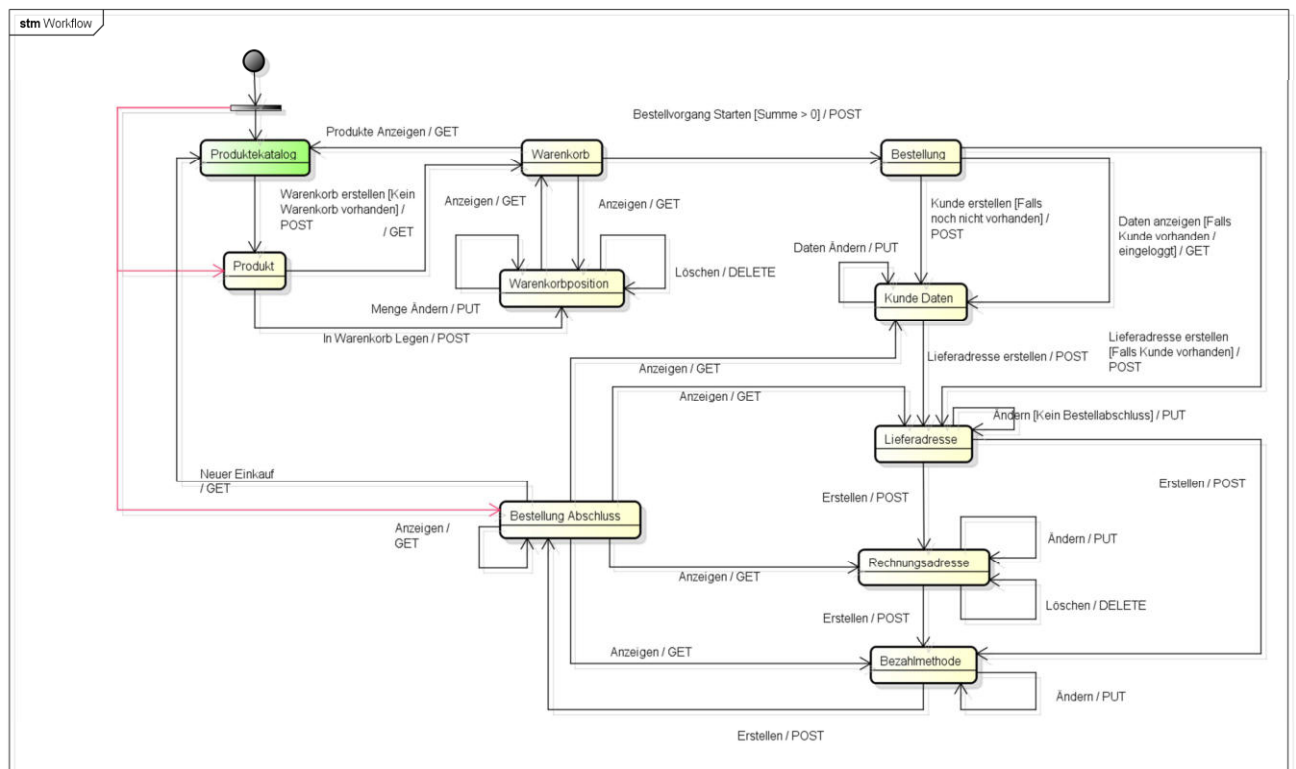
Verb	Beschreibung
GET	Alle lesenden Anfragen auf eine Ressource.
POST	Für das Erstellen neuer Warenkorbpositionen
PUT	Menge Ändern bei Warenkorbpositionen
DELETE	Löschen von Warenkorbpositionen

Medientypen³⁷

Media-Type	Beschreibung
application/xml; text/xml	Ober-Typ für XML-Content

³⁷ [IANA]

5.3.3.2 HATEOAS / REST LEVEL 3



powered by Astah

Abbildung 38: Workflow E-Commerce Applikation als UML-State-Diagramm

Wiederum sind die roten Übergänge, solche, welche direkt aufgerufen werden können. Der Produkt-Katalog (Grün) markiert den Einstiegspunkt. Von da aus wird der Client mittels Links (Ausgehende Pfeile einer Ressource) durch den Workflow geführt.

5.3.3.3 Bemerkungen

Die Bezahlmethoden müssen nicht selber angeboten werden, sondern können auch externe Dienstleister sein (z.B. PayPal).

Gewisse Produkte können nicht per Rechnung bezahlt werden (z.B. E-Books).

5.3.4 REST-Schnittstelle E-Commerce

Dieser Abschnitt beschreibt die REST-Schnittstelle E-Commerce und basiert auf dem *Beispiel 3: E-Commerce*. Im Folgenden wird das Business-Modell, deren Konfiguration und ein interessanter Aspekt der REST-Schnittstelle E-Commerce beschrieben.

5.3.4.1 REST-Schnittstelle in actifsource

In der *Abbildung 39* zeigt die REST-Schnittstelle, wie sie in actifsource modelliert ist. Das Business-Modell von E-Commerce enthält Parallelen zu dem in der REST-Schnittstelle RestBucks, beschriebenen *Interessanter Aspekt* der REST-Schnittstelle Bei E-Commerce wird dieses Verfahren hingegen in grösserem Masse angewendet.

Gewisse „Order“-Ressourcen können im Anwendungsverlauf übersprungen werden. Dies ist bei den spezifischen „Order“-Ressourcen der Zahlungsvarianten oder Adresstypen der Fall. Ein Benutzer muss beispielsweise nicht beide Adressvarianten „ShippingAddress“ und „BillingAddress“ erstellen. Lediglich die Versandadresse wird benötigt. Danach ist es ihm offen, eine Rechnungsadresse zu erfassen oder direkt eine der drei möglichen Zahlungsmethoden, Kreditkarte, Rechnung oder PayPal, zu wählen und dadurch dessen Ressource erstellen.

Das Verhalten der dargestellten Links im Modell ist, dasselbe, wie schon in den vorhergehenden Beispielapplikationen.

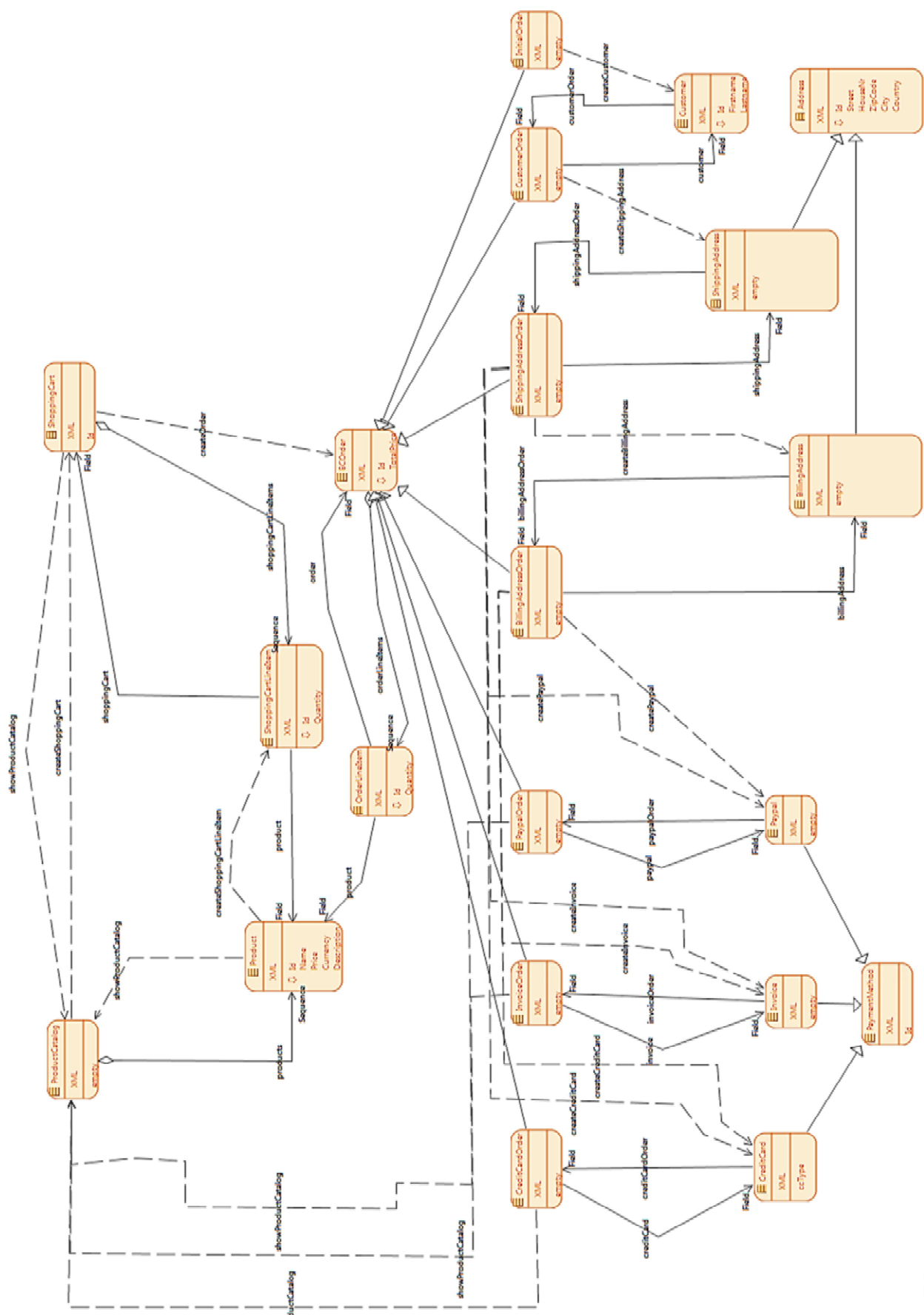


Abbildung 39: Business-Modell der REST-Schnittstelle E-Commerce

5.3.4.2 Konfiguration der REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service
name	ECommerce
package	ch.hsr
host	localhost
mediaTypeDefault	XML : XML
mediaTypeAuxiliary	
entryPoint	ch.hsr.ecommerce.service.ECommerce.ProductCatalog
resource[1]	ProductCatalog : Resource
resource[2]	Product : Resource
resource[3]	ShoppingCart : Resource
resource[4]	ShoppingCartLineItem : Resource
resource[5]	Customer : Resource
resource[6]	Address : Resource
resource[7]	BillingAddress : Resource
resource[8]	ShippingAddress : Resource
resource[9]	PaymentMethod : Resource
resource[10]	CreditCard : Resource
resource[11]	Invoice : Resource
resource[12]	Paypal : Resource
resource[13]	ECOrder : Resource
resource[14]	InitialOrder : Resource
resource[15]	CustomerOrder : Resource
resource[16]	ShippingAddressOrder : Resource
resource[17]	BillingAddressOrder : Resource
resource[18]	OrderLineItem : Resource
resource[19]	InvoiceOrder : Resource
resource[20]	PaypalOrder : Resource
resource[21]	CreditCardOrder : Resource

Abbildung 40: Konfiguration der REST-Schnittstelle E-Commerce

In der Konfiguration der REST-Schnittstelle E-Commerce ist der Anstieg der Anzahl von Ressourcen gut zu erkennen.

Der Standard-Medientyp ist „XML“. Dieser wurde gewählt, da es sich um die Server-Schnittstelle handelt. Eine Client-Schnittstelle, würde die Benutzereingaben des Web-Frontend im „XML“-Format an die REST-Schnittstelle übermitteln.

Dieses Vorgehen ist ein interessanter Aspekt dieser REST-Schnittstelle.

5.3.4.3 Interessanter Aspekt der REST-Schnittstelle

Um Ressourcen zu erstellen, muss eine Client-, die Server-REST-Schnittstelle mit dem erwarteten Aufbau der Eingabe im XML-Format beliefern. Die Eingabe wird mittels „POST“-Methode übermittelt.

In dieser Beispielapplikation existieren acht Ressourcen, welche eine solche Service-Eingabe benötigen, um erstellt zu werden. Möglich ist auch eine Erstellung von Ressourcen ohne jeglichen Zusatzinformationen.

5 Anwendung der Konzepte auf Applikationsbeispiele

Die letztere Variante wird beim Erzeugen der „ShoppingCart“-Ressource verwendet. Für diese würde keine XML-Eingabe benötigt.

Die acht E-Commerce-Ressourcen, werden im Anschluss mit der jeweilig benötigten Eingabe im XML-Format aufgelistet. Dabei ist anzumerken, dass die Struktur der Schnittstellen-Eingabe wichtig ist. Ansonsten schlägt die Erzeugung der entsprechenden Ressource fehl und ein entsprechender HTTP-Status-Code wird zurückgegeben. Die Client-Schnittstelle verbleibt dabei in der E-Commerce-Ressource, von der aus die Erstellung der Zielressource veranlasst wurde. Die enthaltenen Beispieldaten der Eingabebeispiele können beliebig variieren.

Die acht E-Commerce-Ressourcen und ihre benötigten *XML-EINGABEN* zur Erzeugung:

Ressource: „ShoppingCartLineItem“

XML-Eingabe:

```
<scItem>
  <quantity>2</quantity>
  <product>1</product>
  <shoppingCart>1</shoppingCart>
</scItem>
```

Ressource: „InitialOrder“

XML-Eingabe:

```
<order>
  <shoppingCart>1</shoppingCart>
</order>
```

Anmerkung: Bei der Erzeugung der „InitialOrder“-Ressource ist der umschliessende XML-Tag nicht wie die, zu erstellende, Ressource benannt. Der Grund dafür ist, dass eine „Order“ erzeugt wird. Die entsprechenden spezifischen „Order“-Ressourcen ändern nur den Typ der Bestellung und fügen ihre erfassten Daten beim Abspeichern hinzu. Die spezifischen Order-Ressourcen besitzen nur Zugriff auf die von ihnen im Erfassungsschritt verlangten Daten.

Ressource: „Customer“

XML-Eingabe:

```
<customer>
  <firstname>Hans</firstname>
  <lastname>Muster</lastname>
  <orderId>1</orderId>
</customer>
```

Ressource: „ShippingAddress“

XML-Eingabe:

```
<shippingAddress>
  <street>Musterstrasse</street>
  <housenr>17</housenr>
  <zipcode>895317</zipcode>
  <city>MusterCity</city>
  <country>Musterland</country>
  <orderId>1</orderId>
</shippingAddress>
```

Ressource: „BillinAddress“

Anmerkung: XML-EINGABE ist die Selbe wie bei „ShippingAddress“, ausser, dass der umschliessende XML-Tag anstatt <shippingAddress>, <billingAddress> lautet.

Ressource: „CreditCard“

XML-Eingabe Variante 1:

```
<creditCard>
  <ccType>MasterCard</ccType>
  <prevOrderType>shippingAddressOrder</prevOrderType>
  <orderId>1</orderId>
</creditCard>
```

Anmerkung: Die XML-EINGABE VARIANTE 2 dieselbe wie bei der XML-EINGABE VARIANTE 1, jedoch der XML-Tag <prevOrderType> hat den Inhalt „billingAddressOrder“ anstelle von „shippingAddressOrder“.

Ressource: „Invoice“

XML-Eingabe Variante 1:

```
<invoice>
  <prevOrderType>shippingAddressOrder</prevOrderType>
  <orderId>1</orderId>
</invoice>
```

Anmerkung: Die *XML-EINGABE VARIANTE 2* dieselbe wie bei der *XML-EINGABE VARIANTE 1*, jedoch der XML-Tag **<prevOrderType>** hat den Inhalt „billingAddressOrder“ anstatt „shippingAddressOrder“.

Ressource: „Paypal“

Anmerkung: *XML-EINGABE VARIANTE 1 / 2* ist dieselbe wie bei „Invoice“, ausser, dass der umschliessende XML-Tag anstatt **<invoice>**, **<paypal>** lautet.

Diese Art von Eingabe gilt in allen REST-Schnittstellen für die POST-Methoden. Bei den PUT-Methoden würde das gleiche Verhalten gelten, jeweils ohne Angabe der IDs. Denn diese liegen den Ressourcen bereits vor. Es wurde jedoch in keiner der Beispielapplikation implementiert.

5.4 Beispiel 4: Projekt Verrechnung

5.4.1 Einleitung

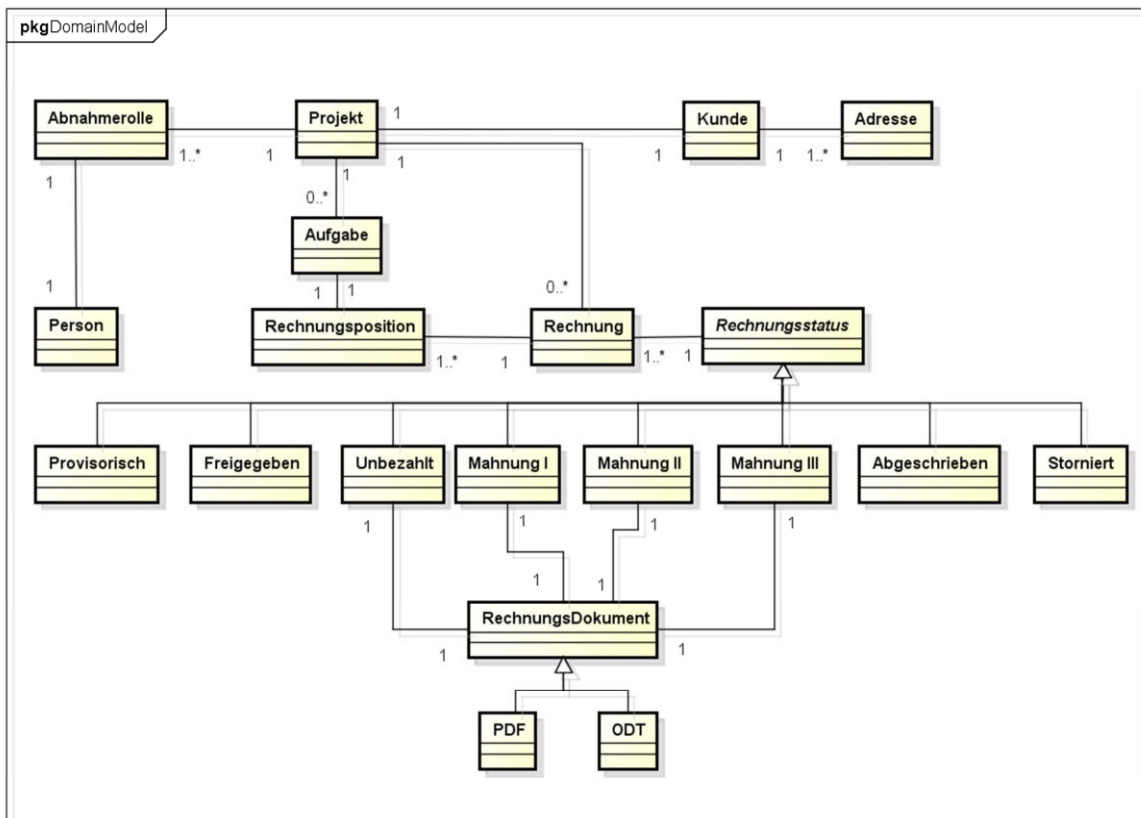
Dieses Beispiel beschreibt den Vorgang, in reduzierter Form, einer Rechnungsstellung, wie sie in der Projektmanagementsoftware todayu³⁸ implementiert ist.

Projekte können abgerechnet werden, wobei ein neuer Rechnungsdatensatz mit Status „Provisorisch“ erstellt wird. Im Projekt können Personen definiert werden, welche diese Rechnung freigeben müssen. Sobald alle diese Personen die Rechnung freigegeben haben, wechselt die Rechnung den Status. Nun kann die Rechnung von einer beliebigen Person mit weiteren Meta-Daten versehen werden. Falls die Rechnung nach einer bestimmten Anzahl Tagen nicht bezahlt ist, wechselt sie in den Status Mahnung. Dies kann bis zu 3 Mal geschehen. Die Endzustände sind: Abgeschrieben, Bezahlt oder Storniert. Wobei der Status „Storniert“, eine neue provisorische Rechnung zum Projekt generiert.

Es gibt einige Rechnungsstatus, zu welchen ein Dokument erstellt wird. Dies kann in Form eines OpenDocument oder PDF geschehen. Weitere Formate sind denkbar.

5.4.2 Ausgangslage

5.4.2.1 Domain Modell

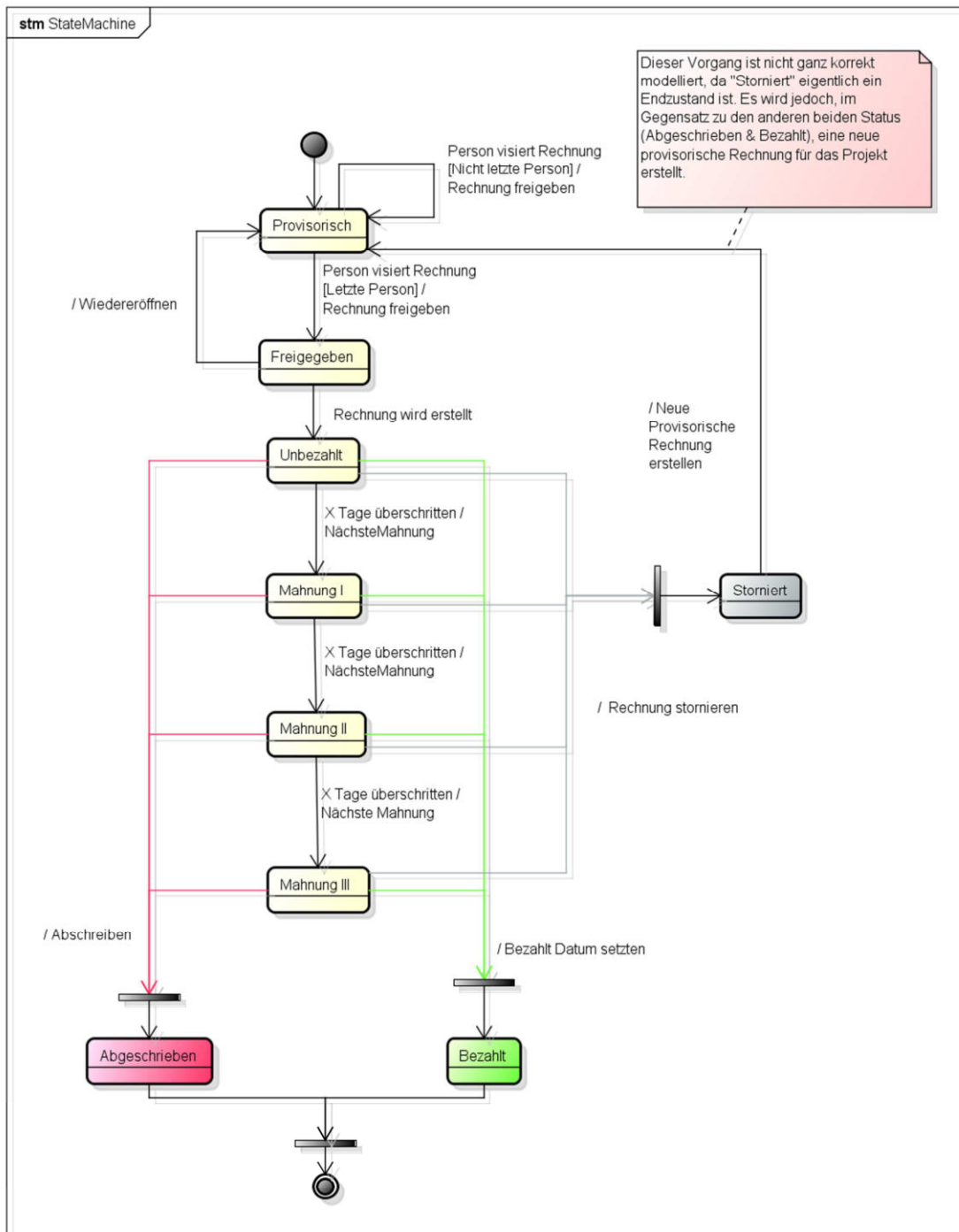


powered by Astah

Abbildung 41: UML-Domain-Model für Rechnungsstellung

³⁸ [todayu]

5.4.2.2 State Diagramm Rechnung



powered by Astah

Abbildung 42: UML-State-Diagramm einer Rechnung

5.4.2.3 State Diagramm Projekt

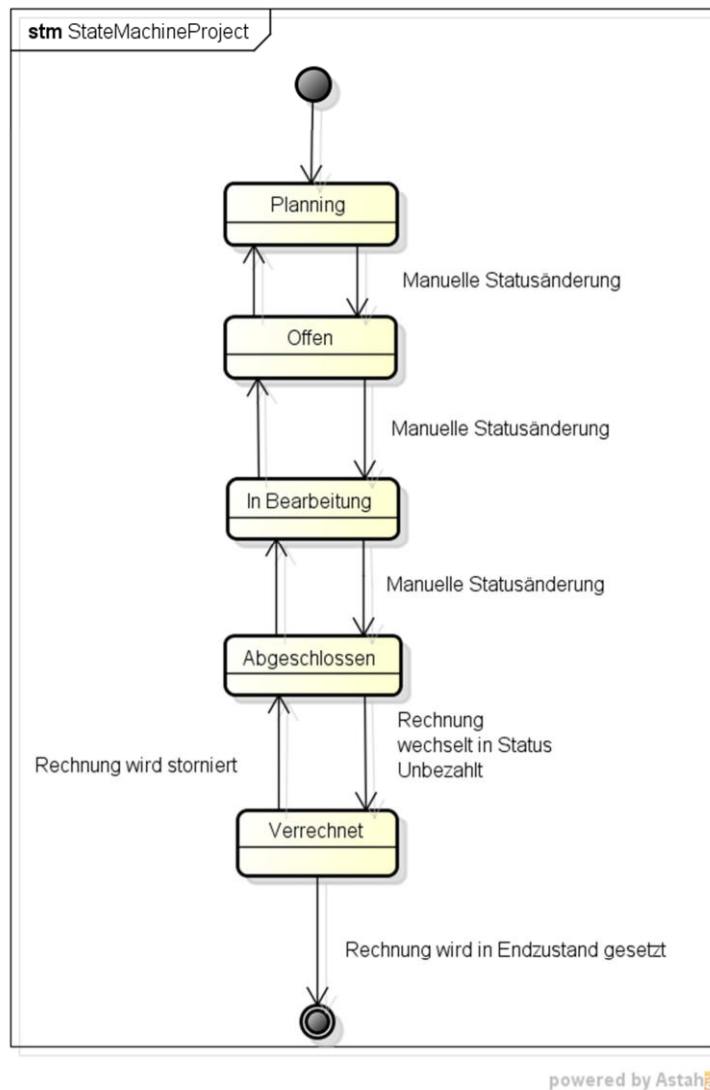


Abbildung 43: UML-State-Diagramm von Projekt

5.4.3 Erkenntnisse

5.4.3.1 REST-Komponenten

Ressourcen

Ressource	Bemerkungen
Rechnung	-
Projekt	-
Aufgaben	Aufgaben sind Bestandteile eines Projektes. Aus einer Aufgabe entsteht später eine Rechnungsposition.
Kunde	Änderungen an dieser Ressource sind nicht vorgesehen. Zugriff über GET ist ausreichend.
Adresse	Änderungen an dieser Ressource sind nicht vorgesehen. Zugriff über GET ist ausreichend.
Rechnungsdokument	Das Rechnungsdokument ist allenfalls keine eigene Ressource, sondern ist eine Repräsentation (pdf / opendocument) der Ressource Rechnung.

HTTP-Methoden

Verb	Beschreibung
GET	Alle lesenden Anfragen auf eine Ressource.
POST	Erstellen einer neuen Rechnung, Erstellen eines neuen Projektes, Erstellen eines neuen Kunden
PUT	Alle schreibenden Anfragen auf eine bestehende Ressource. Als Beispiel: Visieren der Rechnung, das Setzen des Status, Bezahlung einer Rechnung
DELETE	Stornieren einer Rechnung oder Abschreiben einer Rechnung

Medientypen³⁹

Media-Type	Beschreibung
application/vnd.oasis.opendocument.text	Medientyp für OpenDocument
application/pdf	Medientyp für PDF-Dateien
text/html	HTML-Ausgabe (für View)
application/xml; text/xml	Ober-Typ für XML-Content
vnd.hsr.invoicing+xml	Eigener Medientyp: spezialisierter Medientyp für den Rechnungsstellungs-Service

URI / Adressen

Als Basis dient die URI des Services, respektive die Einstiegs-URI. Von hier aus werden alle URIs als Link mitgeliefert.

http://wheretofindthisservice.com/

Jede Ressource bekommt ihre eigene Adresse:

http://wheretofindthisservice.com/{Ressourcenname}

Bestehende Ressource können über eine ID direkt identifiziert werden:

http://wheretofindthisservice.com/{Ressourcenname}/{Ressourcen-ID}

³⁹ [IANA]

5.4.3.2 HATEOAS / REST LEVEL 3

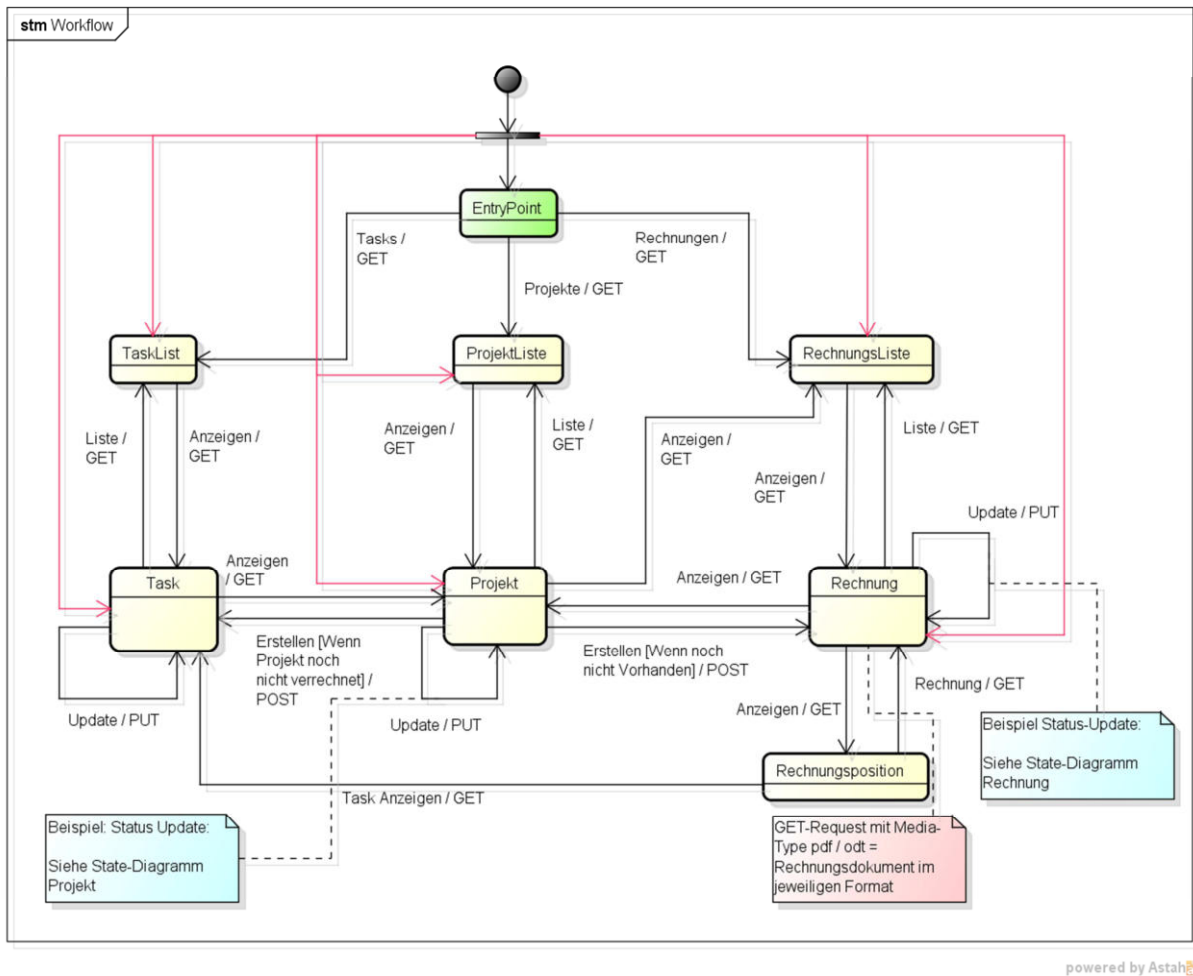


Abbildung 44: Workflow Verrechnung

5.4.3.3 Bemerkungen

Bei den Formaten sind Erweiterungen möglich. Dies würde zur Folge haben, dass neue Medientypen und neue Links entstehen würden.

Eine weitere Möglichkeit ist die Einführung eines neuen Rechnungsstatus.

Die Links, welche dem Client geliefert werden, sind stark abhängig vom internen Status der Ressource „Rechnung“. Die Links sind demnach nicht ausschliesslich Übergänge von einer Ressource in eine Andere.

Es gibt zwei primäre Ressourcen: Rechnung und Projekt. Der Status von Rechnungen beeinflusst denjenigen von Projekten und umgekehrt.

5.4.4 REST-Schnittstelle Accounting

Der Abschnitt 5.4.4 befasst sich mit der REST-Schnittstelle Accounting.

5.4.4.1 *REST-Schnittstelle in actifsource*

Die REST-Schnittstelle Accounting modelliert das *Beispiel 4: Projekt Verrechnung*. Es ist die komplexeste Beispielapplikation, welche im Rahmen der Bachelorarbeit in actifsource modelliert wurde. Das Beispiel ist jedoch nur als Business-Modell grafisch modelliert.

Business-Logik wurde keine implementiert. Diese wäre ähnlich wie diejenige der *REST-Schnittstelle E-Commerce*. Accounting modelliert mit den Ressourcen und Subressourcen von „Project“ und „Invoice“ das beschriebene Konzept aus Kapitel 5.2.4.3 gleich doppelt.

Im Business-Modell von Accounting sind die bisher beschriebenen Konzepte gut ersichtlich. Zum Beispiel das Verhalten der *REST-Schnittstelle E-Commerce* beim Umgang mit der „Order“-Ressource. In diesem Fall ist dieses Verhalten beim Erstellen der „TemporaryInvoice“-Ressource von der „FinishedProject“-Ressource aus ersichtlich. Ebenso kann dies beim Umgang mit den „InvoiceDocument“-Ressourcen beobachtet werden.

Die *Abbildung 45* zeigt den aktuellen Stand der *REST-Schnittstelle Accounting*. Der Erfahrung nach kann sich das Modell während der Implementation leicht ändern.

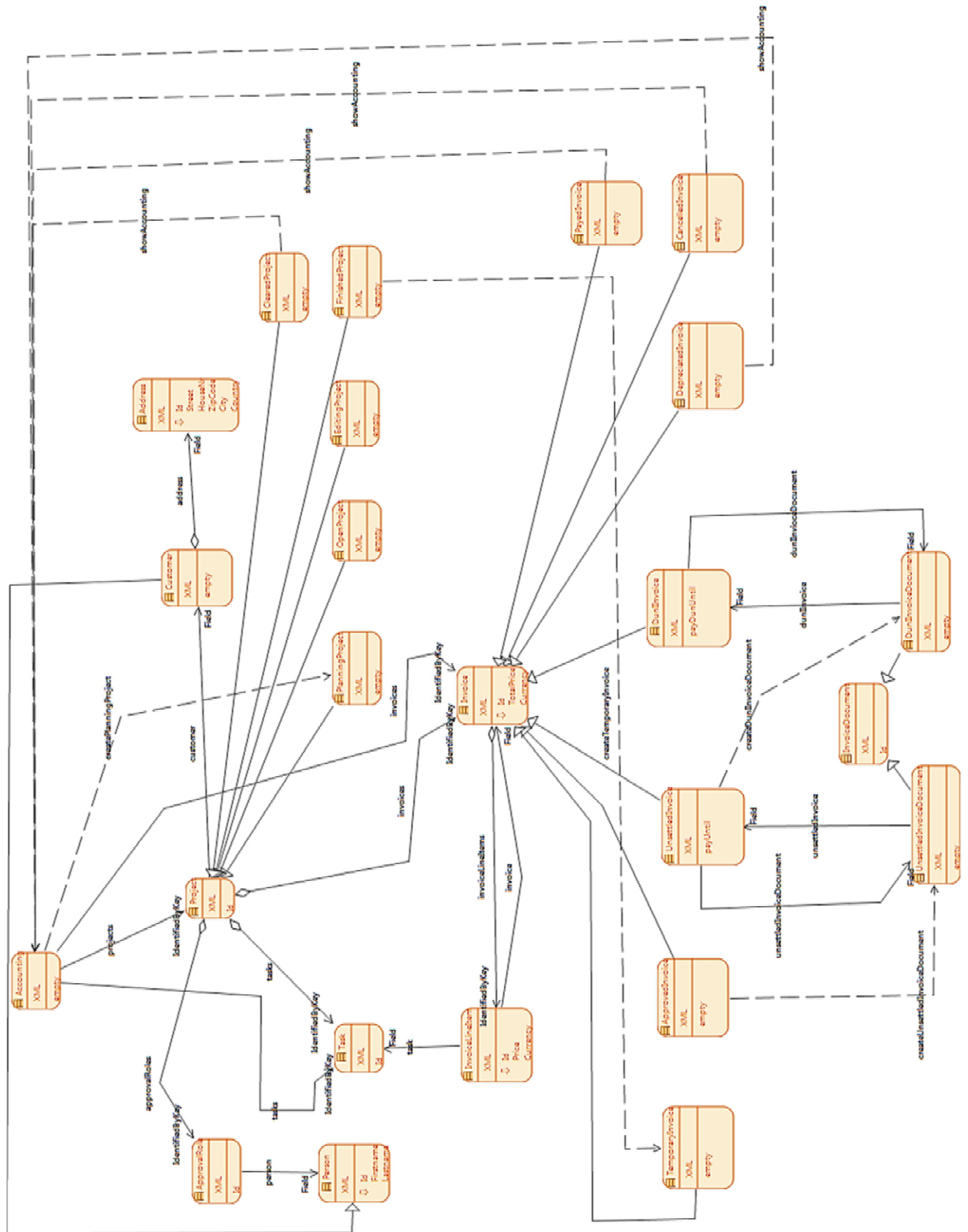


Abbildung 45: Business-Modell der REST-Schnittstelle Accounting

5.4.4.2 Konfiguration der REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service
name	Accounting
package	ch.hsr
host	localhost
mediaTypeDefault	XML : XML
mediaTypeAuxiliary	
entryPoint	ch.hsr.accounting.service.Accounting.Accounting
resource[1]	Person : Resource
resource[2]	ApprovalRole : Resource
resource[3]	Project : Resource
resource[4]	Customer : Resource
resource[5]	Address : Resource
resource[6]	Task : Resource
resource[7]	InvoiceLineItem : Resource
resource[8]	Invoice : Resource
resource[9]	TemporaryInvoice : Resource
resource[10]	ApprovedInvoice : Resource
resource[11]	UnsettledInvoice : Resource
resource[12]	DunInvoice : Resource
resource[13]	DepreciatedInvoice : Resource
resource[14]	CancelledInvoice : Resource
resource[15]	InvoiceDocument : Resource
resource[16]	Accounting : Resource
resource[17]	PayedInvoice : Resource
resource[18]	PlanningProject : Resource
resource[19]	OpenProject : Resource
resource[20]	EditingProject : Resource
resource[21]	FinishedProject : Resource
resource[22]	ClearedProject : Resource
resource[23]	UnsettledInvoiceDocument : Resource
resource[24]	DunInvoiceDocument : Resource

Abbildung 46: Konfiguration der REST-Schnittstelle Accounting

Die Konfiguration der REST-Schnittstelle von Accounting unterscheidet sich hauptsächlich in der Verlinkung und Anzahl der Ressourcen zu den bisherigen Beispielapplikationen. Der Service Accounting verwendet als Medientyp „XML“. Der Einstiegspunkt dieser REST-Schnittstelle ist die Ressource „Accounting“.

5.4.4.3 Interessanter Aspekt der REST-Schnittstelle

Nachfolgend wird die Konfiguration der Ressourcen erläutert. Dazu dienen *Abbildung 47* und *Abbildung 48*. Die dargestellten Ressourcen „Project“ beziehungsweise „ApprovedInvoice“ zeigen alle drei verschiedenen Linktypen und deren Konfiguration auf. Zusätzlich ist die Einrichtung der „selfUri“ und der URIs der Ressource ersichtlich.



Abbildung 47: Konfiguration der Ressource „Project“

Wie in der Abbildung 47 ersichtlich, besitzt die Ressource „Project“ zwei verschiedene URIs und vier Links zu anderen Ressourcen.

Der URI[1] ein statischer URI. Zu erkennen ist dies am „uriPart“-Element. Dieser URI-Part legt fest, dass die Ressource „Project“ über den URI „/projects“ erreichbar ist. Es ist zudem eine GET-Methode „getProjects“ definiert.

URI[2] definiert den URI „/projects/{projectId}“. Dies wird bei Abbildung 48 ausführlicher erklärt. Auf der URI sind zudem die PUT-Methode „updateProject“ und GET-Methode „getProject“ konfiguriert.

Bei link[1] wird ein CompositionLink mit dem Verbindungsnamen „approvalRoles“ definiert. Dieser Link-Eintrag bildet den Link-URI „ApprovalRoles“ der Ressource „ApprovalRole“ mit der URI „/approvalRoles“ ab. Dem Link ist zudem die Kardinalität „IdentifiedByKey“ zugewiesen. Dies entspricht im konkreten Code einer `HashMap` in der Java-Klasse „Project.java“

link[2] ist als AssociationLink konfiguriert. Der Link-Eintrag bildet der Link-URI „Customer“ mit der URI „/customer/{customerId}“ ab. Die Kardinalität des Links ist „Field“ und bedeutet im konkreten Code, dass eine Variable vom Typ „Customer“ in der Java-Klasse „Project.java“ hinzugefügt wird.

Es ist zusätzlich ersichtlich, dass die Ressource „Project“ ein Feld namens „ID“ hat, welches vom Typ „String“ ist.

Um einen „selfUri“ zum konfigurieren muss dazu einen entsprechenden URI definiert werden. Bei „Project“ ist dies der URI[2].

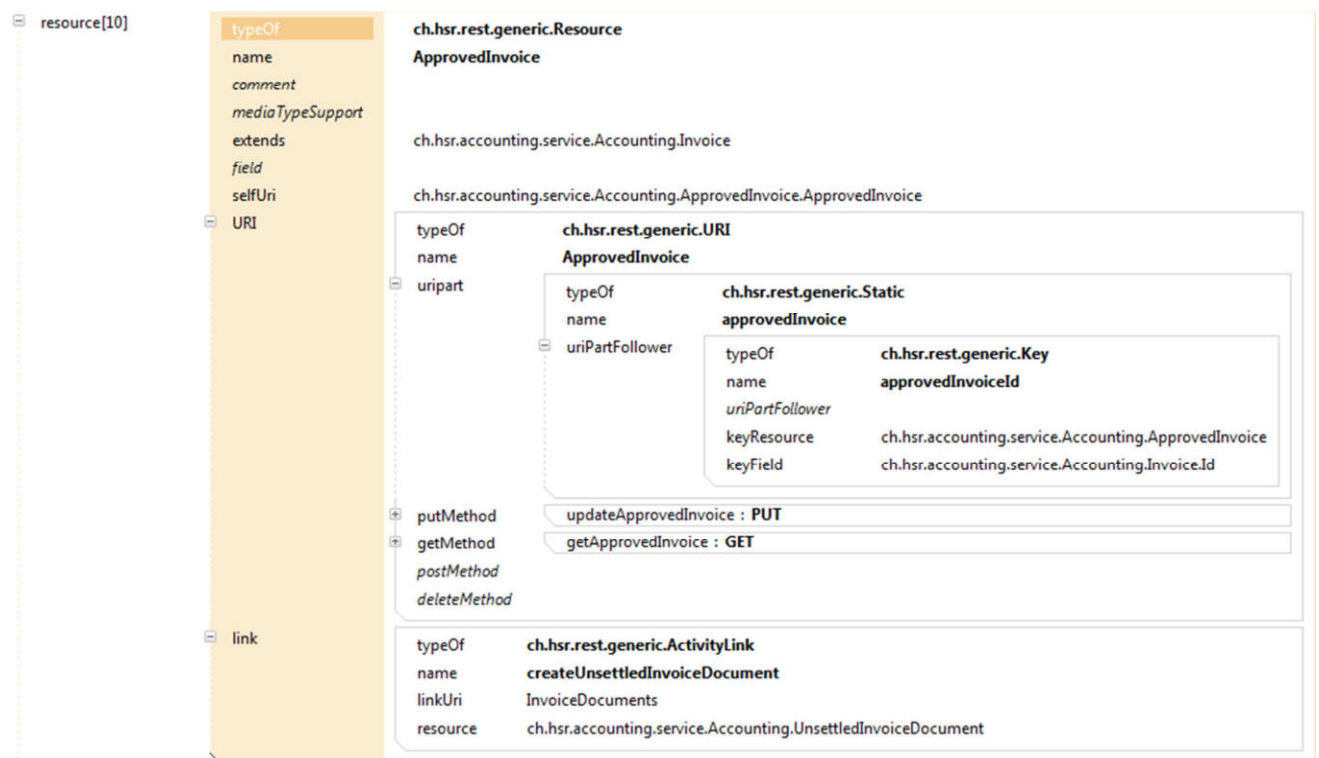


Abbildung 48: Konfiguration der Resource "ApprovedInvoice"

Aus der *Abbildung 48* ist zu erkennen, dass die Ressource „ApprovedInvoice“ einen ActivityLink aufweist.

Um einen solchen Link zu definieren muss auf der Zielressource ein statischer Link konfiguriert sein. Activitylinks können nur auf statische Link abgebildet werden. Die ist für Compositionlinks und Associationslinks nicht der Fall. Diese können auf einen variablen

URI-Part abgebildet werden. Die Art und Weise, wie ein ActivityLink definiert, ist dieselbe wie die anderen beiden Link-Typen.

In der Ressource „ApprovedInvoice“ ist dynamischer Link mit Name „ApprovedInvoice“ konfiguriert. Ein variablen Link wird konfiguriert, in dem zuerst der URI einen UR-Part vom Typ „Static“ und anschliessend als „uriPartFollower“ einen URI-Part vom Typ „Key“ hinzugefügt wird. Zudem müssen noch „keyResource“ und „keyField“ definiert werden. In *Abbildung 48* ist dies mittels „ApprovedInvoice“ als „keyResource“ und „Invoice.Id“ als „keyField“ konfiguriert.

Auf dem eingerichteten URI ist zu erkennen, dass eine PUT- sowie eine GET-Methode definiert sind. Diese wird als „selfUri“ verwendet.

Unter „extends“ ist die Superressource zu erkennen. Es wäre durchaus möglich der Ressource einen weiteren Medientyp unterstützen zu lassen oder einen Kommentar hinzuzufügen. Dazu müsste der Medientyp bei „mediaTypeSupport“ und der Kommentar bei „comment“ definiert werden.

Die Ressourcen aller REST-Schnittstellen werden auf diese Weise konfiguriert.

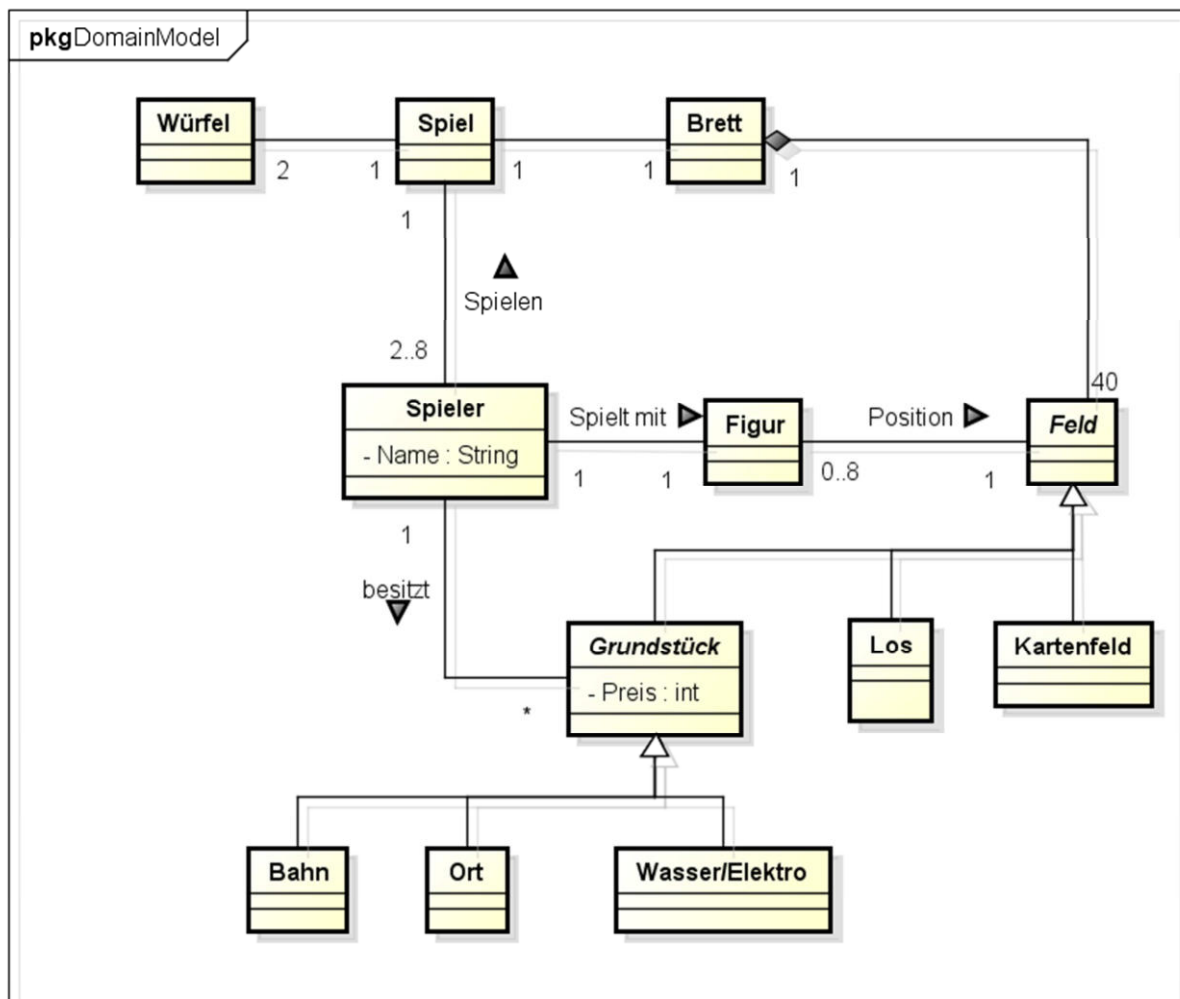
5.5 Beispiel 5: Monopoly

5.5.1 Einleitung

Dieses Beispiel stammt aus dem Buch „Applying UML and Patterns“⁴⁰. Es beschreibt das bekannte Spiel Monopoly. Dabei ist anzumerken, dass das Modell nicht vollständig ist, also nicht alle Regeln des Spiels ausgearbeitet sind.

5.5.2 Ausgangslage

5.5.2.1 Domain Model

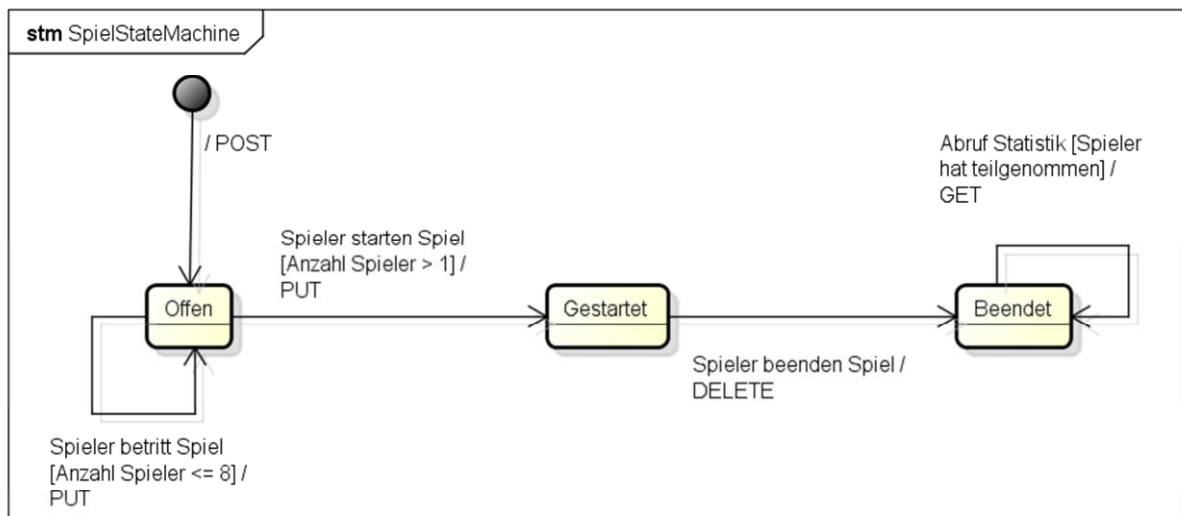


powered by Astah

Abbildung 49: UML-Domain-Model des Monopoly Beispiels

⁴⁰ [Larm1]

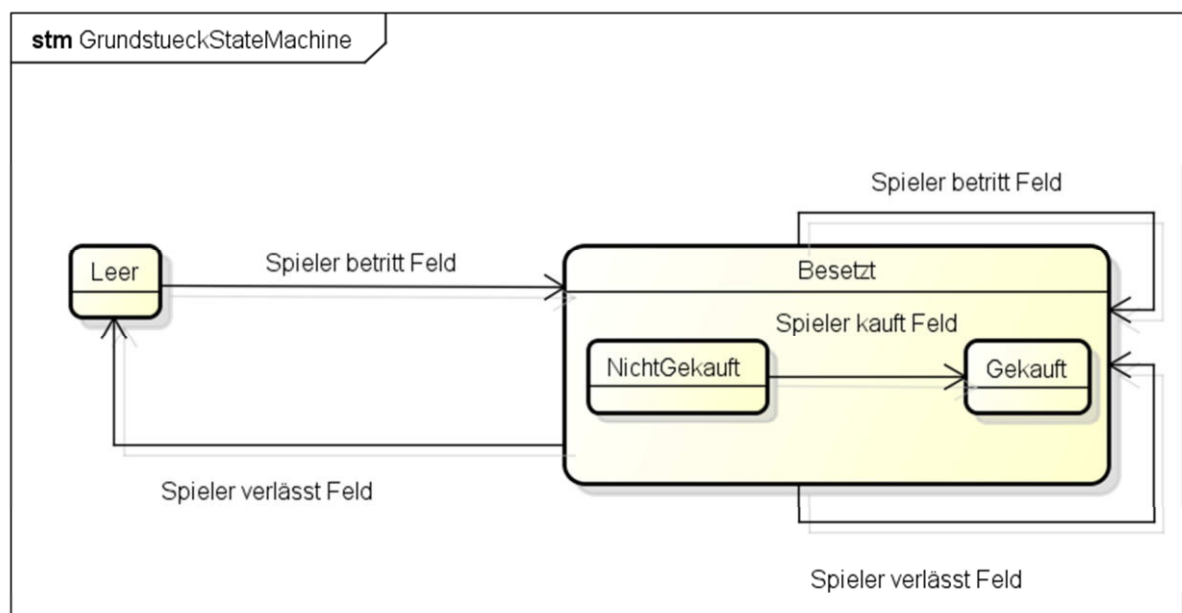
5.5.2.2 State Diagramm Spiel



powered by Astah

Abbildung 50: UML-State-Diagramm für ein Monopoly-Spiel

5.5.2.3 State Diagramm Grundstück



powered by Astah

Abbildung 51: UML-State-Diagramm für ein Grundstück

5.5.3 Erkenntnisse

5.5.3.1 REST-Komponenten

Ressourcen

Ressource	Bemerkungen
Spieler	Jeder Spieler, der sich für ein Spiel anmeldet wird durch eine Ressource repräsentiert.
Feld	Felder haben einen bestimmten Typ, die beim Betreten eines Spielers eine bestimmte Aktion zur Folge hat. Zum Beispiel: Ziehen einer Karte, Kaufmöglichkeit, Bezahlen, Geld-Beziehen
Spiel	Die Haupt-Ressource. Es kann mehrere Spiele geben, zu denen sich, wenn sie noch nicht gestartet sind, Spieler anmelden können

HTTP-Methoden

Verb	Beschreibung
GET	Alle lesenden Anfragen auf eine Ressource.
POST	Neues Spiel erstellen. Als Spieler anmelden
PUT	Spielzüge

Medientypen⁴¹

Media-Type	Beschreibung
application/xml; text/xml	Ober-Typ für XML-Content
application/json	Ober-Typ für JSON-Content
application/vnd.hsr.monopoly+xml	Eigener Typ für Monopoly Anwendung
Application/vnd.hsr.monopoly+json	Eigener Typ für Monopoly Anwendung
Application/atom+xml	Atom-Feed, welcher alle Spielzüge des Spieles enthält

5.5.3.2 Bemerkungen

Das *Beispiel 5: Monopoly* ist nur Beschreibung für die mögliche Umsetzung als REST-Schnittstelle vorhanden. Für dieses Beispiel wurde aus zeitlichen Gründen kein „HATEOAS / REST LEVEL 3“-Workflow-Diagramm erstellt, sowie keine zur Beispielapplikation dazugehörigen REST-Schnittstelle modelliert.

Dieses Beispiel weist einen sehr hohen komplexitätsgrad auf. Es sollte jedoch einem Interessenten, mittels der anderen existierenden Beispiele möglich sein, diese Beispielapplikation nachträglich selbst zu modellieren und zu implementieren.

⁴¹ [IANA]

5.6 Beispiel 6: Visualisierungssoftware

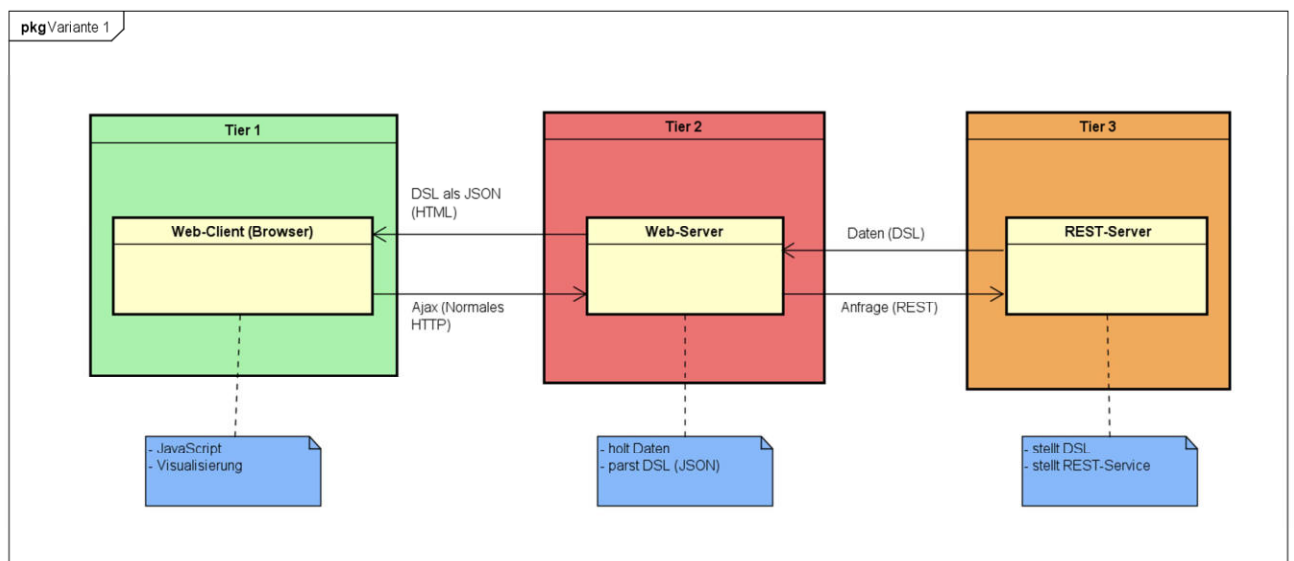
5.6.1 Einleitung

Dieses Kapitel beschreibt die Architektur der Visualisierungssoftware.

5.6.2 Übersicht

Nachfolgend werden zwei mögliche Varianten der Visualisierungssoftware erläutert. Es werden ebenso die Vor- beziehungsweise Nachteile der Varianten aufgezeigt.

5.6.3 3 Tiers: Variante 1



powered by Astah

Abbildung 52: Variante 1 der Visualisierungssoftware

Die erste Variante beinhaltet drei Tiers. Im ersten Tier befindet sich der Web-Client. Im Normalfall ist dies ein beliebiger Browser. Der zweite Tier enthält einen Web-Server und der Tier 3 einen REST-Server.

Der Browser schickt mittels Ajax ein Request an den Web-Server. Dadurch wird eine geparste Domain Specific Language (DSL) angefordert. Der Web-Server holt die angeforderten Daten, durch eine Anfrage beim REST-Server, ab. Dieser stellt den REST-Service und die DSLs zu Verfügung. Der Server von Tier 3 schickt die DSL an den Tier 2 Server, welcher die erhaltene DSL parst und eine JSON-Repräsentation daraus generiert. Anschliessend leitet der Web-Server die DSL als JSON, möglich wäre auch HTML, an den Web-Client. JavaScript, dass im Browser läuft verarbeitet die erhaltene Repräsentation und lässt diese vom Web-Client rendern und auf dem Bildschirm darstellen.

5.6.3.1 Tier 1

Im Web-Client (zum Beispiel ein Browser) wird ein Ajax-Call abgesetzt, um vom Webserver die JSON-Repräsentation der gewünschten DSL zu erhalten.

5 Anwendung der Konzepte auf Applikationsbeispiele

Nach erfolgreichem erhalten der zuvor genannten Repräsentation, wird diese dem JavaScript übergeben. Das Script generiert eine SVG-Repräsentation daraus, welche im Web-Client dargestellt wird. Dieser Client ist für das Rendern der SVG Grafik zuständig.

5.6.3.2 Tier 2

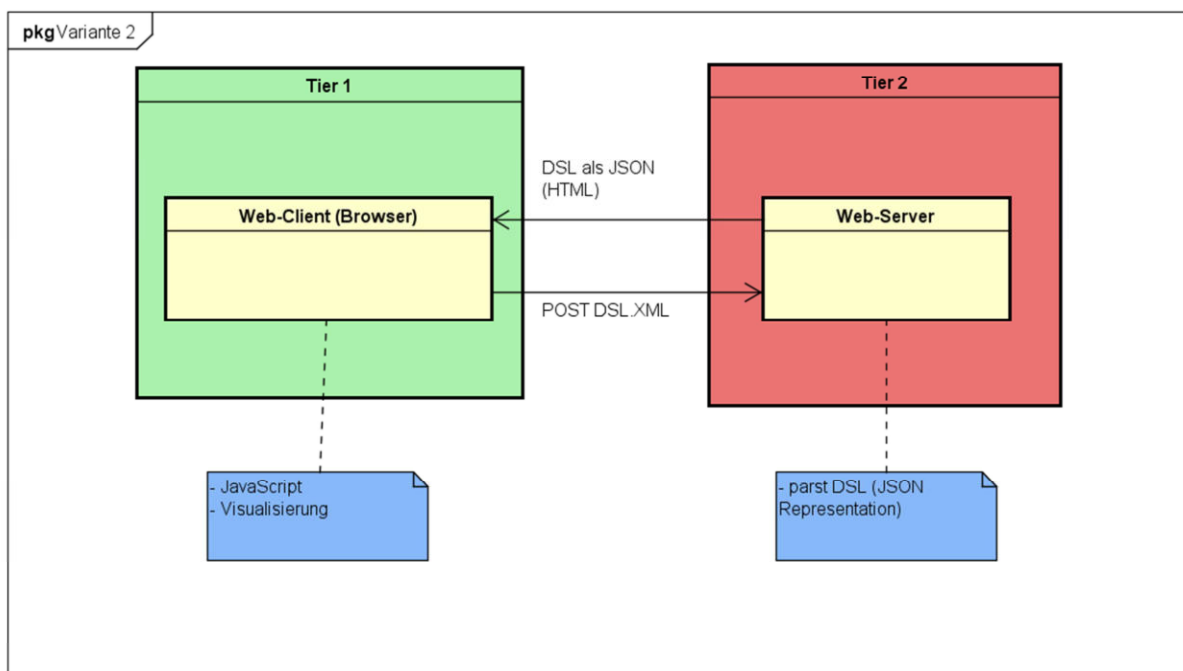
Der Web-Server leitet die Anfrage des Web-Clients weiter an den REST-Server.

Nach Erhalt der DSL in der gewünschten Repräsentation wird diese in eine JSON Repräsentation geparkt. Anschliessend wird sie an den Web-Client weitergeleitet.

5.6.3.3 Tier 3

Der REST-Server bietet Links an, um die vorhandenen DSL's anzubieten. Wenn der Web-Server eine DSL über den entsprechenden Link anfordert, wird im diese übermittelt.

5.6.4 2 Tiers: Variante 2



powered by Astah

Abbildung 53: Variante 2 der Visualisierungssoftware

Die zweite Variante besteht nur aus zwei Tiers, den Tier 1 mit einem Web-Client und den Tier 2 mit einem Web-Server.

Das Vorgehen ist leicht anders als bei der Variante 1. In diesem Fall wird vom Browser, via POST eine DSL, als XML, dem Web-Server übergeben. Dieser parst die DSL in eine JSON Repräsentation, welche anschliessend zurückgeschickt wird. Möglich wäre auch, dass die DSL als eine HTML-Repräsentation zurückgeliefert wird. Anschliessend verarbeitet das JavaScript die Repräsentation und lässt diese durch den Browser rendern und auf dem Bildschirm anzeigen.

5.6.4.1 Tier 1

Einem JavaScript, welches im Web-Client (zum Beispiel ein Browser) läuft, wird die JSON-Repräsentation der DSL übergeben. Das JavaScript generiert dann anhand der erhaltenen Repräsentation eine SVG(Scalable Vector Graphics) - Repräsentation. Diese wird im Web-Client dargestellt. Der Web-Client ist zuständig für das Rendern der SVG-Repräsentation.

5.6.4.2 Tier2

Auf dem Webserver wird mittels Java die erhaltene XML-Repräsentation der DSL in JSON geparkt. Dies geschieht mittels „json-lib“.

5.6.5 Variante 1 vs. Variante 2

	+	-
3-Tier Variante	<ul style="list-style-type: none"> • Beide Server werden generiert <ul style="list-style-type: none"> ◦ Nur Business-Logik muss noch implementiert werden. 	<ul style="list-style-type: none"> • Die Architektur hat eine zusätzliche Indirektion.
2-Tier Variante	<ul style="list-style-type: none"> • Einfacher 	<ul style="list-style-type: none"> • Nur ein Server wird generiert

5.6.5.1 Entscheid

Auf Grund der zeitlichen Verhältnisse während der Bachelorarbeit und Anzahl Beispielapplikationen, welche auszuarbeiten sind, haben wir uns für die leichtere „2 Tiers: Variante 2“ entschieden. Zudem kommt zum Tragen, dass die Client-Schnittstelle nicht generiert und somit die „3 Tiers: Variante 1“ nicht implementiert werden kann.

Die Variante 2 besitzt nur einen Server-Teil und einen Web-Client, welcher in unserem Falle ein Browser ist. Da nur die Server-Schnittstelle generiert werden muss, ist der Entscheid sich auf die 2-Tier-Variante zu konzentrieren zusätzlich ein für die Erfüllbarkeit der Anforderungsspezifikation wichtiger Entschluss.

5.6.6 Spezifikation der DSL

In diesem Abschnitt wird der genaue Aufbau der DSL erläutert, wie es die REST-Schnittstelle RestVisualisation erwartet.

5.6.6.1 Spezifikation des XML – Aufbau

```

1  <dsl>
2  <resource name="BeispielResource" id="0">
3  <mediaType>
4  <name>XML</name>
5  <eof/>
6  </mediaType>
7  <field>
8  <name>id</name>
9  <eof/>
10 </field>
11 <field>
12 <name>quantity</name>
13 <eof/>
14 </field>
15 <link>
16 <linkType>composition</linkType>
17 <to>1</to>
18 </link>
19 <link>
20 <linkType>inheritance</linkType>
21 <to>12</to>
22 </link>
23 <link>
24 <linkType>association</linkType>
25 <to>17</to>
26 </link>
27 <link>
28 <linkType>activitylink</linkType>
29 <to>24</to>
30 </link>
31 </resource>
32 </dsl>

```

Abbildung 54: Beispiel eines XML-Aufbaus der DSL

In der *Abbildung 54*: Beispiel eines XML-Aufbaus ist der Aufbau einer XML-Datei (eXtended Markup Language) dargestellt, welche der REST-Schnittstelle RestVisualisation übergeben werden kann.

Alle Ressourcen werden von einem Root-XML-Tag umschlossen, wie in der *Abbildung 54* der `<dsl>` XML-Tag zeigt. Die Ressourcen werden innerhalb dieses Tags nacheinander aufgelistet. Name des Root-Knoten ist nicht relevant.

Ein `<resource>` -Kind-Knoten muss immer die Attribute „name“ und „id“ enthalten. Es ist darauf zu achten, dass die IDs bei „0“ beginnen und fortlaufend sind. Des Weiteren können innerhalb des `<resource>` -Knoten die Kinder-Knoten `<mediaType>`, `<field>` und `<link>` definiert werden. Diese XML-Knoten sind nicht zwingend notwendig. Die entsprechenden Kind-Knoten können weglassen werden, wenn keine Medientypen, Felder oder Links vorhanden sind.

Im Knoten `<mediaType>` wird ein Medientyp konfiguriert. Der Kind-Knoten `<name>` enthält den Namen des Medientyps. Es ist wichtig darauf zu achten, dass der XML-Tag `<eom/>` als Kind-Knoten von jedem `<mediaType>` vorhanden ist, welcher definiert wurde. Der `eom`-Tag wird für den korrekten Aufbau der JSON-Datei benötigt. Wenn mehrere Medientypen benötigt werden, werden diese nacheinander als Kind-Knoten des `<resource>`-Knoten hinzugefügt.

Der Kind-Knoten `<field>` beinhaltet ein auf der Ressource zu definierendes Feld. Wie der vorhergehender Medientypen-Kind-Knoten enthält der `field`-Knoten ein Kind-Knoten `<name>` für den Namen des Feldes. Des Weiteren ist ebenfalls wichtig darauf zu achten, dass der Kind-Knoten `<eof/>` in jedem `<field>`-Knoten vorhanden ist, welcher definiert wurde. Dieser XML-Tag wird aus denselben Gründen, wie der `eom`-Tag, benötigt. Bei mehreren vorhandenen Feldern, werden diese nacheinander aufgelistet.

Um die Links auf den Ressourcen zu konfigurieren, wird der Kind-Knoten `<link>` benötigt. Dieser beinhaltet seine XML-Kind-Knoten `<linkType>` und `<to>`. In `linkType` wird der Typ des Links angegeben. Im `<to>`-Kind-Knoten, wird die ID der Ziel-Ressource angegeben. Die Links werden, wie bisher bei den Medientypen und Felder nacheinander hinzugefügt.

Es existieren die folgenden Linktypen:

- **composition**
- **inheritance**
- **association**
- **activitylink**

`eof` bedeutet „end of field“. Folglich hat `eom` die Bedeutung „end of mediatype“. Die Medientypen, Felder und Links müssen in dieser Reihenfolge, wie es in *Abbildung 54* abgebildet ist, aufgelistet werden.

5.6.6.2 Spezifikation des JSON – Aufbau

```

1  [{
2    "@name": "BeispielResource",
3    "@id": "0",
4    "mediaType": {
5      "name": "XML",
6      "eom": []
7    },
8    "field": [
9      {
10     "name": "id",
11     "eof": []
12   },
13   {
14     "name": "quantity",
15     "eof": []
16   }
17 ],
18 "link": [
19   {
20     "linkType": "composition",
21     "to": "1"
22   },
23   {
24     "linkType": "inheritance",
25     "to": "12"
26   },
27   {
28     "linkType": "association",
29     "to": "17"
30   },
31   {
32     "linkType": "activitylink",
33     "to": "24"
34   }
35 ]
36 }]

```

Abbildung 55: Beispiel eines JSON-Aufbaus

Die *Abbildung 55* zeigt die nach JSON (**J**ava**S**cript **O**bject **N**otation) geparte XML-Datei aus *Abbildung 54*.

Die Attribute „name“ und „id“ der Ressource, sind mit einem führenden „@“ versehen. Ebenfalls zu erwähnen ist, dass sobald mehr als ein gleicher Kind-Knoten in der Ausgangs-XML-Datei vorhanden ist, deren Inhalte von einem Object zu einem Array von Object gepart. Die Namen der mehrfach-vorkommenden XML-Kind-Knoten werden somit, entweder zu einem Object oder zu einem Array mit diesem Namen gepart.

Zum Beispiel „field“ wurde in ein Array übersetzt und „mediaType“ in ein Object. Auch zu erkennen sind die „eof“ und „eom“ – Knoten, welche als Array gepart wurden.

5.6.7 REST-Schnittstelle RestVisualisation

Der Abschnitt 5.6.7 beschreibt die REST-Schnittstelle RestVisualisation, welches die modellierte und implementierte Ausgabe von 2 Tiers: Variante 2 ist.

5.6.7.1 REST-Schnittstelle in acitfsource

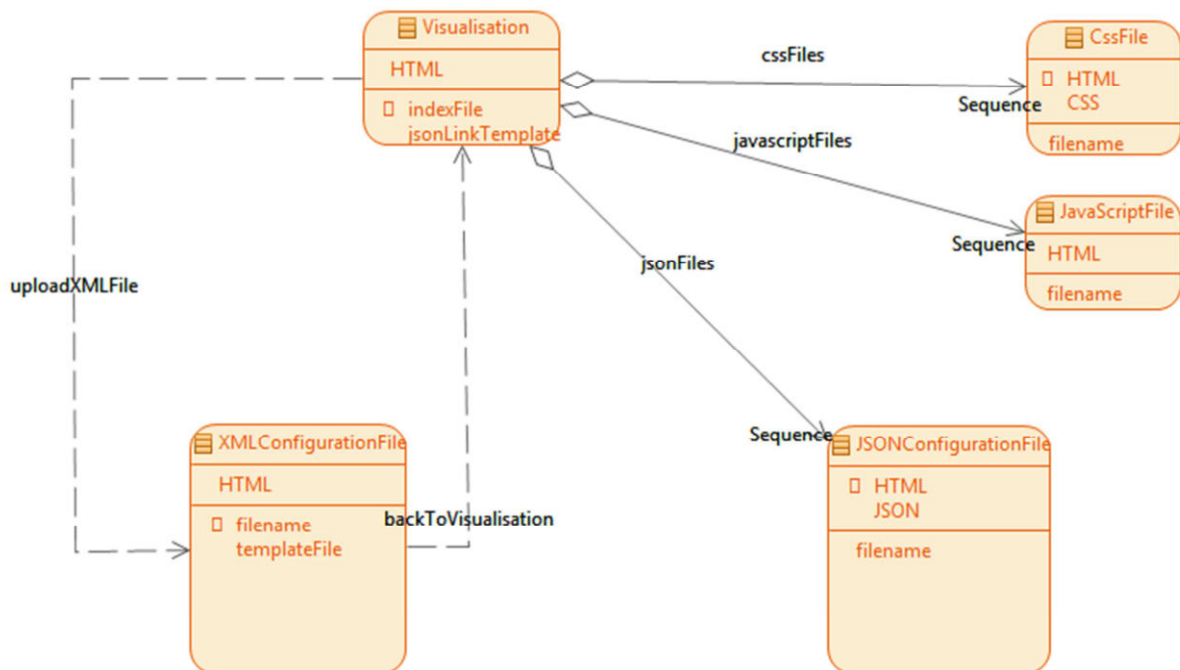


Abbildung 56: Business-Modell der REST-Schnittstelle RestVisualisation

Die Beispielapplikation RestVisualisation besteht aus fünf Ressourcen. Die „Visualisation“-Ressource ist der Einstiegspunkt der REST-Schnittstelle. Die Ressource definiert zwei Felder. Das Feld „indexFile“ wird für die Verlinkung der Datei „index.html“ benötigt. Das „jsonLinkTemplate“-Feld realisiert dasselbe wie „indexFile“ nur für die Datei „jsonFileLink.html“

Der Benutzer wählt mittels Web-Frontend die XML-Datei für den Upload aus und benützt den ActivityLink „uploadXMLFile“, indem er auf den Submit-Button betätigt.

Die Ressource „XMLConfigurationFile“ ist für das Hochladen der XML-Datei, sowie das Parsen dieser Datei in die entsprechende JSON-Datei. Das Feld „filename“ wird für die Namen des XML und des JSON benötigt. Das „templateFile“-Feld ist für die „xmlUploadLandingPage.html“-Datei zuständig. Auf dieser Seite wird der ActivityLink „backToVisualisation“ dargestellt.

Zurück auf der Startseite wird der der Link zur Visualisierung der neuen JSON-Datei angezeigt. Dieser Link-Eintrag wird mittels des CompositionLink zur Ressource „JSONConfigurationFile“ ermittelt. Indem der Benutzer weitere DSLs hochlädt, werden mehr Links in dieser Liste angezeigt.

Für die Verlinkung der JavaScript- und CSS-Datei in „index.html“, werden die Ressourcen „CssFile“ beziehungsweise „JavaScriptFile“ benötigt.

5.6.7.2 Konfiguration der REST-Schnittstelle

typeOf	ch.hsr.rest.generic.Service
name	RestVisualisation
package	ch.hsr
host	localhost
+ mediaTypeDefault	HTML : HTML
+ mediaTypeAuxiliary[1]	XML : XML
+ mediaTypeAuxiliary[2]	CSS : CustomMediaType
+ mediaTypeAuxiliary[3]	JSON : CustomMediaType
entryPoint	
+ resource[1]	Visualisation : Resource
+ resource[2]	XMLConfigurationFile : Resource
+ resource[3]	JSONConfigurationFile : Resource
+ resource[4]	CssFile : Resource
+ resource[5]	JavaScriptFile : Resource

Abbildung 57: Konfiguration der RestVisualisation

In der *Abbildung 57* ist die Konfiguration der REST-Schnittstelle RestVisualisation ersichtlich.

Diese benötigt, um korrekt zu arbeiten, drei Medientypen. HTML ist der Standard-Medientyp und wird von allen Ressourcen benötigt. Den Medientyp CSS braucht die Ressource „CssFile“ und JSON verwendet „JSONConfigurationFile“. Der XML-Medientyp wird zurzeit nicht benötigt.

5.6.7.3 Interessanter Aspekt der REST-Schnittstelle

Diese REST-Schnittstelle zeigt, wie solche Schnittstellen direkt mit Browsern zusammenarbeiten können. Es besteht jedoch eine Einschränkung. Da Browser keine REST-Clients sind und HTML keine PUT- und DELETE-Methoden unterstützen können REST-Schnittstellen im direkten Kontakt nur GET und POST verwenden. Dies entspricht den Vorstellungen von Level 2 des REST-Maturitätsmodell. Es ist jedoch akzeptabel.

Um die komplette Unterstützung aller HTTP-Methoden zu erreichen müsste, wie unter *Abbildung 52: Variante 1 der Visualisierungssoftware* beschrieben die 3 Tiers: Variante 1 verwendet werden. Dazu muss jedoch noch der Meta-Code für die Generierung der Client-Schnittstelle entwickelt werden.

6 ZUSAMMENFASSUNG UND AUSBLICK

6.1 Zusammenfassung

Die Arbeit zeigt auf, dass die Modellierung sowie Generierung von REST-Schnittstellen möglich ist. Als Basis dazu dient das erarbeitete Meta-Modell. Es deckt die für REST-Schnittstellen notwendigen Komponenten ab. Die Integration des Meta-Modells in die actifsource-Software lässt das grafische Modellieren von Ressourcen und deren Verbindungen, also Links, zu. Zudem erlaubt sie das einfache Erfassen der notwendigen Adressierung von Ressourcen in der Schnittstelle. Ebenfalls wichtig, wenn von Hypermedia gesprochen wird, ist die Möglichkeit Ressourcen in den verschiedenen Medien- und Hypertextformate darzustellen. Dies ist mit der Software möglich.

Die generierten Schnittstellen erfüllen die REST-Prinzipien vollständig. Mit dem entwickelten Meta-Code für die Technologien ApacheHTTP und JAX-RS wurden fünf Beispielapplikationen mit verschiedenen Anforderungen konzeptionell umgesetzt und teilweise ausprogrammiert.

Der Meta-Code führt dazu, dass der Programmcode der REST-Schnittstellen einheitlich und übersichtlich ist. Dies resultiert in einer Aufwandsreduktion für Neu- und Weiterentwicklungen sowie Wartung für Software-Entwickler. Dies wird mit Zahlen aus den erhobenen Metriken belegt. Diese Werte besagen, dass bis zu 90% des notwendigen Source-Codes der REST-Schnittstelle generiert wird. Der Aufwand für Entwickler beschränkt sich auf das Schreiben und Integrieren der eigentlichen Business-Logik in die Schnittstelle.

6.2 Ausblick

Für eine Folgearbeit wäre die Unterstützung weiterer Ziel-Technologien in Form von Meta-Code denkbar. Beispiele solcher Technologien sind C#/.NET, PHP oder ähnlichen Programmier- und Skriptsprachen. Dazu kann das ausgearbeitete Meta-Modell in der actifsource-Software verwendet und wenn nötig erweitert werden.

Als weitere Ergänzungsmöglichkeit, wäre das Entwickeln eines ausgereiften Konzepts für Sicherheit und Benutzeridentifikation für REST-Schnittstellen.

Eine dritte Möglichkeit wäre, das Meta-Modell in eine andere Software, wie zum Beispiel eclipse-EMF oder Enterprise Architect zu portieren und damit REST-Schnittstellen zu modellieren und generieren.

ANHANG A

Abbildungsverzeichnis

Abbildung 1: Übersicht der Architektur von REST Hypermedia Modelling & Visualisation	6
Abbildung 2: Ein Ausschnitt aus dem REST-Meta-Modell.	8
Abbildung 3: Graphik zum Richardson Maturity Model	13
Abbildung 4: Übersicht der verschiedenen Modell- und Code-Stufen	16
Abbildung 5: Use Case Diagramm.....	22
Abbildung 6: Übersicht Kontext	29
Abbildung 7: Übersicht Kontext mit Applikationsbeispielen	30
Abbildung 8: REST-Meta-Modell	32
Abbildung 9: REST-Meta-Modell in der actifsource Software	33
Abbildung 5: Die Konfigurationsmaske der Service-Komponente in der actifsource- Umgebung.....	33
Abbildung 11: Formular zum erfassen einer Ressource in actifsource.....	34
Abbildung 12: Formular zum erfassen eines URI in actifsource.....	35
Abbildung 8: Beispiel für Link-Typen	36
Abbildung 9: Die Ressource MediaType und deren Untertypen.....	37
Abbildung 10: Beispiel einer actifsource-BuildConfig: Die BuildConfig für ApacheHttp bindet die Generelle BuildConfig ein	38
Abbildung 16: Aufbau eines actifsource-Templates: Dieses Beispiel zeigt das Resource Template	39
Abbildung 17: Aufbau der Templates für den Generellen Meta-Code.	40
Abbildung 18: Definition eines Feldes in einer Ressource mit get und set Methode	41
Abbildung 19: : Auszug aus Template und generiertem Code des RESTResourceAbstractAdapter	42
Abbildung 20: Auszug aus Template und generiertem Code der RESTLinkFactory	43
Abbildung 16: Model der RESTLink Klasse	43
Abbildung 22: Auszug aus Template und generiertem Code des RESTUriParsers	44
Abbildung 23: Serverseitiger Ablauf der ApacheHTTP Variante als UML–Sequenz-Diagramm	45
Abbildung 24: Serverseitiger Ablauf der JAX-RS Variante als UML–Sequenz-Diagramm.....	47
Abbildung 25: Template und generierter Code eines URI-Handlers in JAX-RS	48
Abbildung 26: Das webxml-Template	49
Abbildung 27: Domainmodel für die Terminreservierung	50
Abbildung 28: Workflow / Erreichbarkeit der Ressourcen	52
Abbildung 29: Business-Modell der REST-Schnittstelle DoctorService.....	55
Abbildung 30: Konfiguration des Business-Modelles der REST-Schnittstelle DoctorService	56
Abbildung 31: Konfiguration der Medientypen vom DoctorService	57
Abbildung 32: UML-Domain Model RestBucks	58
Abbildung 33: UML-State-Diagramm der Bestellung "Order"	59
Abbildung 34: Business-Modell der REST-Schnittstelle RestBucks in der actifsource- Umgebung.....	62
Abbildung 35: Konfiguration der REST-Schnittstelle RestBucks in der actifsource-Umgebung	63
Abbildung 36: Ressource Order deren Subressourcen (Ausschnitt aus <i>Abbildung 34</i>)	64
Abbildung 37: Domain-Model der E-Commerce Anwendung	65

Abbildung 38: Workflow E-Commerce Applikation als UML-State-Diagramm	67
Abbildung 39: Business-Modell der REST-Schnittstelle E-Commerce	69
Abbildung 40: Konfiguration der REST-Schnittstelle E-Commerce	70
Abbildung 41: UML-Domain-Model für Rechnungsstellung	74
Abbildung 42: UML-State-Diagramm einer Rechnung	75
Abbildung 43: UML-State-Diagramm von Projekt	76
Abbildung 44: Workflow Verrechnung.....	78
Abbildung 45: Business-Modell der REST-Schnittstelle Accounting	80
Abbildung 46: Konfiguration der REST-Schnittstelle Accounting	81
Abbildung 47: Konfiguration der Ressource "Project"	82
Abbildung 48: Konfiguration der Resource "ApprovedInvoice"	83
Abbildung 49: UML-Domain-Model des Monopoly Beispiels.....	85
Abbildung 50: UML-State-Diagramm für ein Monopoly-Spiel.....	86
Abbildung 51: UML-State-Diagramm für ein Grundstück	86
Abbildung 52: Variante 1 der Visualisierungssoftware	88
Abbildung 53: Variante 2 der Visualisierungssoftware	89
Abbildung 54: Beispiel eines XML-Aufbaus der DSL	91
Abbildung 55: Beispiel eines JSON-Aufbaus	93
Abbildung 56: Business-Modell der REST-Schnittstelle RestVisualisation.....	94
Abbildung 57: Konfiguration der RestVisualisation	95

Literatur

- [RESTIP] Jim Webber, Savas Parastatidis, Ian Robinson, „REST in Practice“, O'Reilly Media Inc., First Edition 2010, ISBN: 978-0-596-80582-1
- [IANA] Webseite mit reservierten / gängigen HTTP-Medien-Typen
<http://www.iana.org/assignments/media-types>
Zuletzt aufgerufen: 19.12.2013
- [todayu] Webseite der Projektmanagements-Software todayu
<http://www.todayu.com>
Zuletzt aufgerufen: 19.12.2013
- [Larm1] Craig Larman, „Applying UML and Pattern“, Pearson Education Inc., 3rd Edition 2005, ISBN: 0-13-148906-2
- [FowWeb] Blogbeitrag zum Richardson Maturity Model
<http://martinfowler.com/articles/richardsonMaturityModel.html>
Zuletzt aufgerufen: 19.12.2013
- [W3org] Webseite der W3C
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
Zuletzt aufgerufen: 19.12.2013
- [OAUTH] Website von OAuth

- <http://oauth.net/>
Zuletzt aufgerufen: 19.12.2013
- [OID] Website von OpenID
<http://openid.net/>
Zuletzt aufgerufen: 19.12.2013
- [HTML5BUG] Bug 10671 - consider adding support for PUT and DELETE as form methods
https://www.w3.org/Bugs/Public/show_bug.cgi?id=10671
Zuletzt aufgerufen: 19.12.2013
- [SILEX] Website des PHP-Framworks Silex
<http://silex.sensiolabs.org/>
Zuletzt aufgerufen: 19.12.2013
- [PAUT1] Some REST Design Patterns (and Anit-Patterns), from Cesare Pautasso, Faculty of Informatics, University of Lugano
<http://de.slideshare.net/cesare.pautasso/some-rest-design-patterns-and-antipatterns>
Zuletzt aufgerufen: 19.12.2013
- [PAUT2] RESTful Service Design, from Cesare Pautasso, Faculty of Informatics, University of Lugano
http://www.academia.edu/2817878/RESTful_Service_Design
Zuletzt aufgerufen: 19.12.2013
- [JAXRS] Java API for RESTful Services (JAX-RS)
<https://jax-rs-spec.java.net/>
Zuletzt aufgerufen: 19.12.2013
- [RFC3986] RFC Standard für URI
<http://www.ietf.org/rfc/rfc3986.txt>
Zuletzt aufgerufen: 19.12.2013
- [RFC2616] RFC Standard für HTTP 1.1
<http://tools.ietf.org/html/rfc2616>
Zuletzt aufgerufen: 19.12.2013
- [Field00] Dissertation von Roy Fielding
http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
Zuletzt aufgerufen: 19.12.2013
- [ECLIPSEWTP] eclipse Web Tools Platform Project
<http://www.eclipse.org/webtools/>
Zuletzt aufgerufen: 19.12.2013

- [RAML] RESTful API Modeling Language
<http://raml.org/>
Zuletzt zugegriffen: 19.12.2013
- [WIKSCHABMETH] Schablonenmethode, (englisch: „template method pattern“)
<http://de.wikipedia.org/wiki/Schablonenmethode>
Zuletzt zugegriffen: 19.12.2013
- [WIKIJAVANNOT] Annotation (Java)
[http://de.wikipedia.org/wiki/Annotation_\(Java\)](http://de.wikipedia.org/wiki/Annotation_(Java))
Zuletzt zugegriffen: 19.12.2013
- [JERSEY] Jersey - RESTful Web Services in Java
<https://jersey.java.net/>
Zuletzt zugegriffen: 19.12.2013

Glossary

Begriff	Beschreibung
.Net	Software-Plattform zur Entwicklung und Ausführung von Anwendungsprogrammen.
actifsource	Name der Firma und gleichnamigem eclipse Plugin bzw. Tool (actifsource.com, 2013)
BA	Bachelorarbeit
C#	Programmiersprache
CSS	C ascading S tyle S heets.
DSL	Domain Specific Language
EMF	E clipse M odelling F ramework. Framwork zum Modellieren von Java-Applikationen.
FR	F unktionale R equirements
GUI	G raphical U ser I nterface
HATEOAS	H ypermedia a s the E ngine of A pplication S tate.
HTML	H ypertext M arkup L anguage
HTTP	H ypertext T ransfer P rotocol.
JAX-RS	Java API für RESTful Web Services
JERSEY	Eine Implementation der JAX-RS API
JSON	JavaScript Object Notation -
JUnit	Java-Framwork für automatisiertes Testen von Applikationen
Meta-Modell	Model auf der Meta-Ebene. Es beschreibt ein Modell.
MOF	M eta O bject F acility
NFR	N icht- F unktionale R equirements
OMG	O bject M anagement G roup
PHP	Serverseitig interpretierte Skriptsprache.
POX	Plain Old XML
RAML	RESTful API Modelling Language
REST	R epresentational S tate T ransfer.
RESTful	Alle Prinzipien von REST sind erfüllt.
Silex	PHP-basiertes Framework zur Entwicklung von RESTful Webseiten.
SQLite	Leichtgewichtige SQL-Basierte Datenbank
SVG	S calable V ector G raphics
todayu	Webbasierte Projektmanagementssoftware

Tomcat	Webserver für Java-basierte Applikationen.
UC	U se C ase
UML	U nified M odeling L anguage. Sprache, die es erlaubt unter anderem Software grafisch zu modellieren.
URI	U niform R esource I dentifier .
URL	U niform R esource L ocator
XML	E xtensible M arkup L anguage

ANHANG B

Im Anhang befinden sich eine Benutzeranleitung sowie die Auswertung der Metriken der generierten REST-Schnittstellen.

REST Hypermedia Modelling & Visualisation

Benutzeranleitung

INHALTSVERZEICHNIS

Inhalt

Inhaltsverzeichnis.....	0
Inhalt.....	1
1 Einleitung	3
1.1 Ziel	3
1.2 Installation notwendiger Software.....	3
1.3 (Optional) Tomcat-Server und JavaEE–Webprojekt-Umgebung.....	3
1.3.1 Installation Web-Umgebung	4
1.4 Installation und Erstellen eines Tomcat Server.....	5
2 Modellierung.....	7
2.1 Vorbereitung.....	7
2.2 Import Basisprojekt.....	7
2.3 Neues Projekt erstellen	8
2.4 Neuer Service	9
2.5 Neues Diagramm	12
2.6 Ressourcen und Links modellieren.....	13
2.7 REST-Schnittstelle konfigurieren.....	15
2.7.1 Ausgangslage	15
2.7.2 Ressourcen.....	15
2.7.3 Felder und Feldtypen	16
2.7.4 URI.....	17
2.7.5 URIPart	17
2.8 HttpMethode.....	19
2.9 Links	19
2.9.1 Medientypen	20
3 Einbinden von Applikationscode.....	21
3.1 Controller	21
3.2 Repräsentation.....	23
4 Technologie spezifische Konfigurationen	25
4.1 ApacheHTTP.....	25
4.2 JAX-RS	27
4.2.1 Neues „Dynamic Web Project“	27
4.2.2 Deployment auf dem Tomcat-Server.....	31

5	Testen der Schnittstelle	33
5.1	REST-Client	33
5.2	Bedienung	33

1 EINLEITUNG

1.1 Ziel

Das Ziel dieser Anleitung ist, die Funktionsweise der Modellierungssoftware kenne zu lernen und ein erstes Projekt zu erstellen.

1.2 Installation notwendiger Software

Als erstes muss die notwendige Software installiert werden. Voraussetzungen sind Java Version 1.6 (jre7) und eclipse Kepler (eclipse Version 4.3.1) Minimale Voraussetzung ist eclipse in der Version 3.8. Die nötige Software kann von folgenden Quellen bezogen werden.

- Eclipse: <http://www.eclipse.org/downloads/>
- Java: <http://www.java.com/de/>

Die actifsource Umgebung kann von der actifsource Webseite¹ bezogen werden. Notwendig ist die Enterprise-Edition. Bitte die Voraussetzungen beachten und der Installationsanleitung folgen².

1.3 (Optional) Tomcat-Server und JavaEE–Webprojekt-Umgebung

Für den Fall, das ein Projekt mit JAX-RS erstellt werden soll müssen zusätzlich ein Tomcat-Server und die Web-Entwicklungsumgebung von eclipse installiert werden.

¹ <http://www.actifsource.com/download/index.html>

² http://www.actifsource.com/_downloads/ActifsourceTutorial_InstallingActifsource.pdf

1.3.1 Installation Web-Umgebung

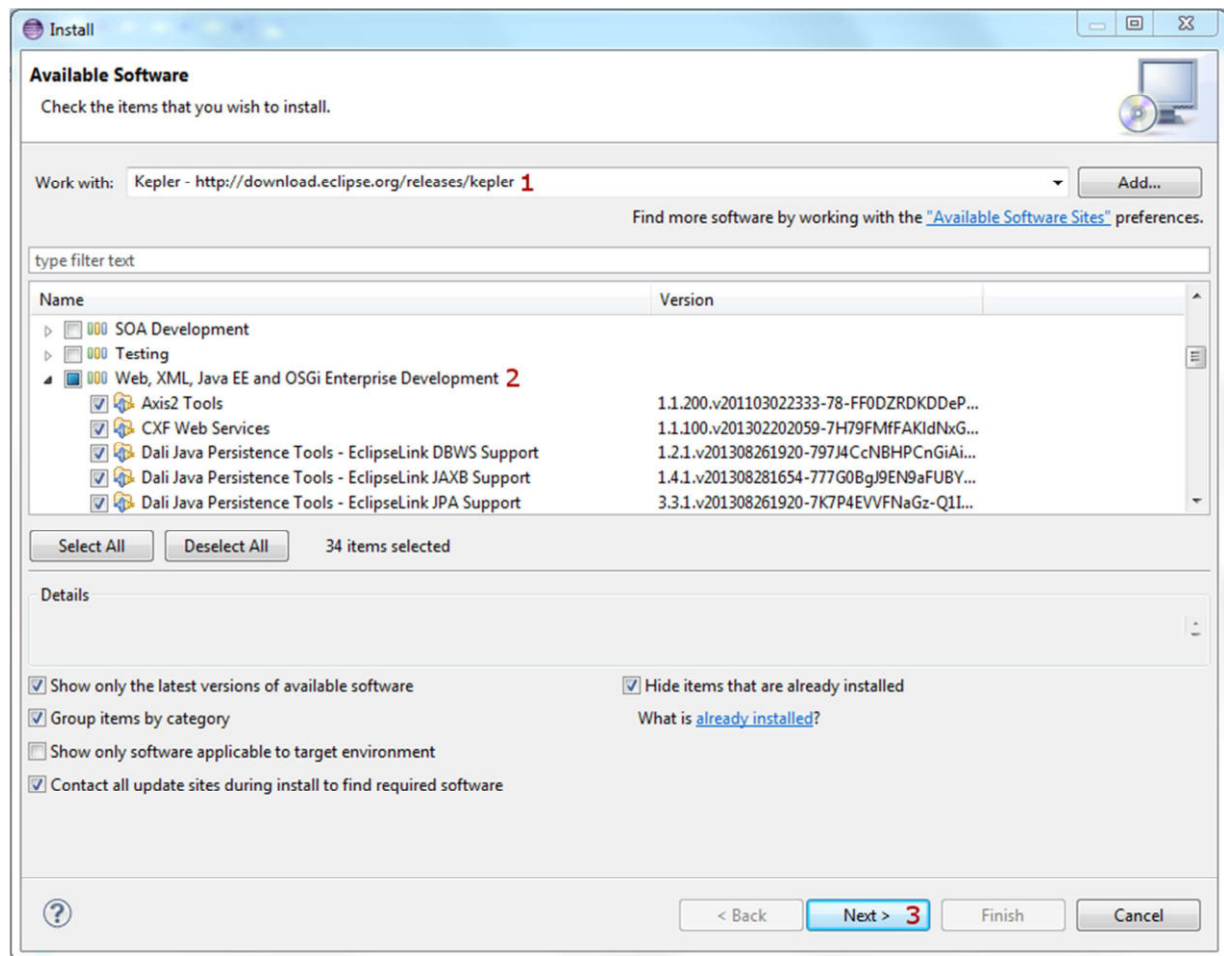


Abbildung 1: Eclipse Ansicht zum Installieren von neuer Software

1. Starten Sie **eclipse** und wählen Sie eine **workspace**
2. Über den Menüpunkt **Window > Install new Software...** kann die Software installiert werden
3. Die nötige Downloadsite³ sollte bereits im Dropdown vorhanden sein. Wählen Sie diese aus (1)
4. Wählen Sie das Paket **Web, XML, Java EE and OSGi Enterprise Development** (2)
 - 4.1. Nicht nötig sind die Pakete *RAP Tools* und *PHP Development Tools*.
5. Klicken Sie auf den Button **Next** (3) und folgend Sie den Anweisungen.
 - 5.1. Akzeptieren Sie die Lizenzvorgaben und klicken Sie Danach auf den **Finish** Button
6. Nach dem Installationsvorgang muss eclipse neu gestartet werden.

³ <http://download.eclipse.org/releases/kepler>

1.4 Installation und Erstellen eines Tomcat Server

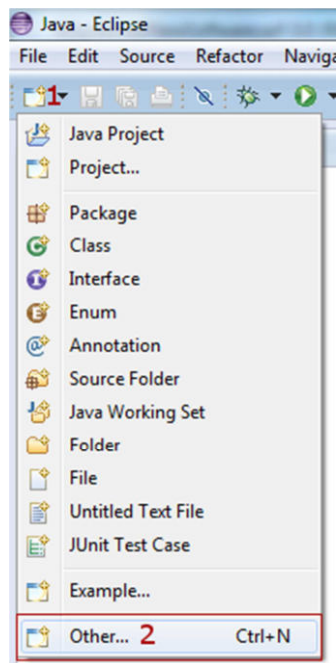


Abbildung 2: Neuer Tomcat Server in eclipse Schritt 1

1. *Abbildung 2*: Um einen neuen Server zu erstellen die **New** Schaltfläche (1) drücken und den Eintrag **Other...** wählen (2). Mit der Tastenkombination *Ctrl+N* kann das auch direkt aus der eclipse Umgebung gemacht werden.

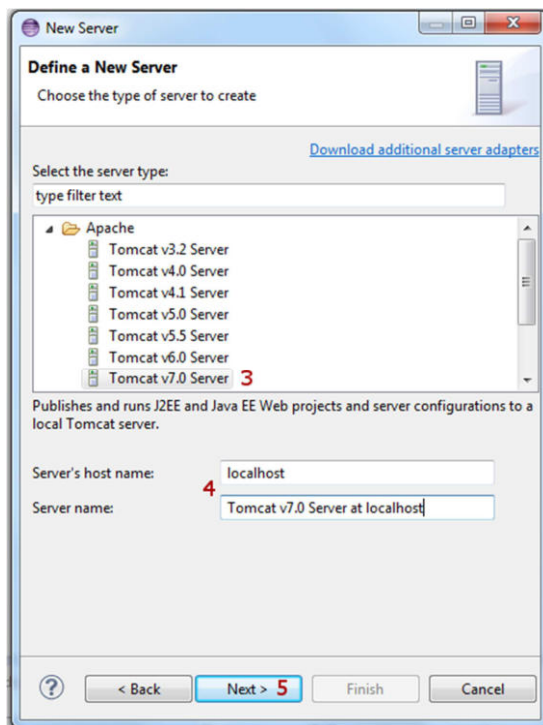


Abbildung 3: Neuer Tomcat Server in eclipse Schritt 2

2. *Abbildung 3*: Wählen Sie einen Eine Tomcat Server Verson (3). Ausserdem den host name und den Servername auswählen (4). Die Standardeinstellungen sind

ausreichend. Anschliessend auf den Button **Next >** klicken.

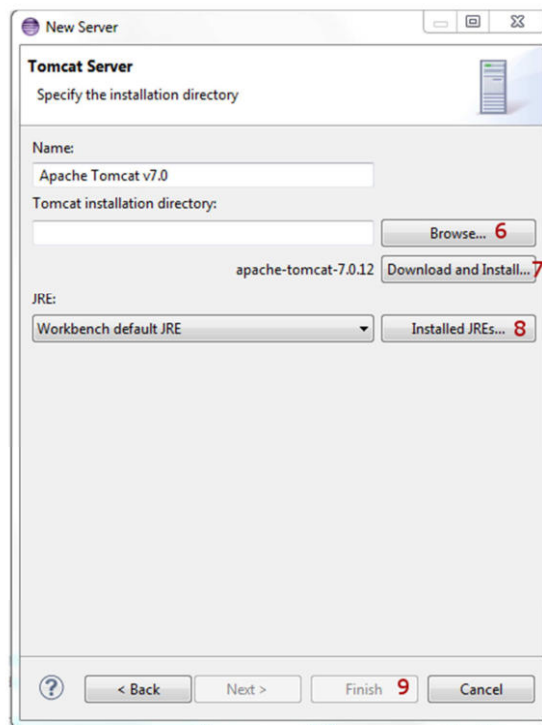


Abbildung 4: Neuer Tomcat Server in eclipse Schritt 3

3. *Abbildung 4*: Entweder unter (6) den Ordner mit der Tomcat Software Installation wählen oder mit **Download and Install...** (7) die Tomcat Software herunterladen.
 - a. Zum Installieren die Schaltfläche **Download and Install...** anklicken.
 - b. Im neu geöffneten Dialog die Lizenzbedingungen akzeptieren
 - c. Im neu geöffneten Dialog den Zielordner anwählen. In diesem Ordner wird die Tomcat Software installiert!
 - d. **Achtung**: Nachdem die Installation gestartet wird gibt es keine Fortschrittsanzeigen. Warten Sie bis die Fehlermeldung im Fenster verschwindet.
4. Bei (8) ein installiertes jre wählen. Empfohlen ist jre7.
5. Mit **Finish** wird die Installation beendet.

2 MODELLIERUNG

2.1 Vorbereitung

Als Vorbereitung im Umgang mit der actifsource-Umgebung sind die Tutorials⁴ hilfreich:

2.2 Import Basisprojekt

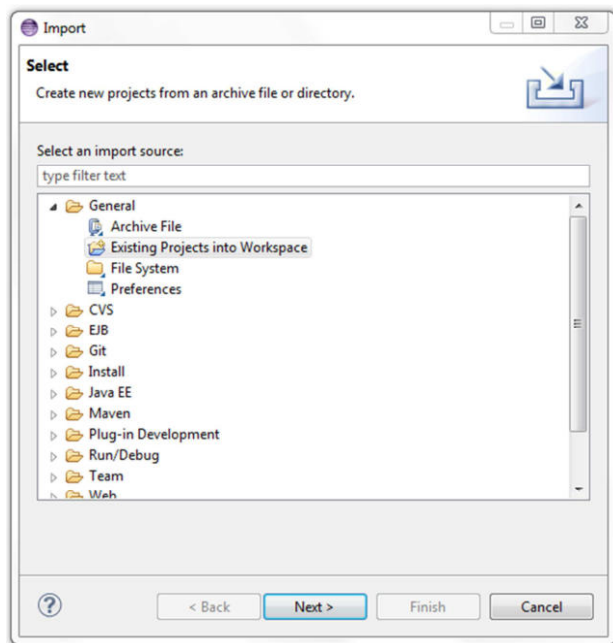


Abbildung 5: Import REST Basis Projekt Schritt 1

4. Zum Importieren die Schaltfläche **Finish** klicken.
5. Das Projekt ist jetzt in der Workspace vorhanden und kann als Basis-Projekt verwendet werden.
6. Wechseln Sie in die actifsource – Ansicht. Wählen Sie in der Rechten oberen Ecke von Eclipse die Schaltfläche **Open Perspective** und anschliessend im Dialog die Option **actifsource**. (Abbildung 7)

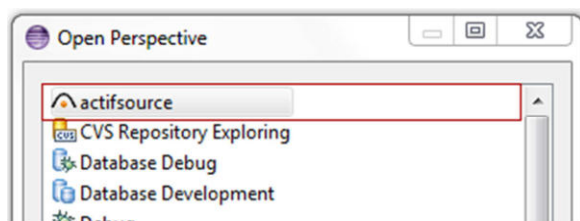


Abbildung 7: Dialog zum Öffnen der actifsource Perspektive

1. Bevor ein neues Projekt erstellt werden kann, muss das Basis-Projekt importiert werden. Über **File > Import** öffnet sich der Importdialog. (Abbildung 5)

2. Wählen Sie beim Punkt **General** die Option **Existing Projects into Workspace**. Es öffnet sich der Dialog aus *Abbildung 6*

3. Wählen In der Auswahl die Option **Select archive file**. Die Archiv-Datei des Projekts kann mit der Schaltfläche **Browse...** gewählt werden. (Abbildung 6)

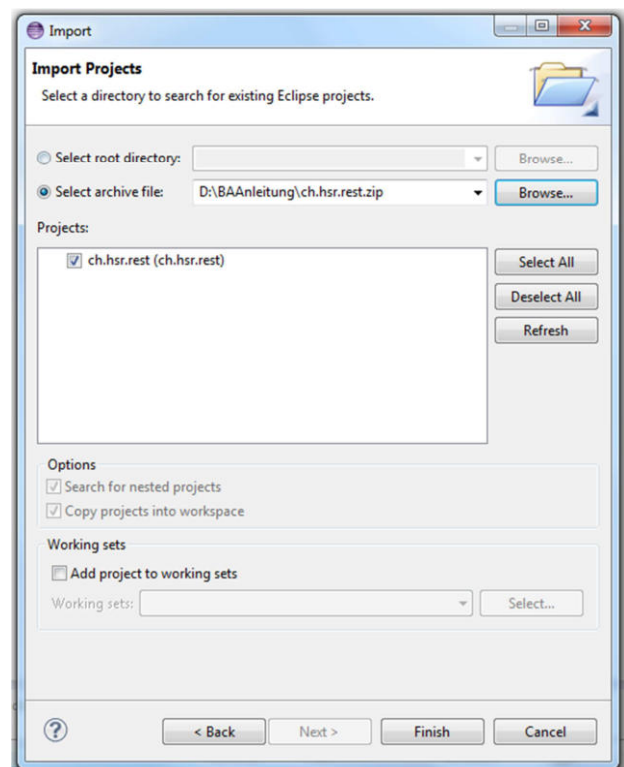


Abbildung 6: Import REST Basis Projekt Schritt 2

⁴ <http://www.actifsource.com/tutorials/index.html>

2.3 Neues Projekt erstellen

1. Unter **File > New > Project...** erstellen Sie ein neues Projekt. Im anschliessenden Dialog wählen Sie die Option **Actifsource Project** und klicken anschliessend auf **Next >**
2. Als nächstes muss ein Projekt-Name angegeben werden. Klicken sie anschliessend auf **Next >**

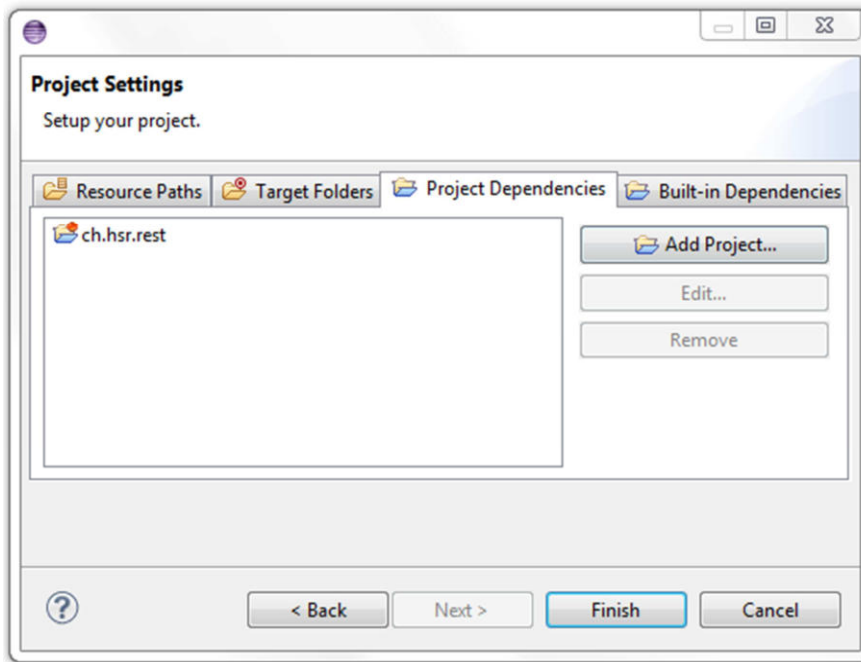


Abbildung 8: Projektabhängigkeit einstellen

3. *Abbildung 8*: Um die Modellierungssoftware verwenden zu können muss die Projektabhängigkeit konfiguriert werden. Wechseln Sie dazu in den Tab **Project Dependencies**. Über die Schaltfläche **Add Project...** wählen Sie das importierte Basis-Projekt aus Kapitel 2.2 *Import Basisprojekt* und klicken Sie anschliessend auf **OK**.

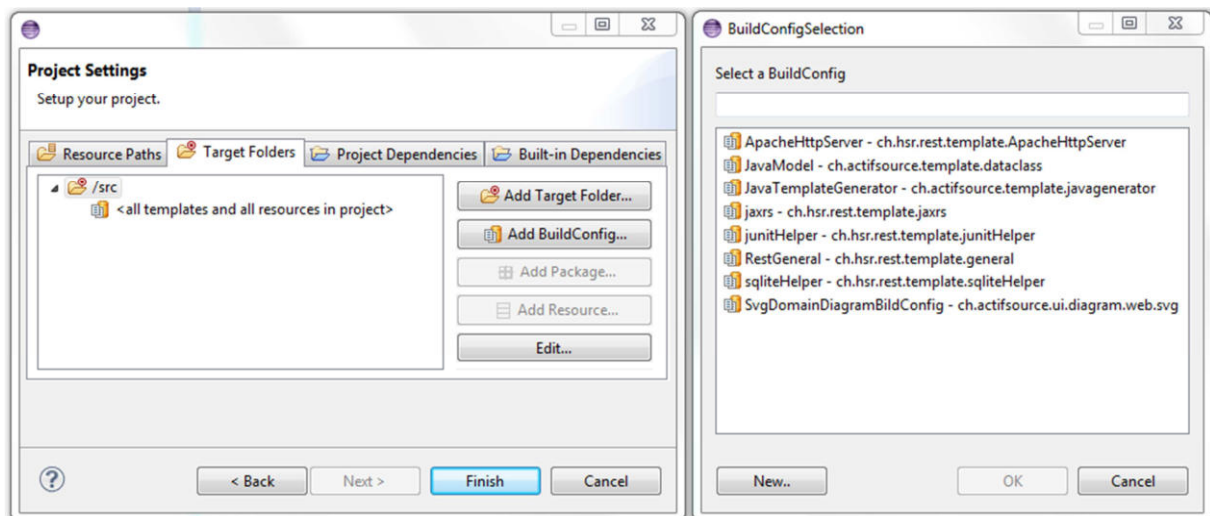
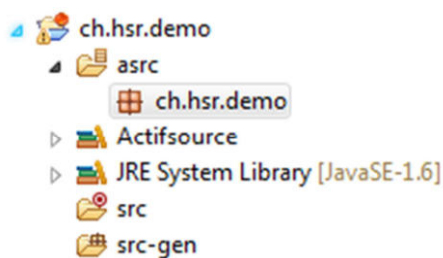


Abbildung 9: BuildConfig wählen

4. Für die spätere Code-Generierung muss ein Zielordner erstellt und die **BuildConfig** ausgewählt werden. Wechseln Sie dazu in den Tab **Target Folders**. Mit der Schaltfläche **Add Target Folder...** können Sie ein neuen Zielordner für den generierten Code erstellen. Klicken Sie im Dialog die Schaltfläche **Create Folder**. Im nächsten Dialog geben Sie einen Namen für das Verzeichnis an und bestätigen anschliessend mit **OK**. Wählen Sie den neu erstellten Ordner und schliessen den Dialog mit **OK**:
5. Nun muss noch die gewünschte **BuildConfig** gewählt werden. Aktivieren Sie dazu den Ziel-Ordner auf der linken Seite und klicken auf **Add BuildConfig...**
6. Es öffnet sich nun ein neuer Dialog. Zur Auswahl stehen diverse **BuildConfigs** zur Verfügung
7. Wählen Sie die gewünschte **BuildConfig**, entweder **ApacheHttpServer** oder **jaxrs**. Die Funktionsweise dieser beiden **BuildConfig** wird in den Kapiteln 4.1 und 4.2 genauer erklärt
 - a. Optional können noch die BuildConfig **sqliteHelper** für SQLite Unterstützung und **junitHelper** für automatisierte Tests hinzugefügt werden. Alle anderen zur Auswahl stehenden Optionen sind nicht relevant.
 - b. **Empfehlung:** Für die jUnit Unterstützung kann ein zweiter Zielordner mit der Buildconfig **junitHelper** erstellt werden.



8. Bestätigen Sie die Konfiguration anschliessend mit **Finish**

9. Das Projekt sollte nun wie in der gezeigt aussehen.

Abbildung 10: Projekt Setup im Project Explorer

2.4 Neuer Service

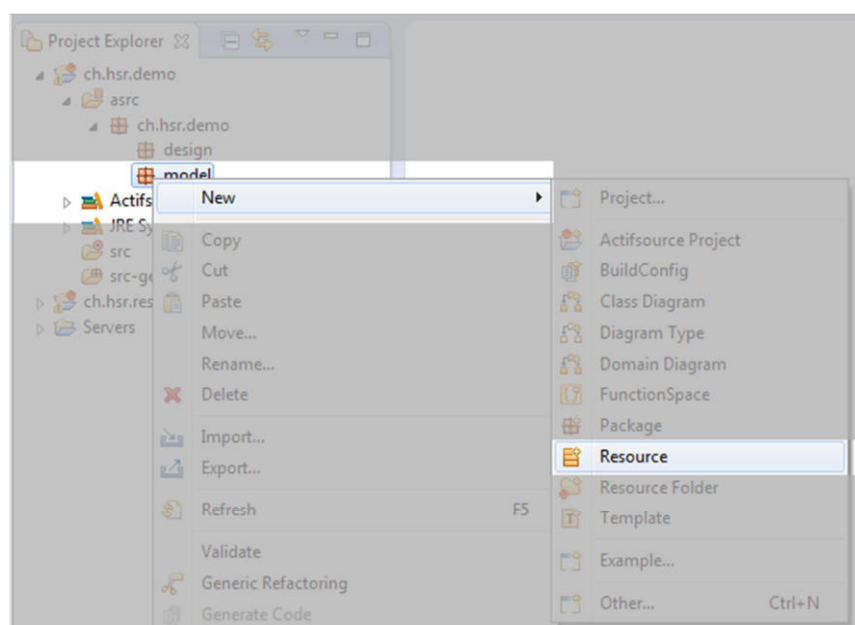
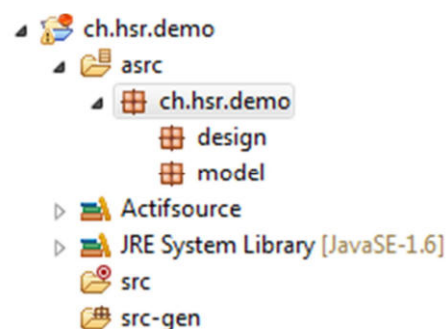


Abbildung 12: Package Setup

Abbildung 11: Neuen Service erstellen Schritt 1

1. Erstellen Sie im Projektordner zwei neue Elemente vom Typ **package design** und **model** wie in der *Abbildung 12* gezeigt.
2. Anschliessend benötigen Sie eine Service-Ressource. Das Vorgehen zeigt die *Abbildung 11*:
 - a. Rechtsklick auf das **package model**
 - b. Nach dem Auswählen der Option **New > Resource** öffnet sich ein neuer Dialog

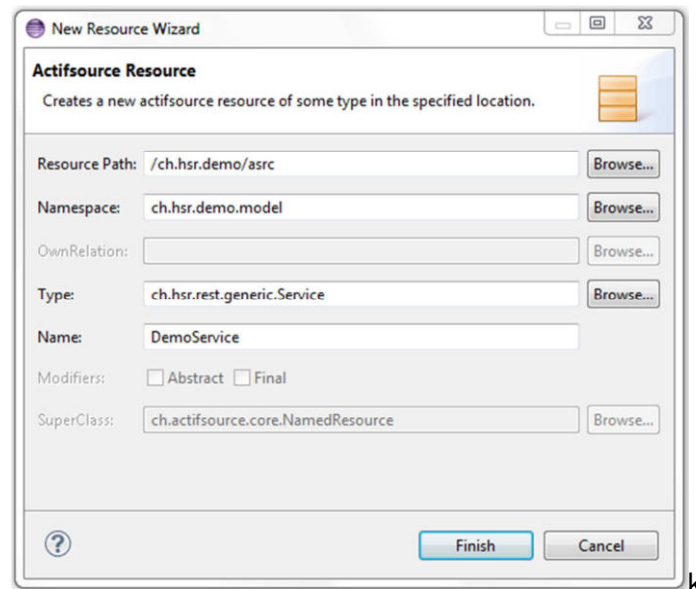


Abbildung 13: Neuen Service erstellen Schritt 2

3. Klicken Sie auf den Punkt **Browse...** beim Punkt **Type** und wählen sie den Ressourcentyp **ch.hsr.rest.Service**.
4. Geben Sie einen Namen für Ihren Service an und bestätigen Sie Anschliessend mit **Finish**. Der Service ist nun erstellt und sollte wie in der *Abbildung 14* dargestellt wird, aussehen.
 - a. Falls sich der Service nicht richtig (z.B. als XML-Darstellung) geöffnet hat. Schliessen Sie die Datei und öffnen Sie sie erneut mittels Doppelklick auf die Ressource im **Project Explorer**.
 - b. Die Fehlermeldungen können vorerst ignoriert werden.

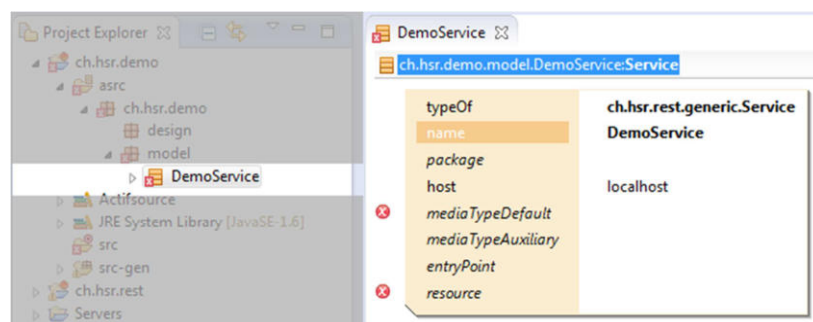


Abbildung 14: Service-Ressource nach der Erstellung

5. Geben Sie beim Punkt **package** Ihr Ziel-Java-Package an. Alle Dateien werden.

anschliessend in dieses Package generiert.

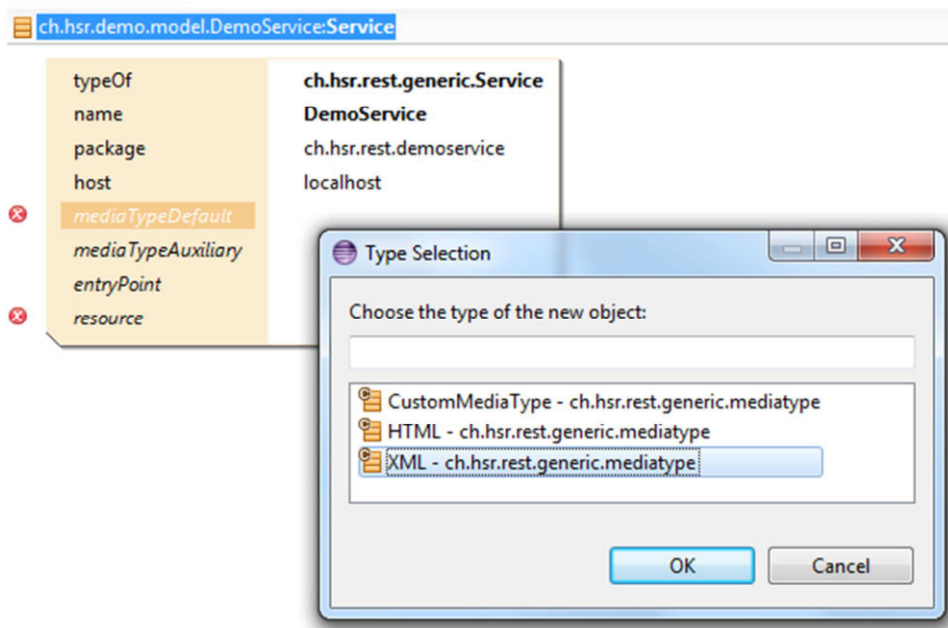


Abbildung 15: Standard-Medientyp der Schnittstelle wählen

6. Ein Doppelklick auf die Option **mediaTypeDefault** öffnet einen Dialog zum Auswählen des Standard Medientyps der Schnittstelle.
 - a. Wenn Sie einen Medientyp gewählt haben geben Sie im Feld **key** den Key des Medientyps an. Beispiel-Key für XML: *application/xml*.
 - b. Falls Sie den Typen **CustomMediaType** gewählt haben müssen Sie zusätzlich den Namen des Medientyps angeben.

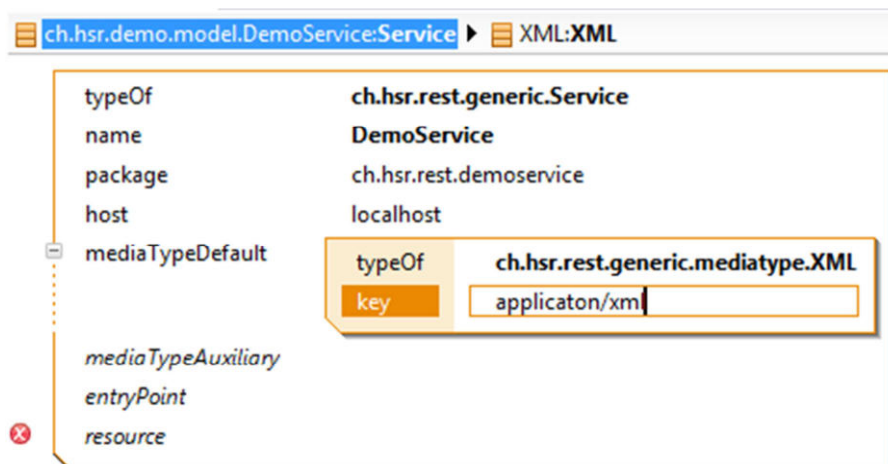


Abbildung 16: Key des Medientyps setzen

7. Speichern Sie anschliessend.
8. Sie können auf dieselbe Weise weitere Medientypen angeben. Nutzen Sie dazu das Feld **mediaTypeAuxiliary**.

2.5 Neues Diagramm

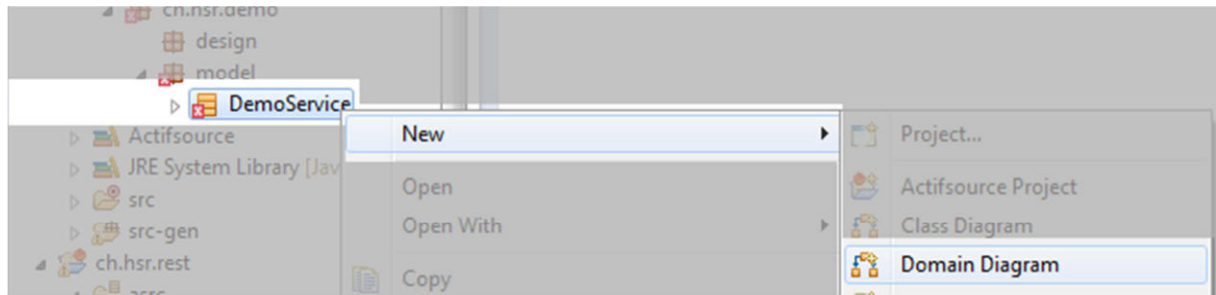


Abbildung 17: Neues Diagramm erstellen Schritt 1

1. *Abbildung 17*: Mit Rechtsklick auf die Service Ressource und anschließender Navigation zu **New > Domain Diagram**. Klicken Sie auf die Option. Es öffnet sich ein neuer Dialog um einen neues Diagramm für den Service zu erstellen.

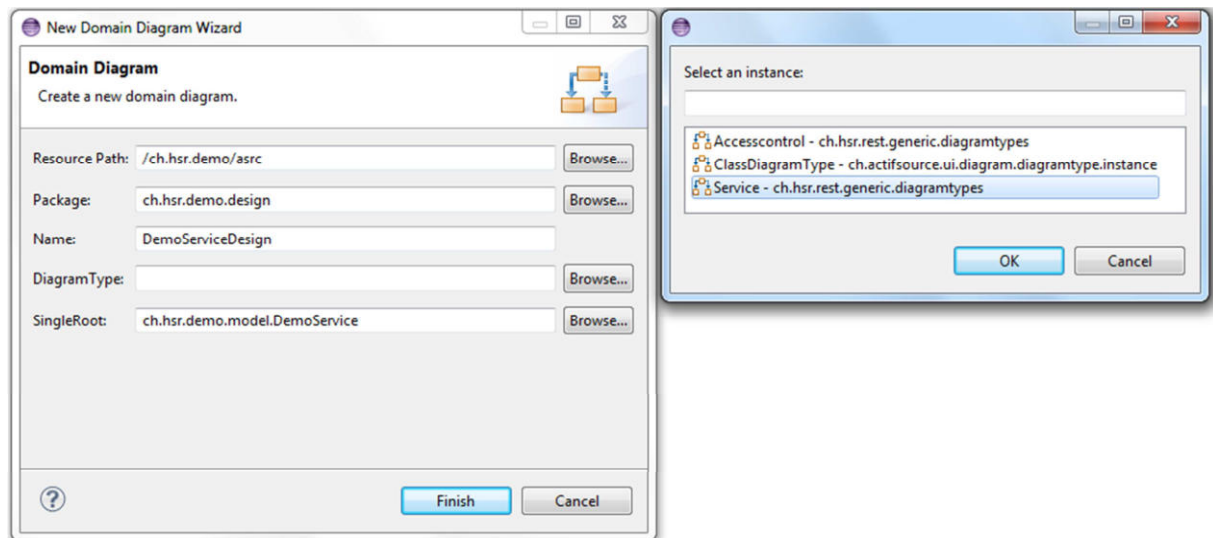


Abbildung 18: Neues Diagramm erstellen Schritt 2

2. Konfigurieren Sie nun das Diagramm wie in *Abbildung 18* dargestellt
 - a. **Package**: Wählen Sie das erstellte Package *design*
 - b. **Name**: Geben Sie einen Namen für das Diagramm an
 - c. **SingleRoot**: Achten Sie darauf, dass Ihr Service ausgewählt ist.
 - d. **DiagramType**: Über die Schaltfläche öffnet sich ein neuer Dialog. Wählen Sie aus der Liste den Diagrammtypen **Service** (*ch.hsr.rest.generic.diagramtypes*) und klicken Sie anschließend auf **OK**.
3. Bestätigen Sie anschliessend mit **Finish**.
4. Es öffnet sich ein neues, leeres Diagramm

2.6 Ressourcen und Links modellieren

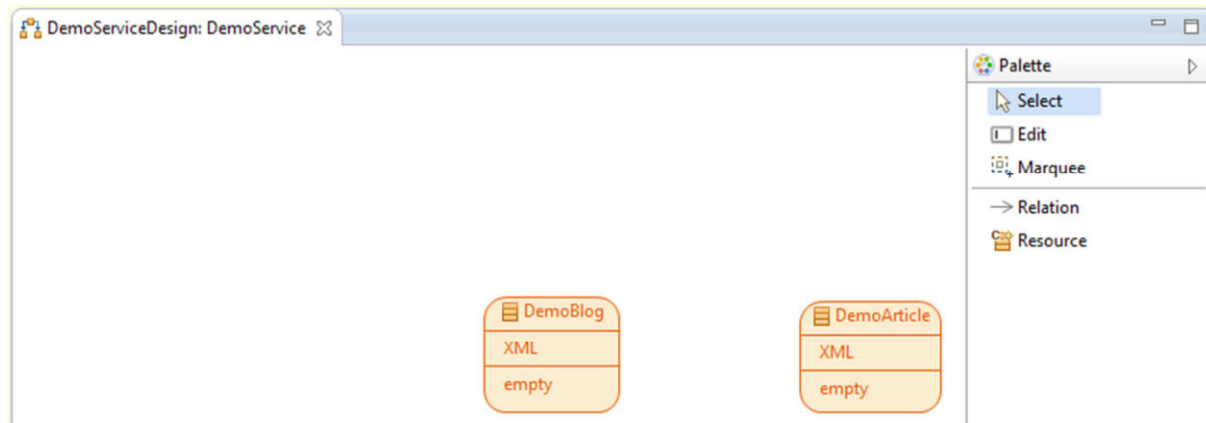


Abbildung 19: Modellieren von Schnittstellen

1. In der Palette auf der rechten Seite können Resource-Objekte gewählt werden (**Resource**) und mit einem Klick in die Hauptebene eingefügt werden. Im sich öffnenden Dialog wird der Name der Ressource angegeben.
2. Um einen Link zu modellieren, wählen Sie aus der Palette die Option **Relation**. Klicken Sie zuerst auf die Quelle des Links und danach auf die Zielressource.

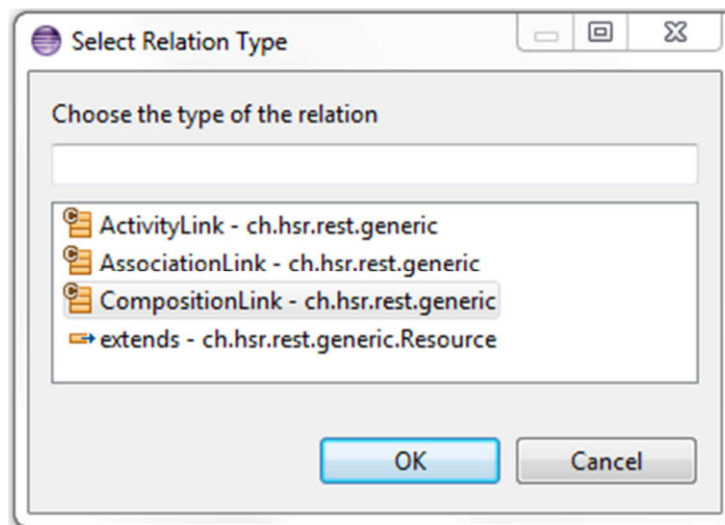


Abbildung 20: Dialog zur Auswahl des Link-Typs

3. Es öffnet sich ein Dialog wie in *Abbildung 20* dargestellt.
 - a. Wählen Sie den gewünschten Link-Typ und bestätigen Sie anschliessend mit **OK**
 - b. Geben Sie im nächsten Dialog den Namen der Beziehung an und bestätigen Sie anschliessend mit **OK**.

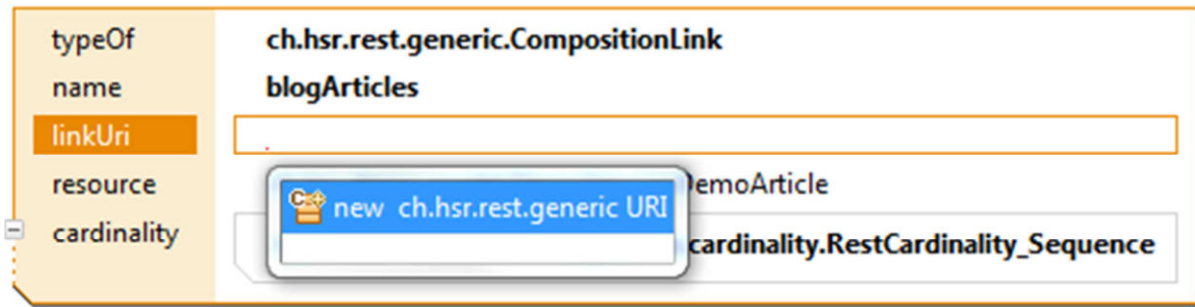


Abbildung 21: Anfügen eines neuen URI-Objektes

4. Es öffnet sich die Konfigurationsansicht des Links. Die Fehlermeldungen können vorerst noch ignoriert werden. Sie werden mit der vollständigen Konfiguration der Schnittstelle behoben.
5. Als erstes setzen Sie die Kardinalität des Links beim Punkt **cardinality**. Klicken Sie dazu doppelt auf die Feldbezeichnung. Wählen Sie im Dialog die gewünschte Kardinalität und bestätigen Sie mit **OK**.
6. Mit einem Doppelklick auf die Feldbezeichnung **linkUri** oder mit der Tastenkombination Ctrl + Leertaste im Textfeld kann eine neue URI angefügt werden *Abbildung 21*. Wie das genau funktioniert wird im Kapitel 2.7.4 *URI* erklärt.
7. Es wird bereits Code generiert. Dieser Code kann im Zielordner aus Kapitel 2.3 *Neues Projekt erstellen* angesehen werden (*Abbildung 22*).

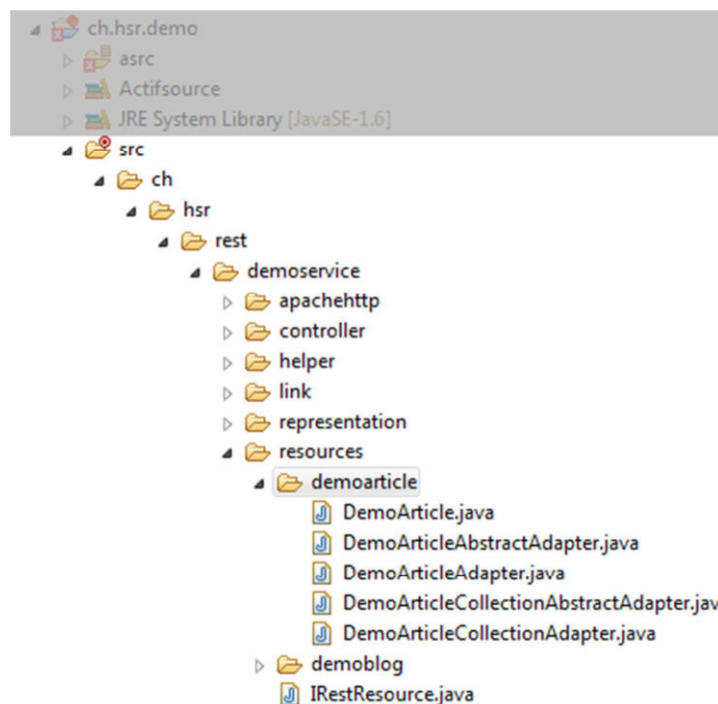


Abbildung 22: Generierter Code

2.7 REST-Schnittstelle konfigurieren

2.7.1 Ausgangslage

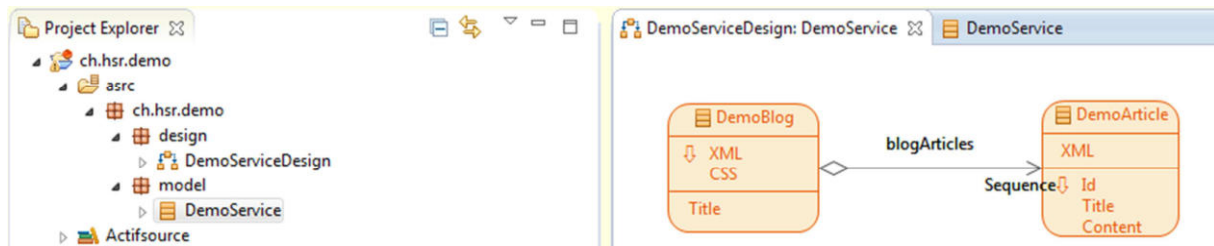


Abbildung 23: Grafisch modellierte REST-Schnittstelle

Bisher wurde die Restschnittstelle grafisch erstellt. Es gibt einen Service mit zwei REST-Ressourcen und einem Link. Wie die einzelnen Komponenten konfiguriert werden und was sie bezwecken wird in den folgenden Kapiteln beschrieben.

2.7.2 Ressourcen

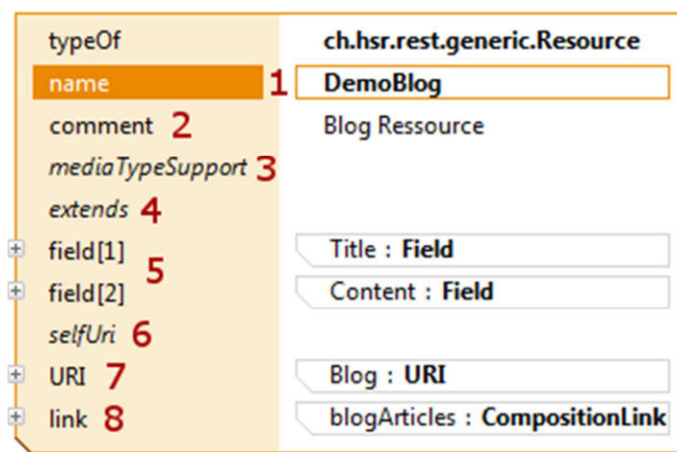


Abbildung 24: Ressource in der actifsource Editor-Ansicht

Die Nummerierung basiert auf den Nummern in der *Abbildung 24*.

1. Name der REST-Ressource. Er wird beim erstellen der Ressource im Diagramm angegeben.
2. Das Feld **comment** hat eine Informative Funktion und wird als *Tooltip* im Diagramm angezeigt.
3. Das Feld **mediaTypeSupport** gibt an, welche Medien-Typen, ausser dem Standard-Medien-Typ des Services, von der Ressource sonst noch unterstützt werden.
4. **extends** wird ausgefüllt, wenn eine Ressource eine andere erweitert.
5. Felder (**field**) sind Attribute von Ressourcen. Das Kapitel *2.7.3 Felder und Feldtypen* beschreibt die Konfiguration von Feldern.
6. **selfUri** gibt an welche URIs angezeigt werden, bei der Repräsentation einer Ressource. Die Auswahl ist auf die in der Ressource definierten URI-Komponenten beschränkt.
7. Es können beliebig viele URI-Komponenten erfasst werden. Das Kapitel *2.7.4 URI* beschreibt die Konfiguration von URIs.

8. Links werden ebenfalls grafisch erfasst (Kapitel 2.6 *Ressourcen und Links modellieren*). Das Kapitel 2.9 *Links* beschreibt die Konfiguration von Links.

2.7.3 Felder und Feldtypen

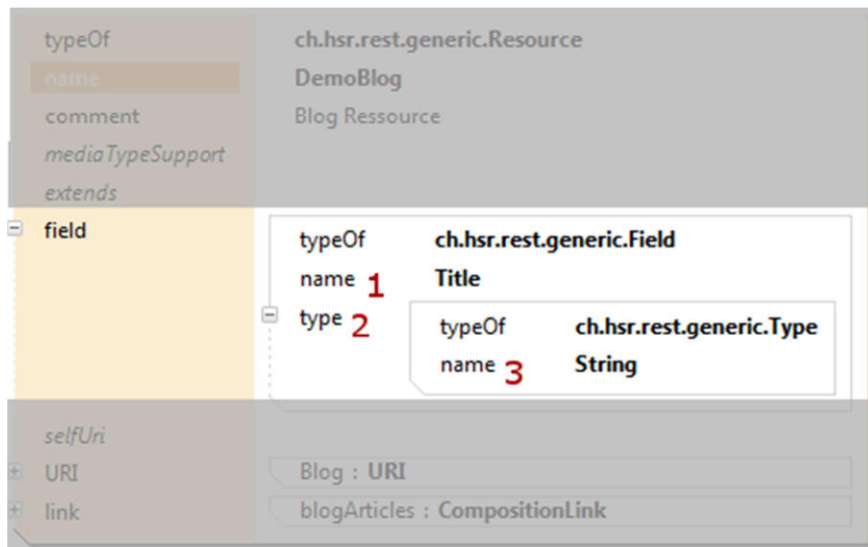


Abbildung 25: Feld in der actifsource Editor-Ansicht

Die Nummerierung basiert auf den Nummern in der *Abbildung 25*.

1. Das Attribut **name** gibt den Namen des Feldes an. Dies wird später im Code verwendet.
2. **Type** ist der Daten-Typ des Attributes.
3. Name des Daten-Typs.
 - a. **Hinweis**: für Java können die bekannten Basis-Datentypen verwendet werden
 - b. **Hinweis 2**: Für Datentypen welche in den Java-Klassen importiert werden müssen, muss der full-qualified-name, mit package-Informationen angegeben werden. Zum Beispiel: `java.util.Date`.

2.7.4 URI

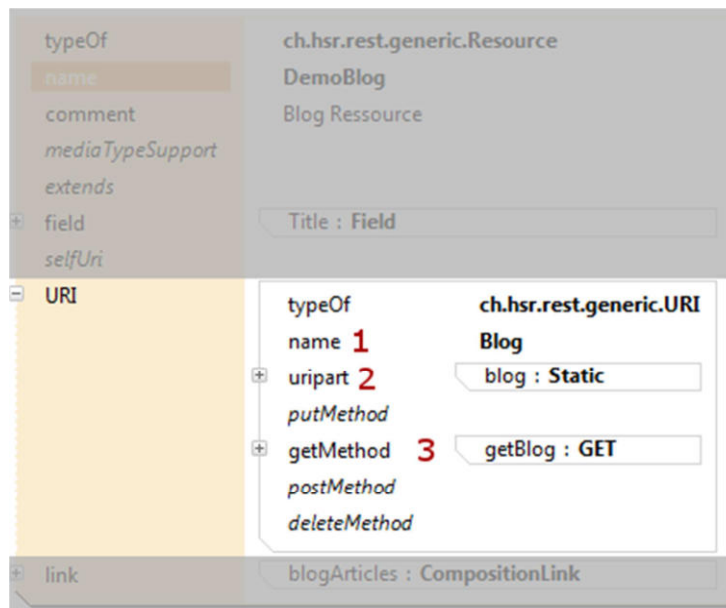


Abbildung 26: URI in der actifsource Editor-Ansicht

Die Nummerierung basiert auf den Nummern in der *Abbildung 26*.

1. Name der **URI**-Komponente. Wird für die Benennung der Controller-Klasse verwendet.
2. Ein **URI** besteht aus mehreren **URIPart**-Komponenten die eine Reihenfolge einhalten müssen. Das Kapitel *2.7.5 URIPart* beschreibt die Konfiguration von **URIPart**-Komponenten.
3. Jede URI-Komponente kann bis zu vier Methoden besitzen. Für jeden HTTP-Methoden-Typ eine. Sie müssen mit einem Namen angegeben werden. Sie werden als Methoden der Controller-Klasse generiert.

2.7.5 URIPart

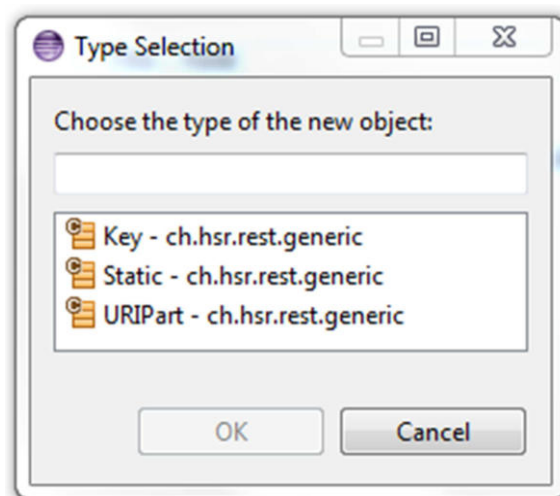


Abbildung 27: Auswahl der URI-Part-Typen

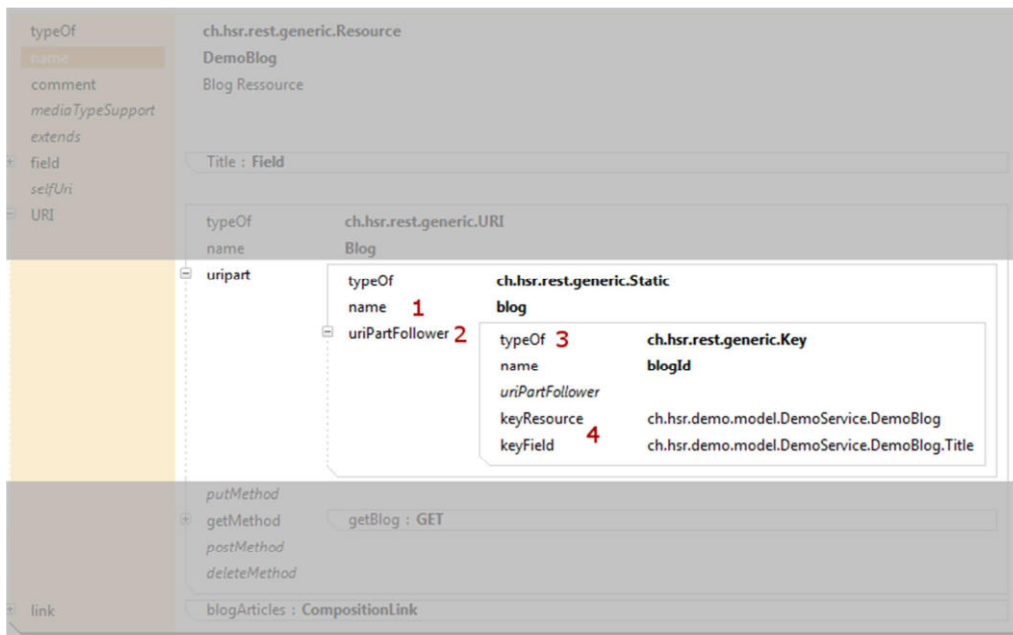


Abbildung 28: URIPart in der actifsource Editor-Ansicht

Die Nummerierung basiert auf den Nummern in der *Abbildung 28*.

1. Der **name** der **URIPart**-Komponente wird unverändert als URI-Teil verwendet.
2. Das Attribut **uriPartFollower** fügt eine weitere **URIPart**-Komponente an die URI an.
3. Es gibt zwei verschiedenen Typen von URIParts (*Abbildung 27*):
 - a. Typ **Key** (Siehe Punkt 4)
 - b. Der Typ **Static** hat nur einen Namen, welcher unverändert als URI-Teil verwendet wird.
4. Key Referenziert auf eine Ressource und benutzt den Wert im **keyField** für die generierte URI. Das **keyField** ist als Field in der Ressource definiert. Die gewünschte Ressource muss mit dem Attribut **keyResource** gewählt werden.

Hinweis: Die in der *Abbildung 28* konfigurierte URI-Komponente hat zwei Mögliche Repräsentationen.

- Als URI-Template: /blog/{blogId}
 - blogId steht als Parameter in den Controller-Methoden zur Verfügung.
- Als generierte URI in der Repräsentation: /blog/MeinBlogTitel.

2.8 HttpMethod



Abbildung 29: Erfassen von HTTP-Methoden im actifsource Editor

Jede definierte HTTP-Methode (1, 2, 4, 5) generiert eine Objekt-Methode in der Controller-Klasse. Diese haben folgenden Namen: *handle<Konfigurierter Name (3)>Request*. Folgende Punkte beschreiben den Zweck der Methode. Die Funktionalität muss von Hand entwickelt werden.

- putMethod erlaubt schreibenden Zugriff auf eine Ressource.
- getMethod erlaubt lesenden Zugriff auf eine Ressource.
- postMethod erlaubt schreibenden Zugriff auf eine Ressource.
- deleteMethod erlaubt löschenden Zugriff auf eine Ressource.

2.9 Links

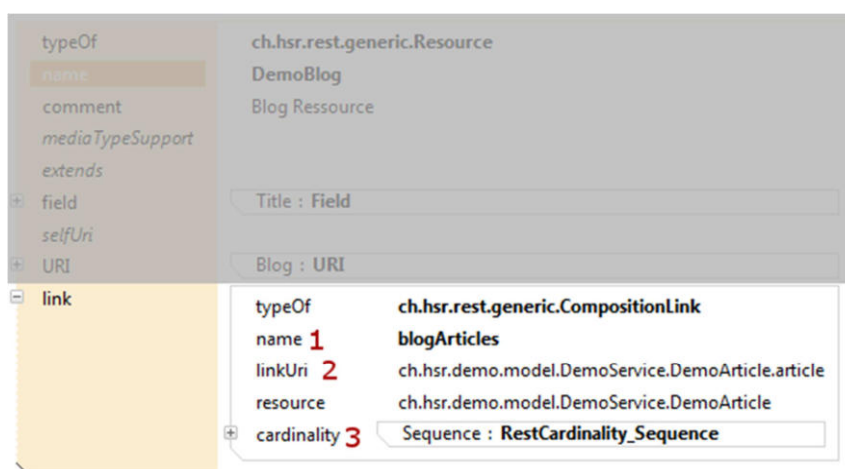


Abbildung 30: Links in der actifsource Editor-Ansicht

Die Link-Komponente bilden die Verbindung zwischen zwei Ressourcen wie sie im Kapitel

2.6 Ressourcen und Links modellieren modelliert wurden.

Die Nummerierung basiert auf den Nummern in der *Abbildung 30*.

1. Name des Links.
2. **linkUri**: Diese URI-Komponente in den Repräsentationen als Link-Uri generiert.
3. **cardinality**: Gibt die Kardinalität des Links an.
 - a. **RestCardinality_Sequence** bildet eine 0..n – Kardinalität ab. Generiert wird sie als Java – ArrayList.
 - b. **RestCardinality_IdentifiedByKey** bildet eine 0..n – Kardinalität ab. Generiert wird sie als Java – HashMap.
 - c. **RestCardinality_Field** bildet eine 0..1 – Kardinalität ab. Generiert wird sie als Objekt-Attribut der Ressource.

2.9.1 Medientypen

mediaTypeDefault	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.XML 1</td> </tr> <tr> <td>key</td> <td>applicaton/xml</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.XML 1	key	applicaton/xml		
typeOf	ch.hsr.rest.generic.mediatype.XML 1						
key	applicaton/xml						
mediaTypeAuxiliary[1]	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.HTML 2</td> </tr> <tr> <td>key</td> <td>text/html</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.HTML 2	key	text/html		
typeOf	ch.hsr.rest.generic.mediatype.HTML 2						
key	text/html						
mediaTypeAuxiliary[2]	<table border="1"> <tr> <td>typeOf</td> <td>ch.hsr.rest.generic.mediatype.CustomMediaType 3</td> </tr> <tr> <td>key</td> <td>text/css</td> </tr> <tr> <td>MediaTypeName</td> <td>CSS</td> </tr> </table>	typeOf	ch.hsr.rest.generic.mediatype.CustomMediaType 3	key	text/css	MediaTypeName	CSS
typeOf	ch.hsr.rest.generic.mediatype.CustomMediaType 3						
key	text/css						
MediaTypeName	CSS						

Abbildung 31: Medientypen in der actifsource Editor-Ansicht

Die Medientypen-Komponente gibt die Repräsentation an. Mit dem **key**-Attribut (4) ist der Medien-Typ in der HTTP-Variante erfasst.

Die Nummerierung basiert auf den Nummern in der *Abbildung 31*.

1. **XML**: Generiert die Repräsentation vollständig automatisch.
2. **HTML**-Repräsentation: Die Repräsentation muss selber implementiert werden.
3. **CustomMediaType**: Wird für eigene Medien-Typen verwendet. Die Repräsentation muss selber implementiert werden.

3 EINBINDEN VON APPLIKATIONSCODE

3.1 Controller

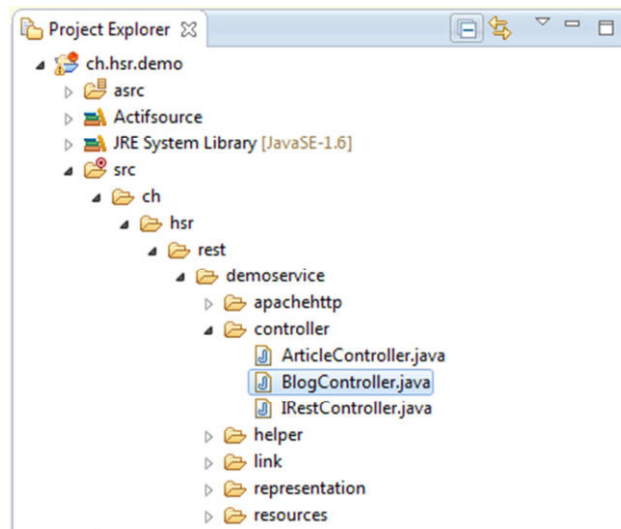


Abbildung 32: Blog Controller

Um den Applikationscode in die Schnittstelle einzubinden sind Änderungen an den einzelnen Controller-Klassen notwendig. Sie befinden sich im Verzeichnis mit dem generierten Code unter folgendem Pfad: `<packet>5/<serviceName>/controller/` (Abbildung 32).

Jeder Controller ist das Resultat der URI-Konfiguration. Die Methoden-Parameter sind wie folgt aufgebaut:

- **HashMap<String, String> parameter:** Sind die Werte, welche in den als Key-Typ definierten URI-Parts stehen. Sie sind als Key-Value Paar in der HashMap abgelegt. Der Key ist dabei der in der Konfiguration angegebene URIPart-Name.
- **HashMap<String, Object> context:** Kontext-Informationen. Dieser ist abhängig von der gewählten Technologie.
 - Bei ApacheHTTP ist darin das Medien-Typ Objekt abgelegt und kann mit dem Key „MediaType“ ausgelesen werden.
 - Bei JAX-RS beinhaltet die HashMap ein Objekt vom Typ `javax.servlet.ServletContext`⁶. Der Zugriff erfolgt mit dem Key „context“.
- **String content:** (Nur `putMethod` und `postMethod`): Der vom Client übertragene Request-Payload als String-Repräsentation.

⁵ Information stammt aus dem `package`-Attribut in der Service-Komponente (Kapitel 2.4 *Neuer Service*)

⁶ <http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html>

```

1 package ch.hsr.rest.demoservice.controller;
2
3 import java.util.HashMap;
4
5
6
7
8
9 /* Begin Protected Region [[customControllerImportArea]] */
10 1
11 /* End Protected Region [[customControllerImportArea]] */
12
13 public class BlogController {
14
15     public BlogController() {
16
17     }
18
19
20
21
22 /**
23  *
24  */
25 public IRestResource handleGetBlogRequest(HashMap<String, String> parameter,
26                                         HashMap<String, Object> context) throws RestException {
27     IRestResource resource = null;
28     /* Begin Protected Region [[02c61ed7-6672-11e3-840f-250e481af443,handleRequestHttpMethodContent]] */
29     2 // TODO handle the request and return a Resource
30     /* End Protected Region [[02c61ed7-6672-11e3-840f-250e481af443,handleRequestHttpMethodContent]] */
31     return resource;
32 }
33
34 /* Begin Protected Region [[customMethodContext]] */
35 3
36 /* End Protected Region [[customMethodContext]] */
37 }
38
39 /* Actifsource ID=[bbb67e43-4ed0-11e3-9255-998041dd62e7,b9d1a803-6662-11e3-840f-250e481af443,ff955511-666a-11e3-9255-998041dd62e7] */

```

Abbildung 33: Controller Code

Die Nummerierung basiert auf den Nummern in der *Abbildung 33*.

1. **CustomControllerImportArea**: Hier die imports der eigenen Klassen einfügen
2. **handleRequestHttpMethodContent**: Bereich für den eigenen Code. Als Resultat sollte ein Adapter Zurückgegeben werden.
3. **customMethodContext**: Falls der Bedarf besteht im Controller eigene Methoden zu schreiben, ist das der Bereich dafür.

Um eine Ressource anzuzeigen, muss sie in den Entsprechenden Adapter geladen werden und der Variable resource zugewiesen werden:

```
resource = new DemoBlogAdapter(new DemoBlog("New Blog"));
```

Alternativ kann auch eine Collection von Ressourcen als Antwort zurückgegeben werden. Analog zur einzelnen Ressource muss eine ArrayList von Ressourcen in den Collection Adapter Geladen werden:

```
// demoBlogList ist eine ArrayList<DemoBlog>
resource = new DemoBlogCollectionAdapter(demoBlogList);
```

3.2 Repräsentation

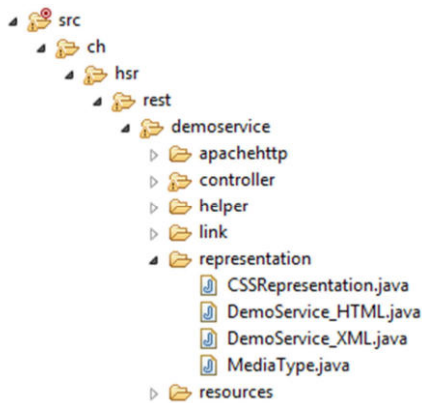


Abbildung 34: Pfad zu den Repräsentation-Klassen

Bis auf die XML-Repräsentation ist es notwendig eigenen Repräsentations-Code zu schreiben. Die Klassen befinden sich im Verzeichnis mit dem generierten Code unter folgendem Pfad: `<packet>7/<serviceName>/representation/` (Abbildung 34).

IRestResource resource: Das Interface hat drei Methoden um auf die Informationen der Ressource zuzugreifen:

- **HashMap<String, String> getFields:** enthält alle Attribut-Werte eines Objektes
- **HashMap<String, ArrayList<IRestLinkable>> getLinks:** enthält alle Links von einer Ressource zu einer anderen Ressource.
 - Die ArrayList enthält vor allem Objekte vom Typ `RESTLink` welches folgende Methoden hat:
 - **getURI:** liefert der gepasste URI als String.
 - **getMethods:** Alle Methoden die über diesen URI ausgeführt werden sind in einer HashMap abgelegt.
 - **getResourceName:** Der Name der Ressource auf die gelinkt wird
- **String getResourceName:** gibt den Namen der Ressource als String zurück.

⁷ Die Information stammt aus dem package-Attribut in der Service-Komponente (Kapitel 2.4 *Neuer Service*)


```

1 package ch.hsr.rest.demoservice.representation;
2
3 import ch.hsr.rest.demoservice.resources.IRestResource;[]
6
7 /* Begin Protected Region [[customImportContext]] */
8
9 /* End Protected Region [[customImportContext]] */
10
11 public class CSSRepresentation implements MediaType {
12
13 /**
14 *
15 */
16 public String getKey() {
17     return "text/css";
18 }
19
20
21
22 /**
23 * CSS Representation of a Resource
24 */
25 public String getRepresentation(IRestResource resource) {
26     StringBuffer representation = new StringBuffer();
27     /* Begin Protected Region [[customRepresentation]] */
28
29     /* End Protected Region [[customRepresentation]] */
30     return representation.toString();
31 }
32
33
34
35 /**
36 *
37 */
38 public HashMap<String, Object> parseRequestContent(String requestContent) throws RestException {
39     HashMap<String, Object> parsedContent = new HashMap<String, Object>();
40     /* Begin Protected Region [[customRequestParser]] */
41
42     /* End Protected Region [[customRequestParser]] */
43     return parsedContent;
44 }
45
46 /* Begin Protected Region [[customMethodContext]] */
47
48 /* End Protected Region [[customMethodContext]] */
49 }
50 /* Actifsource ID=[d3d7a345-600e-11e3-9b76-dd62ebb3c07f,b9d1a803-6662-11e3-840f-250e481af443,8ed64615-

```

Abbildung 35: Code der Repräsentations-Klasse

Die Nummerierung basiert auf den Nummern in der *Abbildung 35*:

1. **CustomImportContext**: Hier die imports der eigenen Klassen einfügen.
2. **customRepresentation**: Code um die Repräsentation zu erstellen. Der Inhalt der Repräsentation wird der Variable (StringBuffer) representation mit Hilfe der Methode append angefügt.
3. **customRequestParser**: Diese Methode parst den Request-Payload vom Client und gibt ihn im Idealfall in lesbarer Form als HashMap zurück.
4. **customMethodContext**: Falls der Bedarf besteht in der Repräsentations-Klasse eigene Methoden zu schreiben, ist das der Bereich dafür.

Hinweis: Die Implementierung der XML-Repräsentation bietet ein Beispiel, wie der Code einer eigenen Repräsentation aussehen könnte.

4 TECHNOLOGIE SPEZIFISCHE KONFIGURATIONEN

4.1 ApacheHTTP

Damit die Schnittstelle mit ApacheHTTP ausgeführt werden kann. Müssen noch kleine Konfigurationen vorgenommen werden:

Dem Projekt muss das Verzeichnis mit dem generierten und angepassten Code als Source-Verzeichnis angegeben werden. Das Vorgehen ist dabei wie folgt:

1. Rechtsklick auf das Projekt und mit die Option **properties** anwählen.
2. Auf der Rechten Seite die Option **Java Build Path** anwählen.
3. Im Tab **Source** (*Abbildung 36*) auf die Schaltfläche **Add Folder...** klicken.

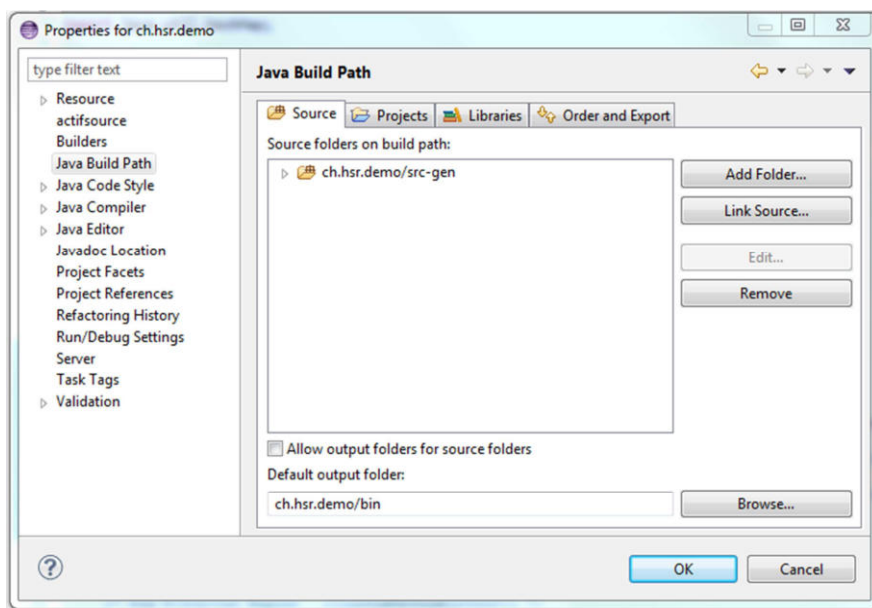


Abbildung 36: Java Build Path Konfiguration

4. Wählen Sie das Verzeichnis mit dem generierten und angepassten Code aus und bestätigen anschließend mit **OK**.

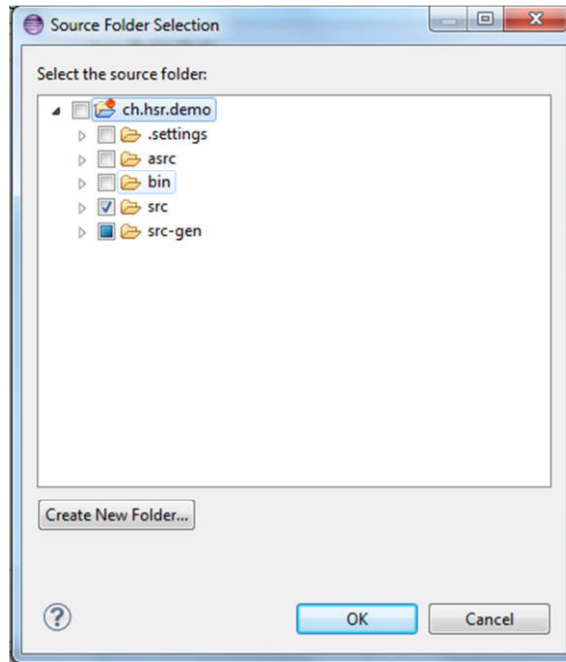


Abbildung 37: Dialog um ein Verzeichnis dem Java Build Path hinzuzufügen

5. Als nächstes muss das Projekt noch die ApacheHttp-Bibliothek einbinden. Wechseln Sie dazu in den Tab **Libraries**. (Abbildung 38)

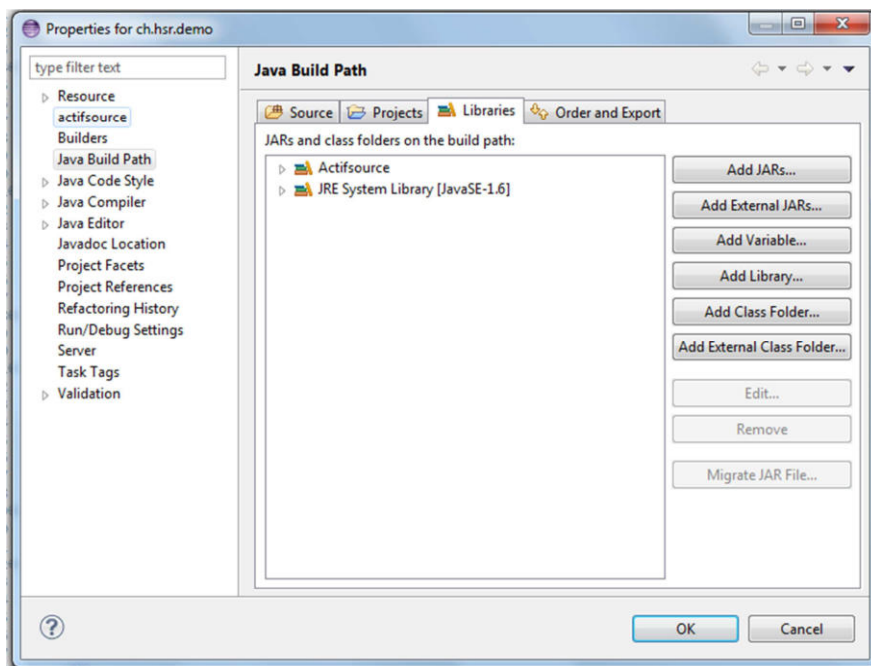


Abbildung 38: Hinzufügen der notwendigen Bibliotheken

6. Klicken Sie auf die Schaltfläche **Add JARs...** (Abbildung 38)

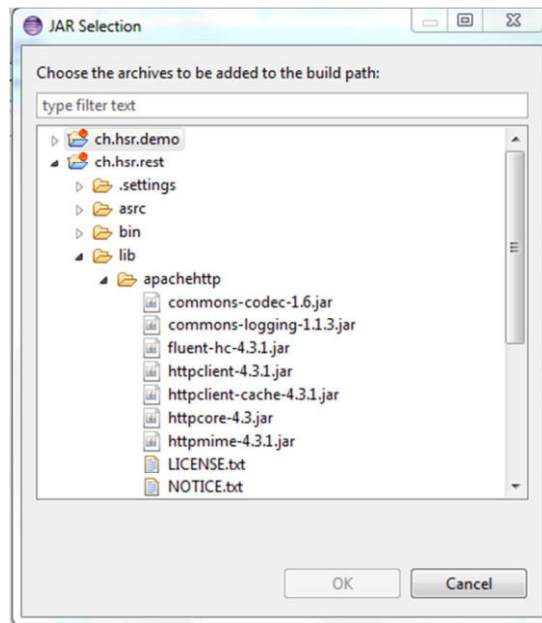


Abbildung 39: Bibliotheken liegen im Verzeichnis lib im Projekt ch.hsr.rest

7. Navigieren Sie in der Baumansicht zum Projekt `ch.hsr.rest` und darin zum Verzeichnis `lib`. Wählen sie alle `.jar`-Dateien im Unterverzeichnis `apachehttp` aus und bestätigen Sie mit **OK**. (Abbildung 39)
8. Bestätigen Sie die Konfiguration mit der **OK**-Schaltfläche.

Nun ist das die Schnittstellen-Applikation ausführbar. Navigieren Sie dazu im **Project Explorer** zur Klasse `<ServiceName>Service.java`. Mit **Run > Run as Java Applikation** kann die Applikation ausgeführt werden.

Die Schnittstelle ist mit folgender URL erreichbar: `http://[host]:[port]/[DefinierteURI]`

4.2 JAX-RS

4.2.1 Neues „Dynamic Web Project“

Es gibt zwei Möglichkeiten aus den generierten Code in einem „Dynamic Web Project“ zu verwenden. Möglichkeit eins ist über **Project > properties > Project Facet** das **Project Facet Dynamic Web Module** hinzufügen. Diese Variante ist aber nicht zu empfehlen, da die zwei Ansichten `actifsource` und „dynamic web project“ zusammen ein fehlerhaftes Verhalten aufweisen.

Die zweite, bessere Variante ist die Folgende:

1. Erstellen Sie mit **New > Project... > Web > Dynamic Web Project** ein neues Projekt. (Abbildung 40)

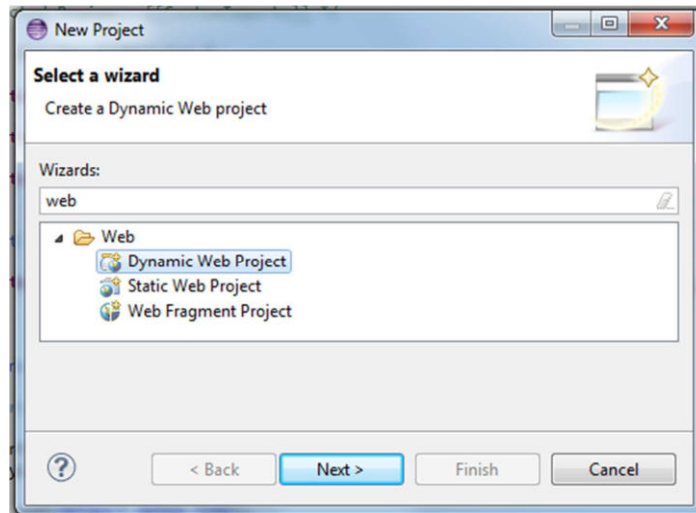


Abbildung 40: Neues Dynamic Web Project erstellen Schritt 1

2. Im nächsten Dialog muss ein Projektname (**Project name**) angegeben werden. Die sonstigen Konfigurationen können belassen werden wie sie sind. Klicken Sie auf die Schaltfläche **Next >**. (Abbildung 41)

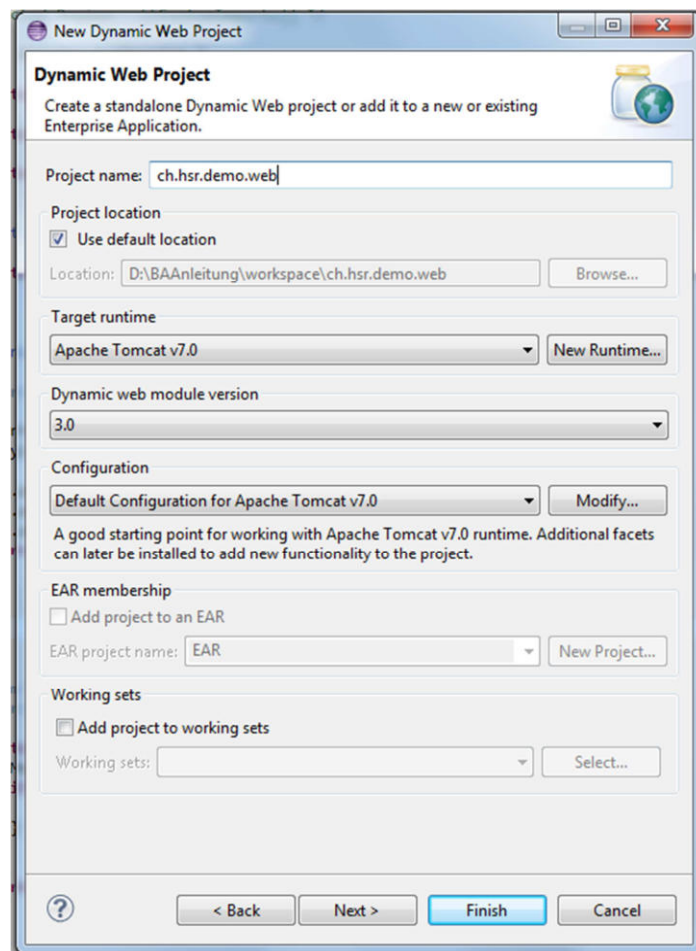


Abbildung 41: Neues Dynamic Web Project Schritt 2

3. *Abbildung 42*: Bei JAX-RS ist allen URI-Pfaden der **Context root** vorangestellt. Beispielsweise für die Blog-Ressource URI `/blog/1` muss der Pfad `demo/blog/1`

verwendet werden.

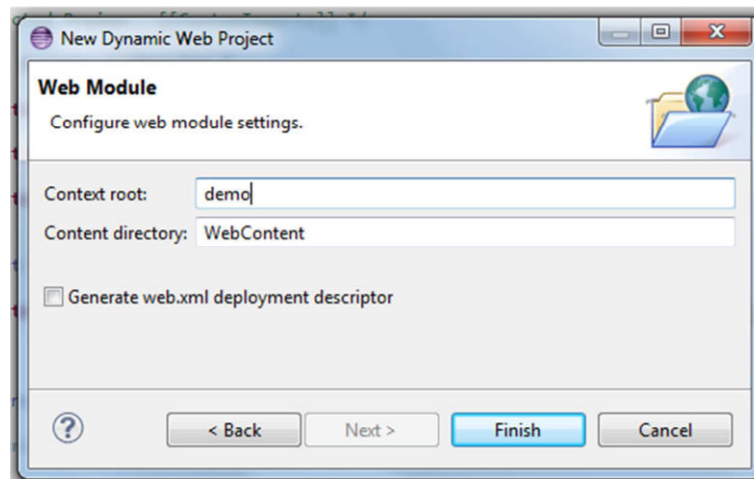


Abbildung 42: Neues Dynamic Web Project Schritt 3

4. Der Code aus dem actifsource-Projekt muss jetzt noch dem Dynamic Web Project bekannt gemacht werden. Öffnen Sie dazu die Projekt Konfiguration **Project > properties > Java Build Path** und klicken Sie die Schaltfläche **Link Source...**

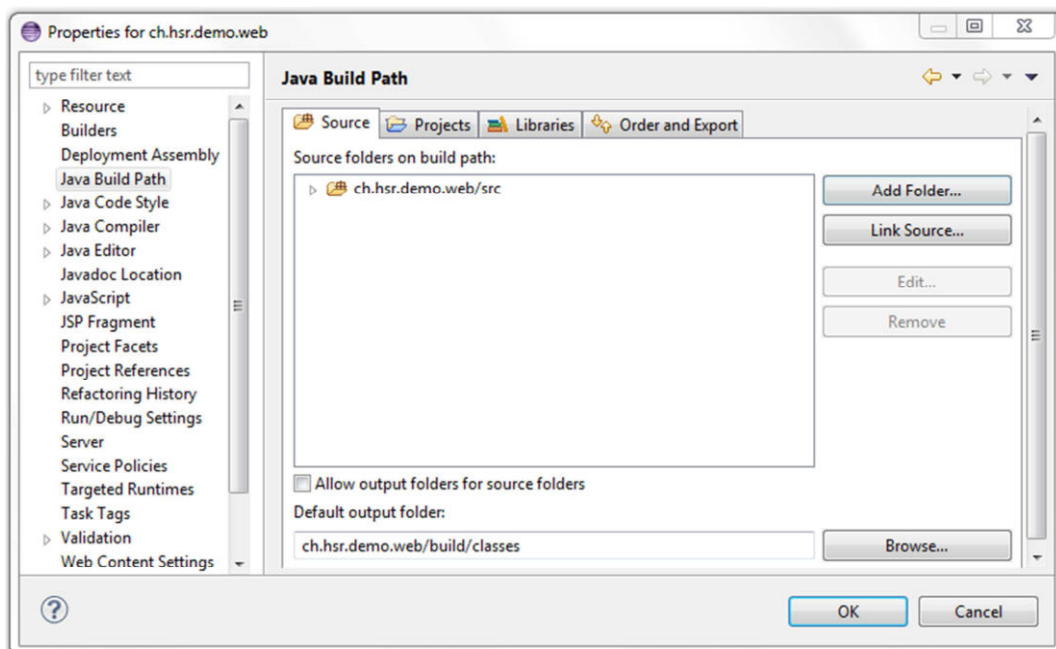


Abbildung 43: Source-Code aus dem actifsource-Projekt linken Schritt 1

5. Es öffnet sich nun der Dialog wie in *Abbildung 44* dargestellt. Im Feld **Linked folder location** muss der Pfad zum Code-Verzeichnis im actifsource-Projekt angegeben werden.
 - a. Hinweis: Benutzen Sie dazu die Pfad-Variable **WORKSPACE_LOC**. Das vereinfacht die Verwendung der Projekt-Konfiguration auf einem andern Computer. Zum Beispiel bei Verwendung eines Version Control System (VCS). (*Abbildung 44*)

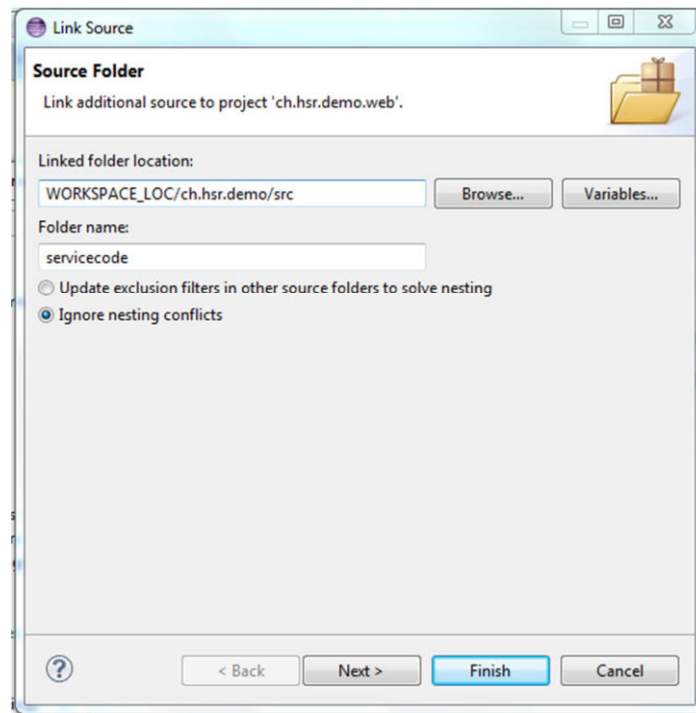


Abbildung 44: Source-Code aus dem actifsource-Projekt linken Schritt 2

6. Als nächstes müssen die JAX-RS Code-Bibliotheken importiert werden. Wechseln Sie dazu in den Tab **Libraries** und drücken Sie die Schaltfläche **Add JARs...** (Abbildung 45).

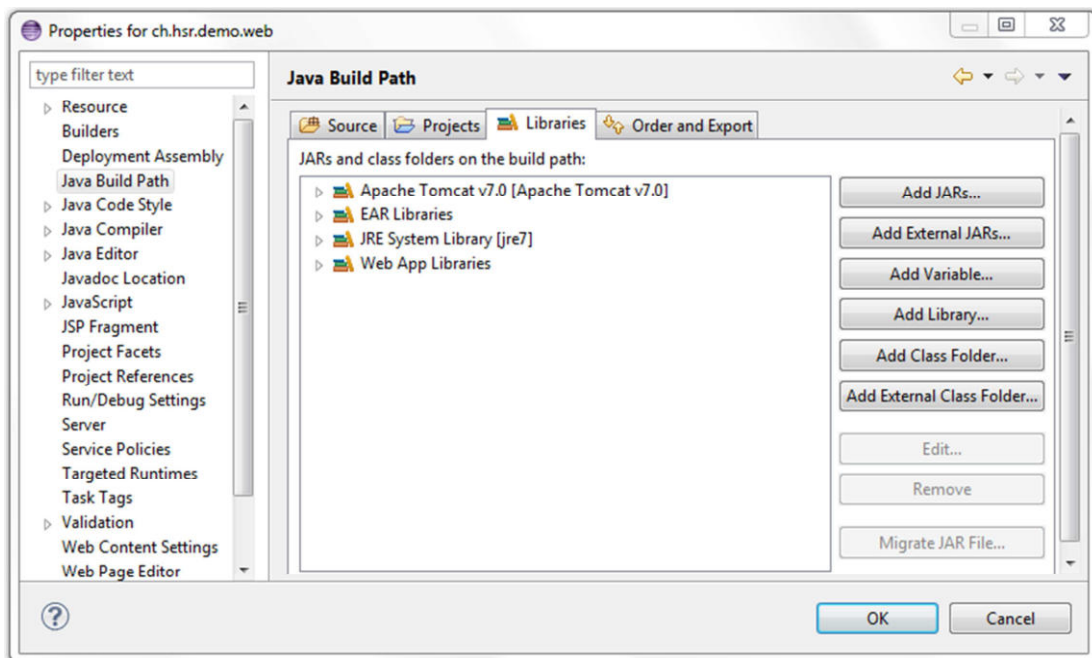


Abbildung 45: JAX-RS Code-Bibliotheken einbinden Schritt 1

7. Im jetzt geöffneten Dialog navigieren Sie zum Basis-Projekt und wählen aus dem **lib/jax-rs** Verzeichnis alle **.jar** Dateien aus. (Abbildung 47). Bestätigen Sie anschliessend mit **OK**.

8. Zum Schluss muss noch das **WEB-INF** Verzeichnis referenziert werden. Löschen Sie dazu zuerst das **WEB-INF** Verzeichnis im **Dynamic Web Project** und legen Sie mit **New > Other... > Folder** ein neues an. Im Dialog klicken Sie auf die Schaltfläche **Advanced >>**. Wählen Sie die Option **Link to alternate Location** und geben Sie wie in *Abbildung 46* gezeigt den Pfad zum WebContent-Verzeichnis des actifsource-Projektes an. Benutzen Sie dazu wieder die Pfad-Variable **WORKSPACE_LOC**.

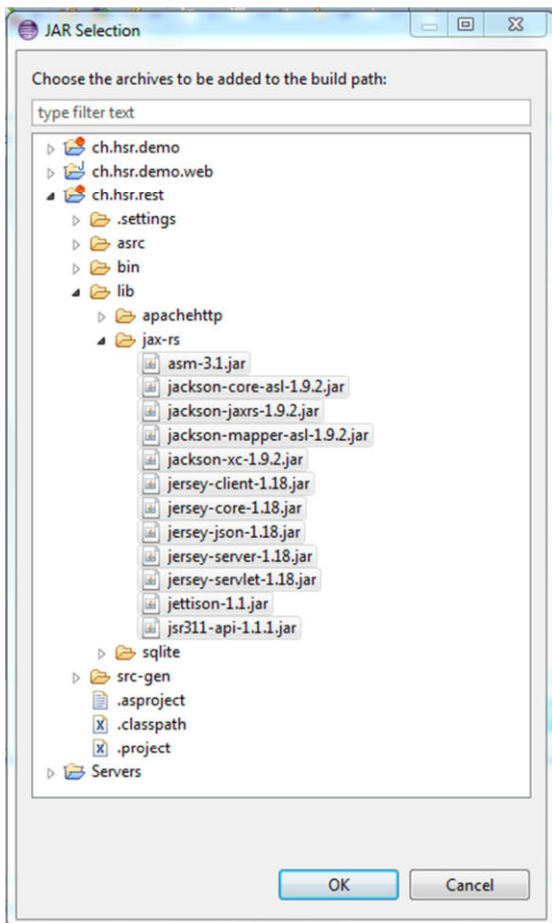


Abbildung 47: JAX-RS Code-Bibliotheken einbinden Schritt 2

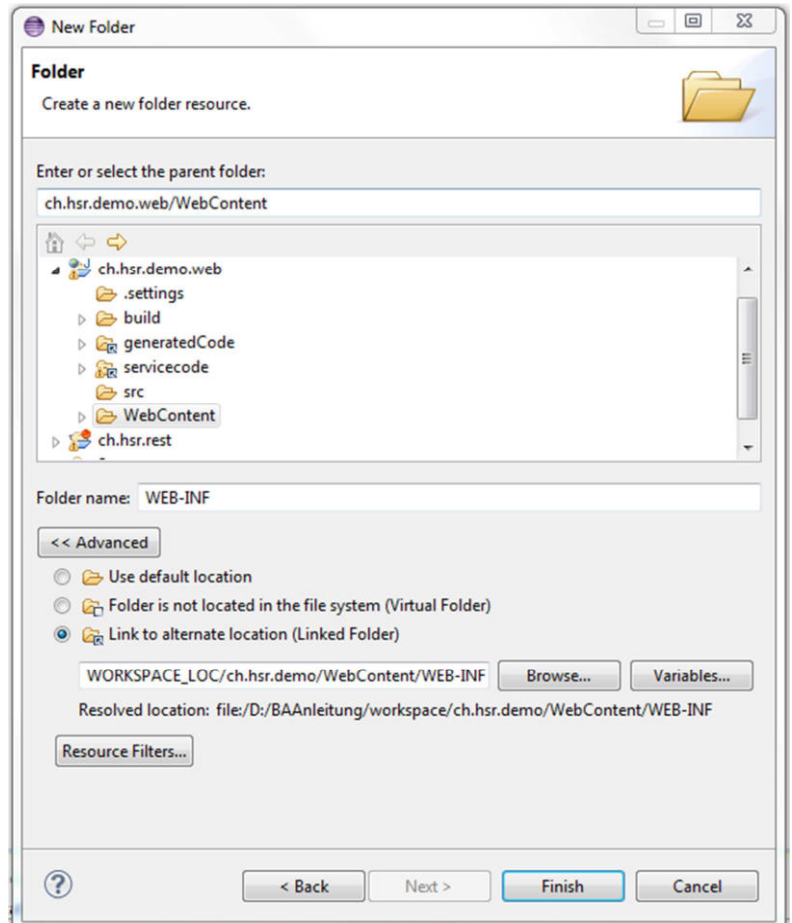


Abbildung 46: Verlinkung des WEB-INF Verzeichnisses

4.2.2 Deployment auf dem Tomcat-Server

Damit die Applikation auf dem Tomcat-Server lauffähig ist, müssen alle verwendeten Java-Bibliotheken zusätzlich ins Verzeichnis **WebContent > WEB-INF > lib** kopiert werden.

Das Projekt wird mit **Run > Run on Server** ausgeführt.

Bevor Ihr Webprojekt als .war Datei exportiert werden kann muss noch die folgende Konfiguration gesetzt werden:

1. Mit **project > project properties > Java Build Path** wird der Konfigurations-Dialog geöffnet.
2. Wählen Sie im Tab **Order and Export** alle vorhandenen Element an. (*Abbildung 48*)
3. Bestätigen Sie mit **OK**.

4. Mit einem Rechtsklick auf das Projekt und danach **export > War file** kann das Projekt nun als Archiv exportiert werden.

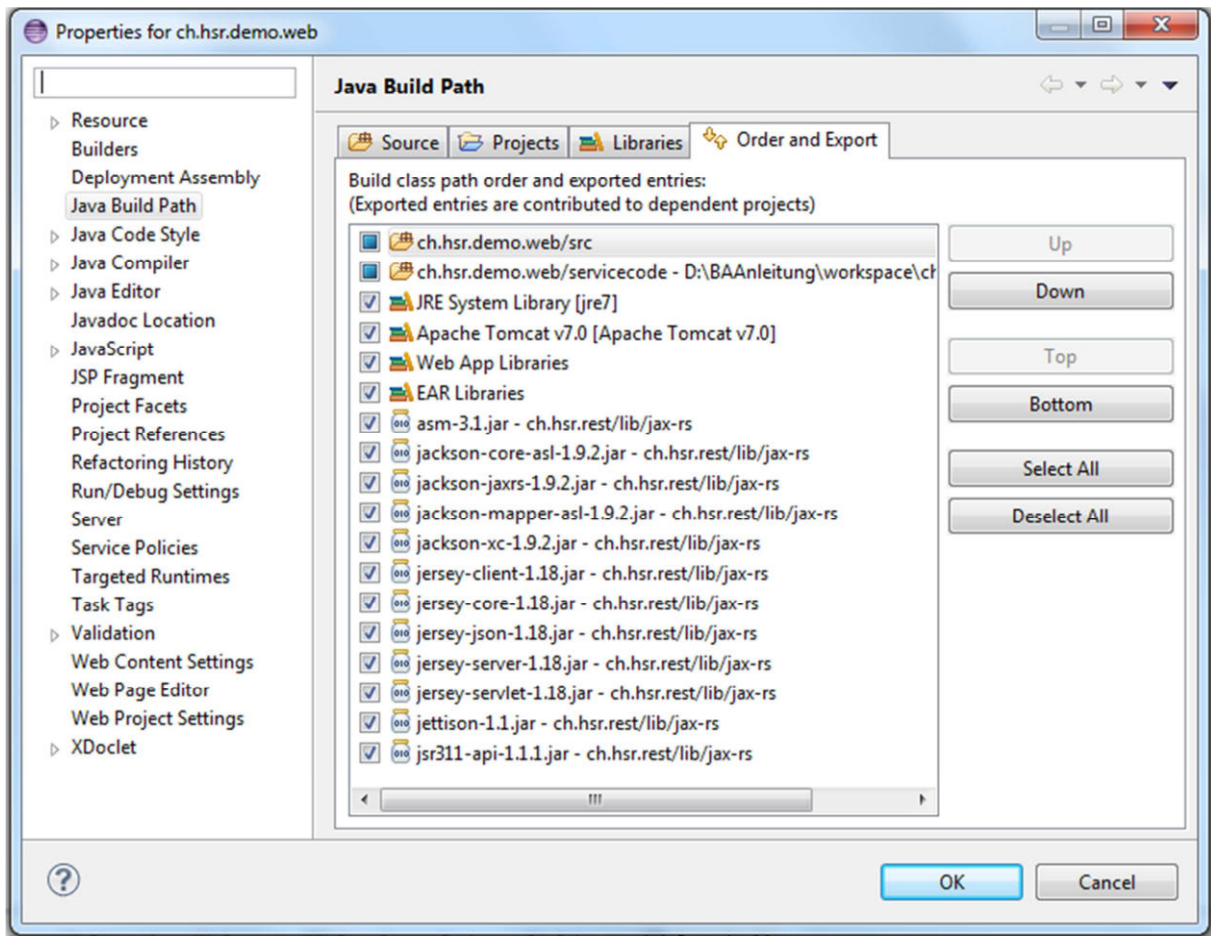


Abbildung 48: Die Order and Export Konfiguration

5 TESTEN DER SCHNITTSTELLE

5.1 REST-Client

Die Schnittstelle lässt sich am besten mit dem in die IDE PhpStorm⁸ integrierten REST-Client testen. PhpStorm unterstützt dieses Feature ab Version 7. Die IDE ist nach 30 Tagen kostenpflichtig.

5.2 Bedienung

Wenn Die IDE gestartet ist, können Sie über die Schaltfläche **Tools > Test RESTful Web Service** den Test-Client starten. Die Ansicht aus *Abbildung 49* wird geöffnet.

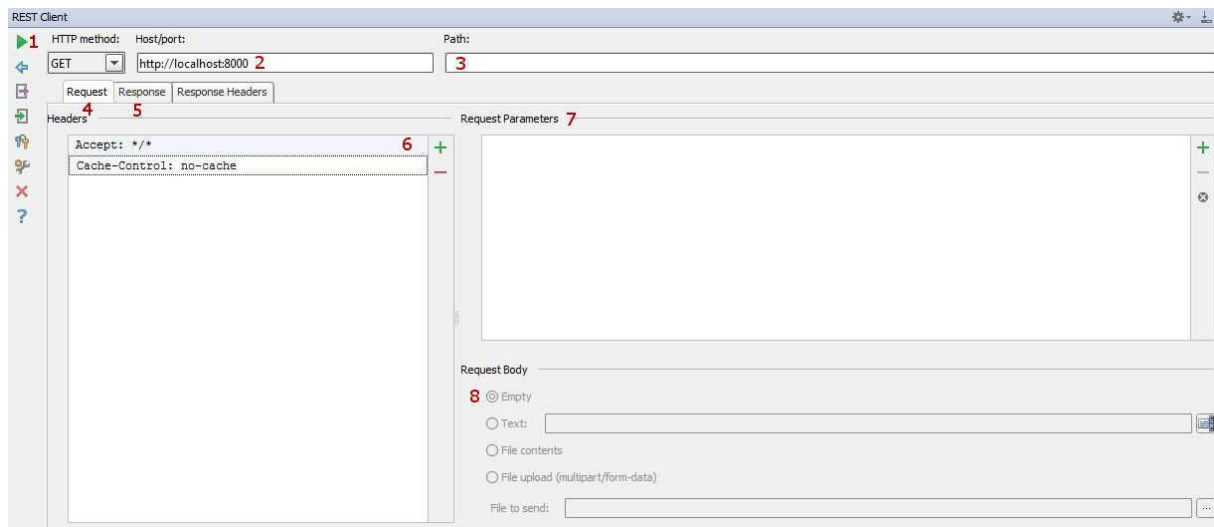


Abbildung 49: Die Rest-Client Ansicht in PhpStorm 7. Der Request-Teil

Die Nummerierung basiert auf den Nummern in der *Abbildung 49*:

1. Mit dem diesem grünen Icon wird die Anfrage gesendet. In der Auswahl rechts davon kann der Typ des Requests ausgewählt werden.
2. Hier können die Host-Informationen des Webservices angegeben werden.
3. Hier können Sie den URI-Pfad eingeben.
4. Der Request Tab zeigt die Informationen für Server-Anfragen
5. Wechselt in die Server-Antwort-Ansicht (*Abbildung 50*).
6. Hier können Header Informationen angelegt werden: Zum Anfügen dient das grüne Plus, zum Entfernen das rote Minus.
7. Hier können Sie Anfrage-Parameter erfassen.
8. Wenn die Methoden POST oder PUT ausgewählt wurden bei 1, kann hier der Anfrage-Inhalt (Payload) erfasst werden. Er kann entweder direkt (Text) oder als Datei eingefügt werden.

⁸ <http://www.jetbrains.com/phpstorm/>

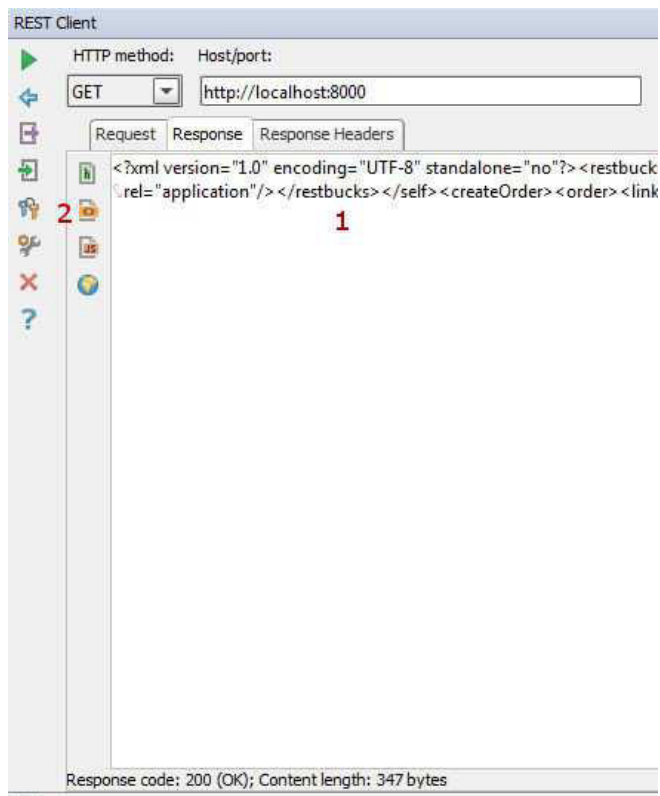


Abbildung 50: Die Rest-Client Ansicht in PhpStorm 7. Der Response-Teil

Die Nummerierung basiert auf den Nummern in der *Abbildung 50*:

1. Das Feld stellt den Inhalt der Antwort.
2. Für eine bessere Lesbarkeit, kann mit den Knöpfen die Antwort formatiert werden. Das Resultat wird anschliessend als neue Datei in die Hauptansicht geladen. Es stehen folgende Formate zur Verfügung.
 - a. HTML (grünes Icon)
 - b. XML (Gelbes Icon)
 - c. JSON (oranges Icon)

REST Hypermedia Modelling & Visualisation

Metriken

INHALTSVERZEICHNIS

Inhalt

Inhaltsverzeichnis.....	1
Inhalt.....	1
1 Einführung.....	2
2 Metriken REST-Schnittstelle DoctorService.....	3
2.1 Generierter Code mit implementierter Business-Logik.....	3
2.2 Generierter Code	4
2.3 Auswertung der Metriken.....	4
3 Metriken REST-Schnittstelle RestBucks.....	5
3.1 Generierter Code mit implementierter Business-Logik.....	5
3.2 Generierter Code	6
3.3 Auswertung der Metriken.....	6
4 Metriken REST-Schnittstelle E-Commerce	7
4.1 Generierter Code mit implementierter Business-Logik.....	7
4.2 Generierter Code	7
4.3 Auswertung der Metriken.....	8
5 Metriken RestVisualisation	9
5.1 Generierter Code mit implementierter Business-Logik.....	9
5.2 Generierter Code	10
5.3 Auswertung der Metriken.....	10

1 EINFÜHRUNG

Dieses Dokument befasst sich mit Metriken der Applikationsbeispielen. Für die Erfassung der Metriken wurde das Programm „LocMetrics“ von <http://www.locmetrics.com/> verwendet.

Die Abschnitte sind auf die vier untersuchten REST-Schnittstellen aufgeteilt. Es wurde jeweils pro Schnittstelle eine Metrik nur mit generiertem Code, sowie eine mit implementierter Business-Logik und automatisch erzeugtem Code erfasst.

In allen erfassten Metriken ist der Wert von SLOC-P (Physical Source Line of Code) entscheidend. Dieser beschreibt wie viele Source-Code-Zeilen die Java-Klassen aufweisen. Bei der SLOC-P-Kennzahl werden keine Leer- und Kommentarzeilen gezählt, nur Source-Code-Zeilen gelten.

Symbol	Definition	
	Englisch	Deutsch
LOC	Lines of Code	Gesamte Anzahl Code-Zeilen
BLOC	Blank Lines of Code	Anzahl leere Code-Zeilen
SLOC-P	Physical Executable Lines of Code	Anzahl physisch ausführbare Code-Zeilen
SLOC-L	Logical Executable Lines of Code	Anzahl logisch ausführbare Code-Zeilen
C&SLOC	Code and Comment Lines of Code	Anzahl Code mit Kommentarzeilen
CLOC	Comment Only Lines of Code	Anzahl Kommentarzeilen
CWORD	Commentary Words	Anzahl Kommentarwörter

2 METRIKEN REST-SCHNITTSTELLE DOCTORSERVICE

Dieser Abschnitt behandelt die Metriken der REST-Schnittstelle DoctorService.

2.1 Generierter Code mit implementierter Business-Logik

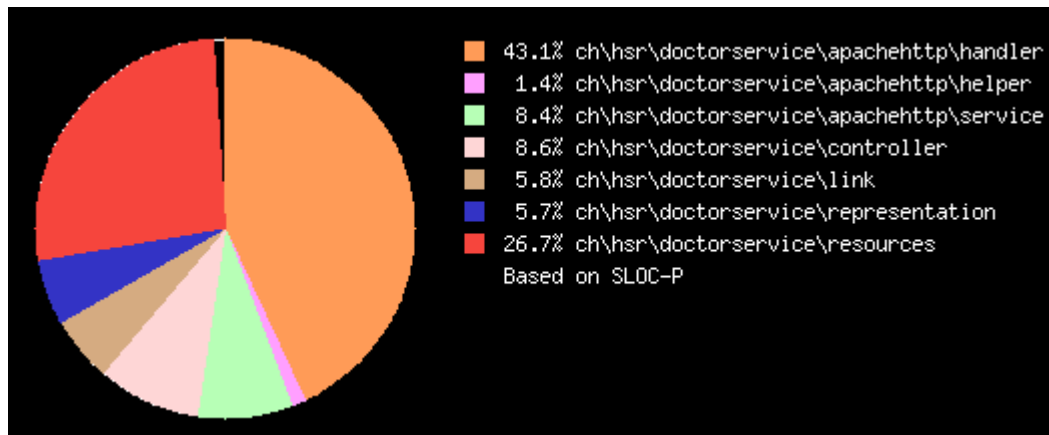


Abbildung 1: Metrik REST-Schnittstelle DoctorService - generierter Code mit erfasster Business-Logik

In der *Abbildung 1* ist die Verteilung der Source-Code-Zeile als Prozentangabe und Diagramm dargestellt. Jeder der Diagrammteile werden einem Java-Package zugeordnet. In den Werten der *Abbildung 1* sind der generierte Code, als auch die von Hand geschriebene Business-Logik der REST-Schnittstelle DoctorService enthalten. Der nicht automatisch erzeugte Programm-Code befindet sich hauptsächlich im Package „controller“.

Die grössten Bereiche im Diagramm, die Java-Packages „resources“ und „handler“ sind ausschliesslich generiert. Die Angaben, welche in der Grafik zu erkennen sind, basieren auf dem Wert SLOC-P, der nachfolgend in der *Tabelle 1* ersichtlich ist. Die zusätzlich vorhandenen Metrik-Werte sind nur der Vollständigkeitshalber aufgelistet.

Symbol	Anzahl	Definition
Source Dateien	85	Source Dateien
Verzeichnisse	23	Verzeichnisse
LOC	7656	Lines of Code
BLOC	2197	Blank Lines of Code
SLOC-P	3685	Physical Executable Lines of Code
SLOC-L	2902	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	1774	Comment Only Lines of Code
CWORD	3972	Commentary Words

Tabelle 1: Metrik REST-Schnittstelle DoctorService - generierter Code mit implementierter Business-Logik

2.2 Generierter Code

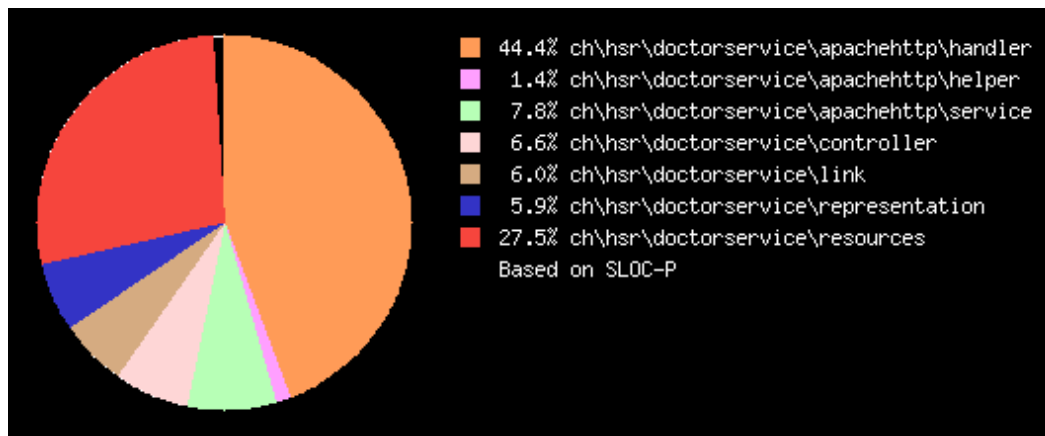


Abbildung 2: Metrik REST-Schnittstelle DoctorService: nur generierter Code

Die Darstellungs- und Berechnungsart der, in der *Abbildung 2*, visualisierten Werte, sind dieselben wie bei *Abbildung 1*. In diesem Fall ist jedoch nur generierter Code berücksichtigt.

Symbol	Anzahl	Definition
Source Dateien	85	Source Dateien
Verzeichnisse	21	Verzeichnisse
LOC	7480	Lines of Code
BLOC	2155	Blank Lines of Code
SLOC-P	3571	Physical Executable Lines of Code
SLOC-L	2796	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	1754	Comment Only Lines of Code
CWORD	3917	Commentary Words

Tabelle 2: Metrik REST-Schnittstelle DoctorService: nur generierter Code

2.3 Auswertung der Metriken

Die Auswertung der Metriken für die REST-Schnittstelle DoctorService berücksichtigt nur die Kennzahl - SLOC-P.

	SLOC-P	SLOC-P in Prozent
Generierter Code mit Business-Logik	3685	100%
Nur generierter Code	3571	96.9%
Von Hand geschriebener Code	114	3.1%

Tabelle 3: Auswertung Metriken DoctorService

Gemäss der *Tabelle 3: Auswertung Metriken DoctorService* wurden 97% des Source-Codes der REST-Schnittstelle generiert und lediglich 3% von Hand implementiert.

3 METRIKEN REST-SCHNITTSTELLE RESTBUCKS

Dieses Kapitel beschreibt die Metriken der REST-Schnittstelle RestBucks. Es besteht aus drei Unterabschnitten.

3.1 Generierter Code mit implementierter Business-Logik

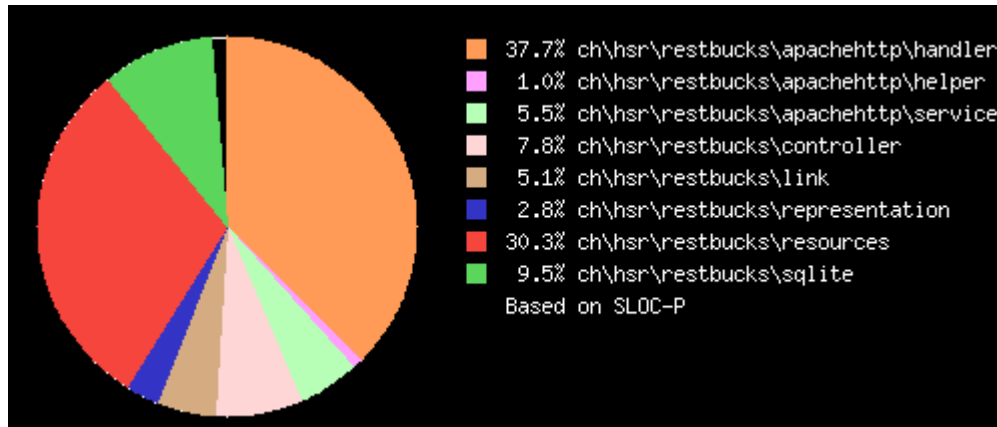


Abbildung 3: Metrik REST-Schnittstelle RestBucks - generierter Code mit implementierter Business-Logik

Die *Abbildung 3* visualisiert die Verteilung der Source-Code-Zeilen der verschiedenen Java-Packages der REST-Schnittstelle RestBucks. Es ist zu beachten, dass nur die Java-Packages „controller“, „sqlite“ und „service“ selbst verfassten Code enthält. Die Grafik basiert auf dem Wert SLOC-P.

Symbol	Anzahl	Definition
Source Dateien	119	Source Dateien
Verzeichnisse	27	Verzeichnisse
LOC	10795	Lines of Code
BLOC	3087	Blank Lines of Code
SLOC-P	5235	Physical Executable Lines of Code
SLOC-L	4123	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	2473	Comment Only Lines of Code
CWORD	5480	Commentary Words

Tabelle 4: Metriken REST-Schnittstelle RestBucks - generierter Code mit implementierter Business-Logik

Bei den Metriken der *Tabelle 4* ist, wie bei den *Metriken REST-Schnittstelle DoctorService*, zu erwähnen, dass nur SLOC-P für die Auswertung der Metriken benötigt wird. Alle übrigen vorhandenen Werte sind zur Vollständigkeit aufgelistet. Der SLOC-P dieser Tabelle, beschreibt den kompletten Schnittstellen Code der in RestBucks vorzufinden ist.

3.2 Generierter Code

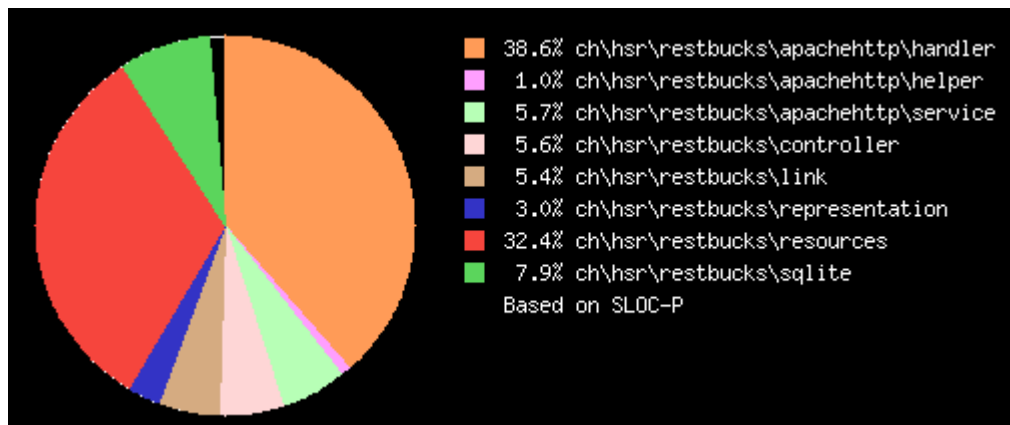


Abbildung 4: Metrik REST-Schnittstelle RestBucks - nur generierter Code

Gleich wie die *Abbildung 4*, basiert diese Grafik auf dem SLOC-P. In diesem Unterkapitel werden ausschliesslich die generierten Code-Zeilen gemessen.

Die untenstehende Tabelle zeigt die Metrik Werte vom generierten Schnittstellen-Code von RestBucks. Die wichtige Kennzahl ist farblich hervorgehoben. Der Rest der Tabelle 5: Metrik REST-Schnittstelle RestBucks - nur generierter Code ist nicht relevant für die Auswertung der Metriken.

Symbol	Anzahl	Definition
Source Dateien	117	Source Dateien
Verzeichnisse	27	Verzeichnisse
LOC	10372	Lines of Code
BLOC	3021	Blank Lines of Code
SLOC-P	4906	Physical Executable Lines of Code
SLOC-L	3864	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	2445	Comment Only Lines of Code
CWORD	5485	Commentary Words

Tabelle 5: Metrik REST-Schnittstelle RestBucks - nur generierter Code

3.3 Auswertung der Metriken

Diese Metriken-Auswertung der REST-Schnittstelle RestBucks berücksichtigt alleinig SLOC-P, der die Anzahl physikalisch vorhandenen Code-Zeilen der Beispielsapplikation zählt.

	SLOC-P	SLOC-P in Prozent
Generierter Code mit Business-Logik	5235	100%
Nur generierter Code	4906	93.71%
Von Hand geschriebener Code	329	6.29%

Tabelle 6: Auswertung der Metriken der REST-Schnittstelle RestBucks

Gemäss der *Tabelle 6: Auswertung der Metriken der REST-Schnittstelle RestBucks* wurde in etwa 93.7% des Source-Codes der Beispielapplikation generiert und nur 6.3% musste von Hand implementiert werden.

4 METRIKEN REST-SCHNITTSTELLE E-COMMERCE

Der Abschnitt Metriken REST-Schnittstelle E-Commerce erläutert die Analyse der Source-Code-Zeile dieser Schnittstelle. Diese Beispielapplikation ist gemäss Code-Zeilen die umfangreichste REST-Schnittstelle. Sie dokumentiert aus diesem Grund am besten die Zeiteinsparungen mittels Modellierens und anschliessendem Generieren des Schnittstellen-Codes.

4.1 Generierter Code mit implementierter Business-Logik

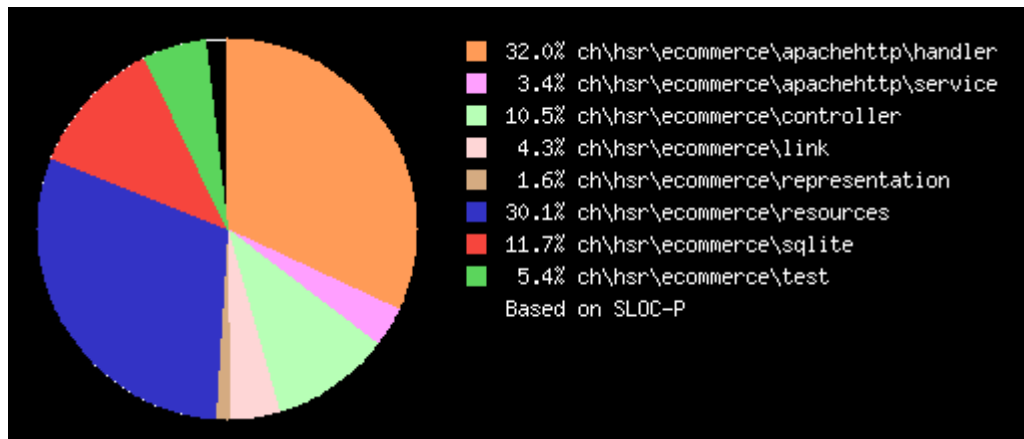


Abbildung 5: Metrik REST-Schnittstelle E-Commerce - generierter Code mit entwickelter Business-Logik

Dieses Unterkapitel veranschaulicht die Metrik des gesamten Source-Codes der REST-Schnittstelle. Das heisst, sowohl der generierte Code, als auch die von uns entwickelte Business-Logik werden in die Analyse mit einbezogen. Der Fokus der Abbildung 5 liegt auf dem Metrik-Wert SLOC-P. Der grösste Anteil von Hand geschriebenem Code befindet sich in den Packages „controller“ und „sqlite“.

Symbol	Anzahl	Definition
Source Dateien	210	Source Dateien
Verzeichnisse	36	Verzeichnisse
LOC	19419	Lines of Code
BLOC	5805	Blank Lines of Code
SLOC-P	9193	Physical Executable Lines of Code
SLOC-L	6993	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	4421	Comment Only Lines of Code
CWORD	10455	Commentary Words

Tabelle 7: Metrik REST-Schnittstelle E-Commerce - generierter Code mit entwickelter Business-Logik

Die Tabelle 7 zeigt die Metrik-Kennzahlen des gesamten Codes der Beispielapplikation E-Commerce auf. Ausschliesslich der farblich hervorgehobene Wert ist für die nachfolgende Auswertung der Metriken dieser REST-Schnittstelle von Bedeutung.

4.2 Generierter Code

Die Abbildung 6: Metrik REST-Schnittstelle E-Commerce: nur generierter Code visualisiert einzig die physisch ausführbaren Code-Zeilen des automatisch erzeugten Codes. Der

grösste Bestandteil davon machen, wie bisher die „handler“ und „resource“-Klassen aus.

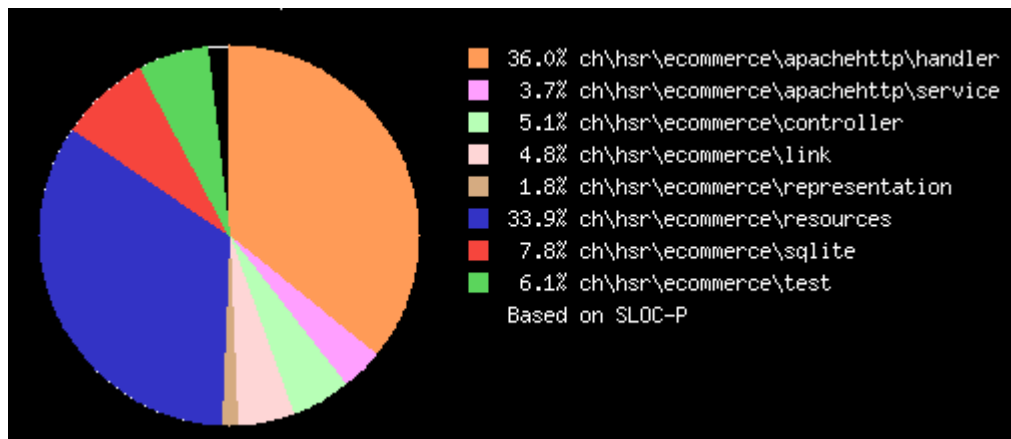


Abbildung 6: Metrik REST-Schnittstelle E-Commerce: nur generierter Code

Die nachfolgende *Tabelle 8* hebt den SLOC-P-Metrik-Wert des generierten Codes der REST-Schnittstelle hervor. Die übrigen Kennzahlen sind lediglich zu Vollständigkeitszwecken vorhanden.

Symbol	Anzahl	Definition
Source Dateien	209	Source Dateien
Verzeichnisse	36	Verzeichnisse
LOC	18126	Lines of Code
BLOC	5597	Blank Lines of Code
SLOC-P	8161	Physical Executable Lines of Code
SLOC-L	6319	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	4368	Comment Only Lines of Code
CWORD	10500	Commentary Words

Tabelle 8: Metrik REST-Schnittstelle E-Commerce: nur generierter Code

4.3 Auswertung der Metriken

Die Auswertung der Metriken für die REST-Schnittstelle E-Commerce berücksichtigt nur SLOC-P Werte, welcher die Anzahl physikalisch vorhandenen Code-Zeilen der Beispielsapplikation berücksichtigt.

	SLOC-P	SLOC-P in Prozent
Generierter Code mit Business-Logik	9193	100%
Nur generierter Code	8161	88.77%
Von Hand geschriebener Code	1032	11.23%

Tabelle 9: Auswertung der Metriken der REST-Schnittstelle E-Commerce

Gemäss der Auswertungs-*Tabelle 9* wurden bei der E-Commerce Beispielsapplikation etwa 89% des Source-Codes generiert. Bloss 11% der fast 9200 Source-Code-Zeilen wurden von Hand ausprogrammiert.

5 METRIKEN RESTVISUALISATION

Das Metrik-Kapitel der REST-Schnittstelle zeigt die Auswertung der Source-Code-Zeilen-Analyse anhand des gesamten und des ausschliesslich generierten Codes auf. Diese Beispielapplikation ist nicht, wie die anderen in diesem Dokument vorhandenen REST-Schnittstellen, mit Apache-HTTP-Server implementiert, sondern mit JAX-RS.

5.1 Generierter Code mit implementierter Business-Logik

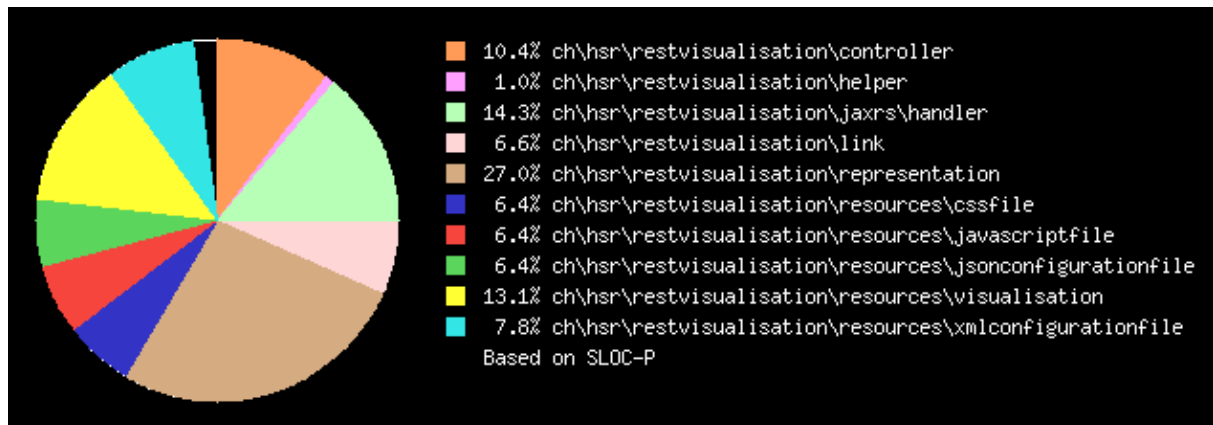


Abbildung 7: Metrik REST-Schnittstelle RestVisualisation - generierter Code mit erfasster Business-Logik

Die von Hand geschriebene Business-Logik der Beispielapplikation E-Commerce befindet sich in den Java-Packages „controller“ und „representation“. Der übrige Code wurde automatisch erzeugt. Wie schon in den vorhergehenden Metriken basiert die Abbildung 7 auf dem Metrik-Wert SLOC-P.

Die beiden Parser-Java-Klassen „MultipartParser“ und „XML2JSON“ sind nicht in den Metriken enthalten, da sie sich nicht im selben Projekt, wie der analysierte Code befindet. Zwar sind die beiden Klassen komplett von Hand implementiert, besitzen jedoch keine Bedeutung für den Aussagegehalt der Metrik.

Symbol	Anzahl	Definition
Source Dateien	47	Source Dateien
Verzeichnisse	25	Verzeichnisse
LOC	3102	Lines of Code
BLOC	979	Blank Lines of Code
SLOC-P	1415	Physical Executable Lines of Code
SLOC-L	1121	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	708	Comment Only Lines of Code
CWORD	1559	Commentary Words

Tabelle 10: Metrik REST-Schnittstelle RestVisualisation - generierter Code mit erfasster Business-Logik

Die *Tabelle 10* dieser Beispielsapplikation hebt, die, für die spätere Auswertung, wichtige Kennzahl farblich hervor. Die übrigen Werte sind für die Verwertung der Metriken RestVisualisation nicht relevant. Sie sind jedoch als ergänzende Zusatzinformationen aufgeführt.

5.2 Generierter Code

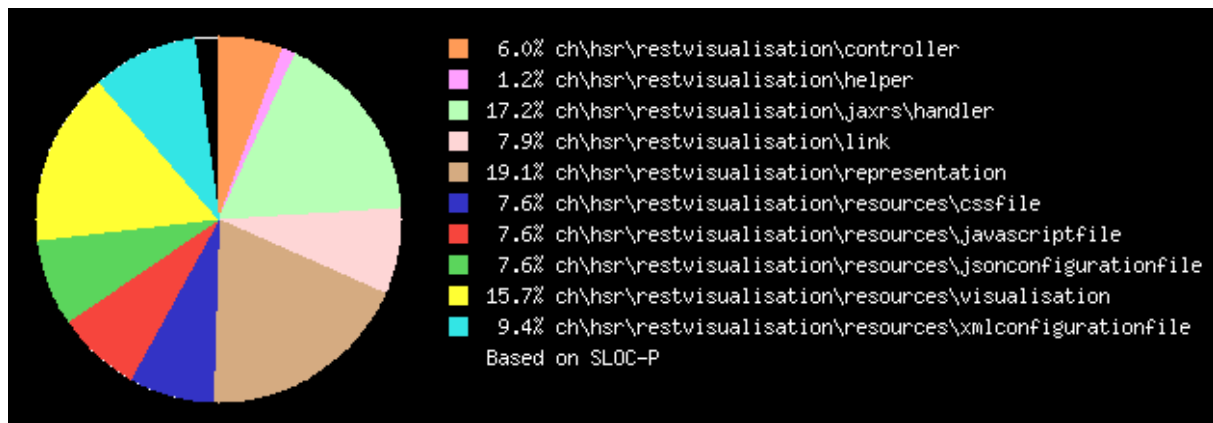


Abbildung 8: Metrik REST-Schnittstelle RestVisualisation - nur generierter Code

Das Diagramm der *Abbildung 8: Metrik REST-Schnittstelle RestVisualisation - nur generierter Code* veranschaulicht die SLOC-P Kennzahl-Auswertung für den generierten Code der Beispielsapplikation RestVisualisation.

Anschliessend sind die kompletten Metrik-Werte des automatisch erzeugten Codes zu erkennen. Farblich betont ist der benötigte SLOC-P für die Auswertung der Metriken dieses Kapitels.

Symbol	Anzahl	Definition
Source Dateien	47	Source Dateien
Verzeichnisse	16	Verzeichnisse
LOC	2789	Lines of Code
BLOC	925	Blank Lines of Code
SLOC-P	1183	Physical Executable Lines of Code
SLOC-L	927	Logical Executable Lines of Code
C&SLOC	0	Code and Comment Lines of Code
CLOC	681	Comment Only Lines of Code
CWORD	1540	Commentary Words

Tabelle 11: Metrik REST-Schnittstelle RestVisualisation - nur generierter Code

5.3 Auswertung der Metriken

Die Metrik-Auswertung der REST-Schnittstelle RestVisualisation berücksichtigt alleinig die SLOC-P-Kennzahl., Dieser zählt die physikalisch vorhandenen Code-Zeilen der Beispielsapplikation.

	SLOC-P	SLOC-P in Prozent
Generierter Code mit Business-Logik	1415	100%
Nur generierter Code	1183	83.6%
Von Hand geschriebener Code	232	16.4%

Tabelle 12: Auswertung der Metriken der REST-Schnittstelle RestVisualisation

Die Auswertungs-*Tabelle 12* veranschaulicht, dass 16.4% des Gesamt-Codes aus manuell implementierter Business-Logik besteht. Wiederum ist der Hauptteil des REST-Schnittstellen Source-Codes automatisch erzeugt worden.