Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis Nr. 33

Model-driven Code Generation for REST APIs

Markus Fischer

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Dipl.-Inf. Florian Haupt, Dipl.-Inf. Dipl.-Wirt.Ing.(FH) Karolina Vukojevic-Haupt
Commenced:	28. April 2015
Completed:	28. Oktober 2015
CR-Classification:	D.2.2, D.2.6

Abstract

In recent years *Representational State Transfer (REST)* has become more and more popular as an architecture style for web applications. An application must obey several constraints to be considered fully REST compliant. Often these constraints are only partially fulfilled by developers. These issue can be addressed by applying *Model Driven Software Development* to the design and development of REST applications, a technique that uses formal models to describe applications and to generate application code. The goal of this thesis is the generation of application code for REST APIs. For this, a REST compliant application is developed manually to identify good and practical source code templates that can be used for the code generation. The manually developed application is also used to derive entities for the formal model that provides the basis for code generation. The solution developed in this thesis defines a platform specific meta model for the generation of REST APIS. The solution also provides a transformation from an already existing meta model for REST APIs to the new platform specific meta model, and the transformation to application code. The solution is integrated into the existing modeling tool and thereby provides an fast and easy way to develop REST compliant applications.

Kurzfassung

In den letzten Jahren ist *Representational State Transfer (REST)* als Architekturstil für Web-Anwendungen immer populärer geworden. Eine Anwendung muss einige Einschränkungen befolgen, um als REST-kompatibel zu gelten. Oft werden diese Einschränkungen nur teilweise von Entwicklern erfüllt. Dieses Problem kann durch den Einsatz *Modellgetriebener Softwareentwicklung* verbessert werden. Modellgetriebene Softwareentwicklung ist eine Technik, die formale Modelle nutzt, um Anwendungen zu beschreiben und daraus Anwendungscode zu erzeugen. Ziel dieser Arbeit ist die Generierung von Anwendungscode. Zu diesem Zweck wird manuell eine REST-konforme Anwendung entwickelt, um gute und geeignete Codestrukturen zu identifizieren, die für die Codegenerierung genutzt werden können. Die entwickelte Anwendung wird auch dazu verwendet, Entitäten für das formale Modell abzuleiten. Die in dieser Arbeit entwickelte Lösung definiert ein plattformspezifisches Modell zur Modellierung von REST APIs. Zusätzlich wird auch noch eine Modelltransformation von einem existierenden Metamodell für REST APIs in das neue plattformspezifische Modell entwickelt, sowie die Generierung von Anwendungscode aus dem plattformspezifischen Modell. Die Lösung wird in das bestehende Modellierungswerkzeug integriert und stellt damit eine schnelle und einfache Möglichkeit bereit, REST-kompatible Anwendungen zu entwickeln.

Contents

List of Figures

List of Listings

1. Introduction

Representational State Transfer is an architectural style for distributed hypermedia applications that was defined by Roy Fielding in his dissertation [Fie00] in the year 2000. The REST architectural style is defined as a set of constraints and aims to improve on certain aspects of quality for software systems, like loose coupling, interoperability, performance, and scalability. *ProgrammableWeb* is a directory for "...discovering and searching for APIs to use in Web and mobile applications"[1]. Figure 1.1 depicts the development of the distribution of different protocols among the listed 5100 APIs in February 2012. The x-axis shows the years, the y-axis shows the amount of APIs compliant with a protocol. In September 2015 there are more than 14000 APIs listed on ProgrammableWeb. For the search term *REST* more than 7600 APIs were listed (Almost 2000 for *SOAP*). Figure 1.2 shows an approximation[2] of the development over the years 2012 - 2015. The axes have the same meaning as in figure 1.1. It can be seen that there are a lot of APIs claiming to be compliant to REST. However many of those APIs do not follow all mandatory constraints defined by REST. Especially constraints that have to be followed explicitly by software developers are often violated. Common "mistakes" are ignoring of the REST constraint *Hypermedia as the Engine of Application State* (HATEOAS, see 2.1), ignoring of multiple MIME Types (described in [Vit10]), the tunneling of requests through single HTTP methods (usually GET or POST) described by [Pau09], ignoring of caching, or breaking self-descriptiveness as described by [Til08].
As described there are many ways to violate the constraints of REST, which results in the loss of the before mentioned positive aspects introduced through the use of the REST architectural style. When developing a RESTful API there is usually some kind of document that describes the structure of the API in terms of resources and relationships. Once all resources are identified, the developer often has to manually implement resource classes as well as their domain specific logic. Since REST demands all resources to have a uniform interface there is a lot of repetitive coding to be done. Assuming that this results in certain patterns that can be used in most of the resources, such as answering a HTTP GET call (which every resource should do), there is a lot of copy and paste work that could be omitted when using a piece of software, that generates this repetitive code from the before mentioned design document.
Model Driven Software Development (MDSD) can achieve the latter by generating runnable application code from well defined formal models. A precondition for model driven software development is a formal model that fully describes certain aspects of a system. Since there

[1] http://www.programmableweb.com/about
[2] The annual increase was approximated since there was no data for the years in between available.

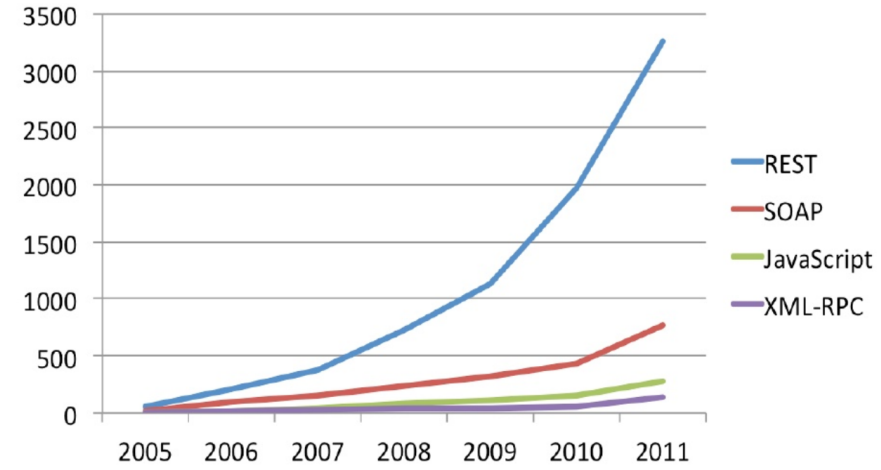


Figure 1.1.: Development of API protocol distribution [Mus12]

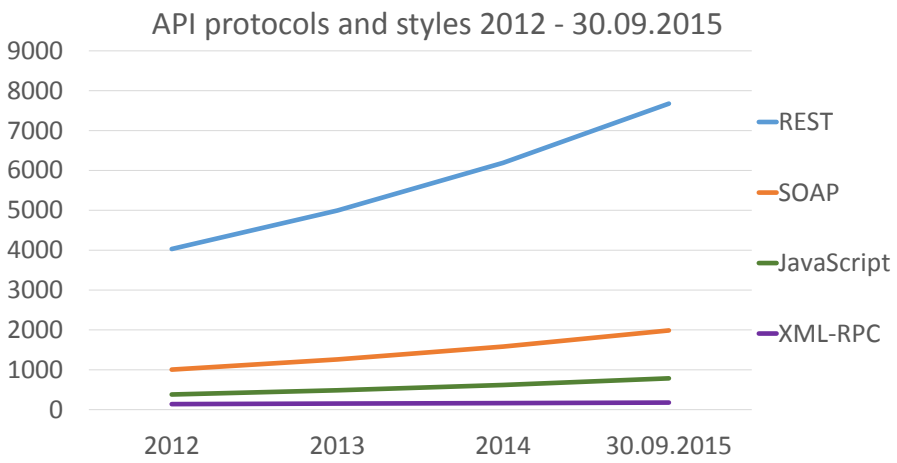


Figure 1.2.: Approximated development of API protocol distribution on programmableweb since 2012

usually is a piece of documentation about resources and their relations it comes to the task of defining a formal model that can be used as basis for code generation. This model can also be abstract and then transformed to other models which then can be enhanced with information about implementation details. Applications developed through MDSD are often a combination of generated and manually implemented code with the advantage that repetitive code can be generated each time the application changes (adding or removing resources for instance).

Motivation

This work aims to generate good and practical code for a specific runtime environment. An existing meta model for REST applications already enforces or eases the fulfilment of certain constraints of REST. The existing meta model is defined independently of any specific runtime environment (it is a *Platform Independent Model (PIM)*). A *Platform Specific Model (PSM)* shall be defined and used to generate application code. Also a *model-to-model transformation* from the PIM to the PSM shall be developed. The generated code shall be integrable in a loosely coupled fashion with manually developed code to fully benefit from the MDSD concept. Since the existing meta model is defined as part of an existing tool for REST and MDSD, the solution of this thesis shall also be integrated into the existing tool.

Outline

The thesis is structured as follows: **Chapter 2 – Background** provides information about REST, Service Oriented Architecture, Model Driven Software Development, the existing modeling tool and the existing meta models. **Chapter 3 – Solution Approach** describes the general idea behind the solution and how it has been developed. **Chapter 4 – Related Work** gives an overview of works that are related to this thesis. **Chapter 5 – Reference Application** introduces the manually developed reference application. **Chapter 6 – JAX-RS PSM Metamodel** provides details on the PSM meta model defined for code generation and **Chapter 7 – Model Transformations** gives details about the model-to-model transformation and the model-to-text transformation (code generation). **Chapter 8 – Implementation** describes the realization of the existing tool and how the solution developed in this thesis was integrated. Finally **Chapter 9 – Summary** gives a summary over the thesis and provides several suggestions for further development of the tool.

2. Background

This chapter provides information about the REST architectural style, Service Oriented Architecture, and Model Driven Software Development. This chapter also introduces the existing modeling tool and the already existing models provided by the tool that are used in this thesis.

2.1. REST - Representational State Transfer

REST is an architecture style for distributed hypermedia systems. REST was defined by Roy Fielding in his dissertation [Fie00]. The basic idea behind REST is that a server exposes resources to clients through a uniform interface. To derive REST, Fielding starts with a null style, basically an empty set of constraints so that there are no boundaries between components. Fielding then adds the following constraints consecutively:

Client-Server: The basic principle of a client-server architecture is the separation of concerns between the user interface and data storage. The user interface lies with the client while data storage is handled by the server component. This separation improves the portability of the user interface across platforms since there can be a distinct client for each platform while the server component has no need to change. This also improves the scalability of systems because the server components can be simplified. According to Fielding, the most important impact of this constraint is, that the separation allows for independent evolution of components.

Stateless: The second constraint added by Fielding restricts the interaction between client and server to a stateless communication. This means that every request made by a client has to contain all information necessary for the server to fully understand the request. So the client component has to store and provide all information about the current state of a session. This improves the reliability of the system because it becomes easy to recover from partial failures. Assuming that there are several server components providing the same functionality (for scalability reasons), it is much easier to recover from a partial failure. If one of the server component fails, requests to the failed server can be routed to the remaining components to be answered. Meanwhile the failed component can be restarted without the need to recover any session information from previous interactions. This also improves the scalability of the system, because server components do not have

the need to store session state. Also the amount of incoming requests can be balanced evenly among server components. Each message can be processed on its own, so there is also no need to manage any resources across different requests, which simplifies the implementation of server components further.
There is a to downside to the stateless constraint: Due to the fact that any session state is kept on the client, all state information required by the server has to be included in each request message sent to the server. This also means that the server loses control over the overall system, so the server has to depend on the correct implementation of clients.

Cache: Caching is a constraint that Fielding adds to improve network efficiency. Caching is the process of reusing already received response messages to future requests that are equivalent to the original request. For example if a client wants to retrieve a certain resource, and the response is cacheable, the client can use the cached answer to its retrieval request later, if the client wants to retrieve the same resource again. This implies that every response from a server has to be marked explicitly or implicitly as cacheable or non-cacheable. A disadvantage of caching is, that clients may use stale data which differs from data that would have been retrieved, if the request would have been processed by a server. Usage of stale data results in decreased reliability of a system, but can be addressed by developers, for example, by adjusting time frames for which messages are cacheable.

Uniform Interface: The uniform interface constraint is the central feature of REST. According to Fielding the emphasis on a uniform interface between components is the main difference that distinguishes REST from other network-based architecture styles. The architecture of a system is simplified by using this principle. Also uniform interfaces enables independent evolving of different system components since all components can be developed and optimized to use the uniform interface. Changes to components do not change the interface, so other components have no need to adapt every time a another component is changed.
Since the uniform interface applies to all components the overall efficiency of a system is decreased, because the information transfer between components has to be realized using standardized information formats that all components understand, instead of using efficient and specialized formats, tailored to specific components. According to Fielding the REST interface is designed to be efficient for large-grain hypermedia data transfer. The uniform interface constraint itself is again defined by four sub-constraints:

- *identification of resources*
- *manipulation of resources through representations*
- *self-descriptive messages*
- *hypermedia as the engine of application state (HATEOAS)*

To understand REST and the uniform interface constraints one has to understand the concept of a resource in REST. A resource is the basic unit of abstraction REST. Within REST a resource can be anything that can be named, like a document, a service, a process, a grouping of other resources, or other things.
Each resource in a system must be uniquely identifiable so it can be referenced. This property describes the *identification of resources* constraint. Fieldings definition of a resource is as follows:

"*A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time*".

So a resource is not a thing, but a mapping to one or more entities[1]. While the entities of a certain resource can change over time, the resource itself stays the same. An example for this would be a resource that provides the three most popular videos of the week. The videos are the entities and are likely to change over time. The resource however stays the same. Resources can be accessed and manipulated through *representations* of resources. A resource can have any number of different representations. A process for example, that is a resource in a system, may be represented through a textual representation that tells its current state. The resource could also be represented as a description of its function, or by its logo, or as a executable file to download, or other things. To interact with a resource and to manipulate a resource a client has to *manipulate resources through their representations*. So a resource is never interacted with directly but only through its representations. A representation consists mainly of data and meta-data describing the data e.g. describing the format of the data. What kind of data is contained in a message is defined through a *media type* assigned to a message.
Especially textual resource representations can contain references to other resources. If a customer browses through articles of a web shop all articles he views are most likely resources. The representations received by the customers browser will most likely not only contain information to the selected article but also to other resources, such as similar articles or supplementary articles, a link to his cart, or a link to the checkout service. This kind of references are *hyperlinks* that can be clicked by users, to retrieve the referenced information. *Hypermedia as the engine of application state (HATEOAS)* is the guiding principle for representations that contain references to other resources. The idea is that the retriever of any resource is guided through references. To order an article of a web shop a customer has to find an item he wants to buy, add this item to the cart, and then go to the checkout. The HATEOAS way to do that could be as follows:
A customer uses his browser to navigate to the homepage of a web shop. On the front page of the web shop are several hyperlinks to special offers. After clicking on one of

[1]The Hypertext Transfer Protocol (HTTP) [htt99] also knows entities, but entities in HTTP are the payloads of messages

the hyperlinks the customer receives information about the items and he decides to buy that item. He can do this by clicking on a link titled *add item to cart*. This click performs an update of his cart (which also is a resource) and retrieves a representation of the cart (a list of his selected items). Embedded in his carts representation is another hyperlink that says *checkout*. By clicking the checkout the cart resource is updated (now empty again). The customer is then guided to give his address and payment information and the purchase is completed. The fourth constraint of the definition of the uniform interface are the *self-descriptive messages*. Self-descriptive messages mainly result from the earlier discussed *Stateless* constraint in that a message must contain all information that is needed for a receiving server to process the message. To be self-descriptive each resource is able to understand a set of *standard methods*. Standard methods have a standardized semantic meaning. Standard methods and media types are sufficient to indicate the semantic and content of a request, thus forming the uniform interface for all resources.

Layered System: The layered system constraint is another constraint that aims to improve the system scalability. The idea is that there may not only be client and server components but also intermediaries, ordered in a hierarchical manner. Layers can only see other layers with which they interact, but may not see beyond that layer, e.g. a client requesting a resource might interact with a load balancer and doesn't know that its request is processed in the layers after the load balancer. In a layered system, servers can themselves be clients to other servers. An example for a system with five layers can be seen in figure 2.1. The examples shows a system with caches at server-side and client-side and a load balancer that tries to evenly distribute all incoming requests among the server components. The usage of intermediaries such as load balancers can also improve the systems scalability.

Code-On-Demand: The final constraint of REST is the code-on-demand constraint. Clients can extend their functionality by downloading applets or scripts. The goal is to simplify clients and improve the extensibility of the system even after deployment. Fielding defines this constraint as optional since it also makes the overall system more complex.

2.1.1. REST and the Web

REST is considered to be the architectural style of the World Wide Web. For simplification the World Wide Web will be called *WWW* in the following chapter. The chapter describes, what technologies the WWW uses to realize a system compliant to the REST architectural style. The WWW obviously uses a client-server architecture since all resources on the WWW are offered by servers and accessed by clients, mostly browsers. According to Fielding, the most significant benefit of REST is, that components can evolve independently. The Code-on-demand aspect is also fulfilled since browsers can extend their functionality easily through addons, or by executing scripts. The WWW uses Hypertext Transfer Protocol (HTTP [htt99]), media types (specified through Multi Purpose Mail Extension (MIME)[mim96], often called MIME-Types),

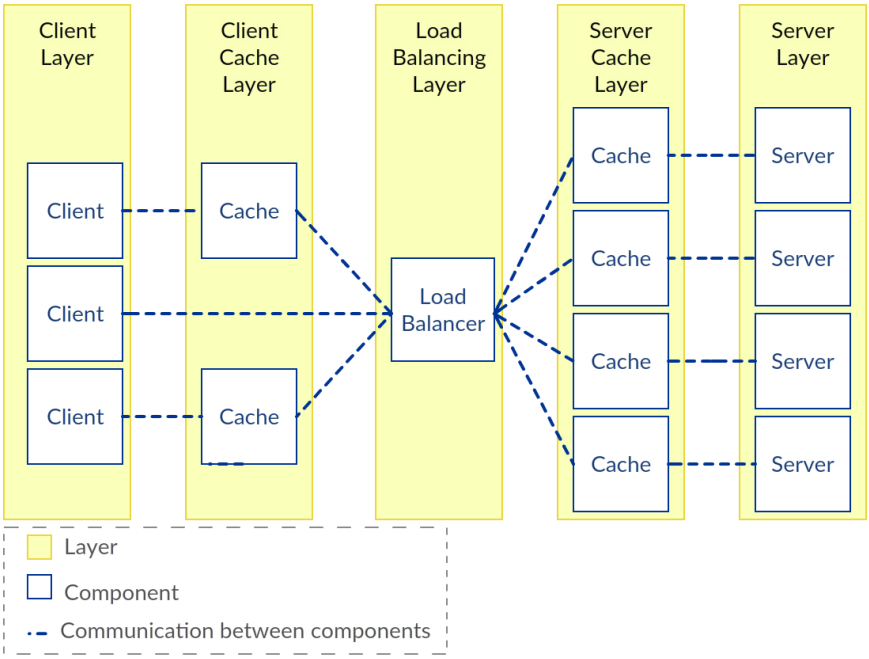


Figure 2.1.: Example for a layered system.

Uniform Resource Locators (URL [url94]) and hyperlinks, to realize a REST compliant system. A central feature of REST is the uniform interface constraint. The WWW uses URLs to address and identify resources. URLs are derived from Uniform Resource Identifiers (URI [uri05]). URLs contain information about the address of a resource as well as a means to access the resource. The generic syntax of a URL is the following:

<scheme>:<scheme-specific-part>

<scheme> determines how an address can be accessed while the <scheme-specific-part> contains the address and scheme specific information that can vary from scheme to scheme. A simple example for an URL is *"http://www.iaas.uni-stuttgart.de"*. The URL indicates that a resource can be accessed via the HTTP protocol at the address *www.iaas.uni-stuttgart.de*. To access and manipulate resources, HTTP defines a set of *standard methods*, including their semantics and properties. Especially GET, PUT, POST, and DELETE are of interest for REST, since they can be used to realize a CRUD-like interface (*Create*, *Retrieve*, *Update*, *Delete*). GET is used to retrieve a resource, PUT replaces a resource with the transmitted message body, DELETE deletes a resource and POST can be used to create new resources. The GET method is defined by HTTP as a safe method, which means that a request using the GET method should not cause side-effects on the server side. The idea is that users can use the GET method to explore resources without causing side-effects. Another important property, shared by GET,

PUT, and DELETE, is idempotency, which means that a request sent multiple times must result in the same response each time.

To understand the content of requests and responses, HTTP uses meta-data in the form of headers, to describe the content of a message. The *Content-Type* header is used to describe the type of the content with a media type. A common media type for example is *text/html*. It stands for a textual representation in the Hypertext Markup Language (HTML). It is the common representation for resources as websites. HTML representations often contain hyperlinks to other websites or resources. Hyperlinks are used to guide users through processes or to provide additional information. Since GET is a safe method the user can follow hyperlinks without the risk of causing negative side-effects. This is the concept of HATEOAS as described in the previous chapter. Other media types are *application/json*, or *application/xml* that are often used for machine-to-machine communication in the WWW. There are also media types for pictures, audio files, video files, and many more. Since resources often have different representations, servers need a means to determine which media type to send to requests.

There are two different kinds of *content negotiation*. Server-driven negotiation, where the server decides which representation is most appropriate, and client-driven negotiation, where the client picks its favoured media type out of an initial response from the server, that includes all available media types. HTTP enables this through *Accept* headers that can be used by clients to specify which media types, charsets, encodings, or languages they accept, so the server can deliver the most appropriate. The accept headers also allow to prioritize certain options e.g. clients can give the highest priority to text/html so they always receive text/html unless the server is not able to respond with text/html in which case the server would try to deliver the second highest prioritized media type. HTTP, URL and MIME-Types are used to cover all aspects of self-descriptive messages and the uniform resource interface. More technical aspects of REST are the layered system constraint and the caching constraint. The layered system constraint enables hierarchical layering of a system in which each only knows those other layers it is interacting with. This principle is used throughout the WWW. Clients direct their resources to specific addresses and get responded to, but the clients are not aware of how the request was processed after sending it to the known address. Often there are caches as intermediaries that reduce the overall traffic.

Caching is another constraint of REST that is complied with by the WWW and is realized through the use of HTTP headers. There are several different cache related headers defined in HTTP, most notably the *cache-control* header. The cache-control header allows to provide requests and responses with*cache directives* that *must* be obeyed by all caches along the communication chain. Examples for directives are *private/public* to let caches know that a message must not be cached (private), or may be cached (public). Other examples of cache related headers are the ETag header, short for entity tag that is often used to compare versions of a resource representation, and the Expires header that allows to set an expiration date to representations. With the compliance to layered system and caching the WWW is compliant to

all of the constraints defined by Fielding and therefore to be considered a REST compliant system.

2.2. Service Orientated Architecture and REST

The term *Service Oriented Architecture* is defined by The Open Group [ope] as follows:

"Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.
A service:
- Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports)
- Is self-contained
- May be composed of other services
- Is a black box to consumers of the service "[2]

In an actual service oriented system, a service is a software component that offers a specific functionality, at a specified address, that can be accessed through the network. The W3C [w3c] defines a Web Service as

"...a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards"[3].

This definition introduces several standards that are used to realize web services. The *Web Service technology* stack consists of a set of standards that can be used to connect heterogeneous web services over a network, and thereby encouraging loose coupling between a *service consumer*(client) and a *service provider* (server), which is a basic design goal of SOA. To consume a service the client needs information about where and how to access a service. Consumers and producers communicate using self-contained documents that contain few assumptions about technical aspects of the receiver of a message. SOAP[4] is a message architecture that specifies rules of processing, binding to a transport protocol, and the format of SOAP messages. Usually HTTP is used to transport SOAP messages. The structure of a

[2]https://www.opengroup.org/soa/source-book/soa/soa.htm
[3]http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice
[4]Originally *Simple Object Access Protocol* but now a name for itself

SOAP message is shown in figure 2.2 b). A SOAP document has a root element called *envelope*, which contains a *header* element and a *body* element. The header element contains an arbitrary number of *Header Blocks* containing information for SOAP processors along the message path and may be modified by intermediaries. The header elements can be used for several purposes such as routing, or configuration of transactions, reliability, or security. The body element is intended for the ultimate receiver of the message and contains the payload for the service provider. The body can be composed of several *Body Sub-elements*. The structure of SOAP messages are defined by XML Schema, which enables easy marshalling and unmarshalling for SOAP processors.

WSDL (Web Service Description Language [wsd]) is an interface definition language based on

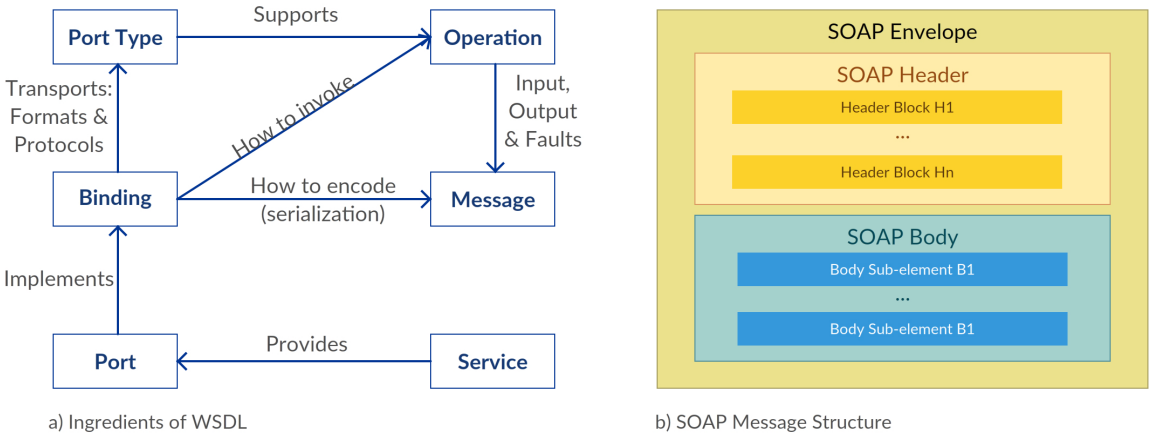


Figure 2.2.: a) Ingredients of WSDL and b) SOAP message structure based on [WCL+05]

XML used to describe web services. A WSDL document is comprised of several elements. The ingredients for WSDL are shown in figure 2.2 a). *Operations* represent the actual functions that can be performed at a specific port type and are grouped within a *portType* element. Messages are used by abstract *operations* as input and output. The *types* element (omitted in the graphic) contains definitions of data elements that can be used within *Message* elements to specify the structure of specific messages. *Binding* elements are used to specify concrete protocol and data format specifications for operations and messages defined by specific portType. *Ports* are used to provide an endpoint, e.g. an address, for specific bindings. And lastly a *service* is a set of related ports . Web services (as defined above) provide a way to realize services as described by The Open Group. The top level elements of a WSDL definition are shown in listing 2.1. The *import* element can be used to import other definitions to modularize WSDL definitions. The *documentation* element is optional and can be used within any other WSDL element. It can be used as a container for human readable content. The *extensibility element* can be used to define technology specific bindings, but must use a different namespace as that of WSDL.

REST is based on resources and their identification, as well as a uniform interface for all

Listing 2.1 The top level elements of a WSDL definition[wsd]

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
   <import namespace="uri" location="uri"/>*
   <wsdl:documentation .... /> ?
   <wsdl:types> ?
   <wsdl:message name="nmtoken"> *
   <wsdl:portType name="nmtoken">*
   <wsdl:binding name="nmtoken" type="qname">*
   <wsdl:service name="nmtoken"> *
   <-- extensibility element --> *
</wsdl:definitions>
```

?=0..1
*=0..n

resources, and stateless communication through self-descriptive messages. A service can also be perceived as a resource, since a resource can be a *logical representation of a repeatable business activity with a specified outcome*. The layered system constraint also provides a *black box condition* for consumers of such a service (clients that access the resource) and a REST API is usually *self-contained* as REST APIs are often realized as servlets and can be deployed to HTTP servers without further knowledge. Service composition can also be achieved through a resource implementation capable of invoking other resources or by the use of BPEL as described in [HFK+14]. REST is fully capable to be used to provide services. As pointed out in the introduction, REST is being used to realize a lot of APIs. The vast majority of APIs on Programmableweb are REST APIs (see figure 1.2).

In [PZL08], Pautasso et.al. discuss the weaknesses and strengths of RESTful web services and WSDL based web services and provide a means to objectify the decision between using WSDL or REST. For the discussion of this paper, services compliant with the REST architectural style may also be called *RESTful*. According to Pautasso et.al SOAP and WSDL have been adopted widely as means to provide interoperability between heterogenous middleware systems. *Protocol transparency and independence* is provided by SOAP, as messages may be transported through various nodes via various protocols. Security aspects are declared specifically as SOAP headers and are therefore independent from the means of transportation along the way. WSDL provides a machine-processable description of syntax and structure of corresponding request and response messages, independent of actual implementation, so the same interface can easily be bound to different implementations without consumers of the interface noticing. Also WSDL is capable of modeling synchronous behaviour as well as asynchronous behaviour. Another strength of WSDL and SOAP is the existence of mature engines and tools that hide a lot of complexity to developers [PZL08]. A downside to the current tools however is that they can be misused such that interoperability problems can occur when the interface description

contains structures, that are native to a implementation, when generating interface descriptions from already existing software components. They suggest to enforce *contract-first* development to mitigate that particular weakness. Other weaknesses of SOAP/WSDL are stated in [PZL08] as misinterpretations partly due to the impedance mismatch between XML and object-oriented programming languages, also XML Schema as a very rich language is not supported completely by all SOAP/WSDL implementations. As a work around it is suggested to use *good enough* constructs known to be interoperable.

Services compliant to the REST constraints are perceived to be simpler than the SOAP/WSDL approach, because REST makes use of existing standards such as HTTP, XML, URL and MIME. HTTP clients and servers are present in all major programming languages and platforms. Developing clients to RESTful services requires less effort, because services can be tested using a web browser and there is no need to develop customized clients. The HATEOAS principle incorporated into the definition of REST allows service discovery without the need for a central registry. It is also stated that RESTful Web services are doing very well to scale, to serve large numbers of clients, due to the support for caching, clustering and load balancing built into REST. REST is also more flexible in terms of data format. It is common for RESTful web services to use JSON or even more simpler formats such as plain text. As a weakness it is stated that there is quite some confusion with regards to accepted best practices in the development of RESTful web services. So called *Hi-REST* recommends the use of the major HTTP verbs (GET, PUT, POST, DELETE) and the use of XML, while *Lo-REST* only uses 2 verbs (GET and POST) and thereby tunnels other requests through the use of special HTTP headers such as *X-HTTP-Method-Override*, or through hidden form fields. Since [PZL08] was published in 2008 the discussion might not be up to date. As it seems the majority of REST developers use *Hi-REST* and the discussion of what is really RESTful has moved to the use of hypermedia and HATEOAS.
The *Richardson Maturity Model* rewards the highest level of maturity, e.g. REST compliance, to APIs using multiple HTTP verbs and most importantly, make use of HATEOAS [rmm]. But there still might be cases where tunneling requests through one HTTP Verb is inevitable. This can result in malformed URIs since *GET* is not allowed to have a body. Input data can then only be encoded to the URI, which then can get too long (Status Code 414 Request-URI Too Long [htt99]).

As a method of comparison between the two styles (REST and SOAP/WSDL aka. the Web Service Stack WS-*) Pautasso et.al. use the notion of *architectural decisions* and *architecture alternatives*. Architectural decisions represent the basic design issues and ideas behind a technical solution and are seen as conscious decisions determining non-functional properties of components or a system as a whole. Each *architectural decision* is associated with a certain amount of alternative options.
The article describes 4 different levels of comparison:

Comparison of architectural principles: This level focusses on principles that actually define the two integration styles. The first principle addresses how the web is used by the two styles. REST uses HTTP as part of the application since it is encouraged to use different verbs with different semantics while SOAP uses HTTP solely as a transport protocol. SOAP is always transmitted using the HTTP *POST* method which has no clear semantic meaning. What operation is to be invoked is determined at the ultimate receiver through the payload (a SOAP envelope) of the HTTP request. Every SOAP message is *tunnelled* through HTTP POST.
The second principle to be compared is the capability of dealing with heterogeneity. It is argued that the web is a rather homogeneous client-server environment since all participants use the same protocol, which is HTTP. Heterogeneity is derived from different browsers that competed with each other and resulted in different renderings of HTML or being incompatible to JavaScript libraries, but all browser support the same HTTP standard and a large set of standard document types. The SOAP style comes from more heterogeneous environments since it originated in enterprise computing. Many of the involved systems have components that are implemented with different kinds of technologies and are legacy software. So the WS-* standardizations provide the *plumbing* for integrating different enterprise components.
The third aspect of the comparison of principles is *Loose Coupling*. Important aspects of loose coupling are *time and availability*, *location transparency*, and *service evolution*. The time and availability aspect addresses the issue if service consumers are still able to interact with service providers even if they are permanently not available. This is possible when using WS-* through the use of persistent reliable queues, while a RESTful web service relies on synchronous communication that can not prevent, that requests will fail when the receiver is not available. Dynamic late binding of services is supported by most WS-* tools, while it is only possible to a degree for RESTful applications through DNS address translation, but requires additional effort. Another important aspect of loose coupling is the evolution of web services. This is supported especially by REST, through the uniform interface constraint and the the basic extensibility of XML, which is also shared by WS-*. In summary both styles support all three examined principles.

Comparison of conceptual decisions: Conceptual decisions are concerned with the following decisions and their architectural alternatives: *integration style* (shared database, file transfer, remote procedure call, or messaging), *contract design* (first, last, or without contract, meaning if the well formed interface is defined before implementing the service, or the other way round), *resource identification* (finding the abstractions), *URI-Design* (with or without scheme), *Resource Interaction Semantics* (Lo-REST vs. Hi-REST), *Data Representation/Model* (whether XML Schema can, or must be used, or if a custom format is applicable), *Message Exchange Patterns* (capability to *Request-Response* or *One-Way* communication), and *Service Operations Enumeration* (Ability to define which set of operations is exposed by a service interface). REST requires eight architectural decisions out of the above listed nine with only ten alternatives, leaving

five decisions with only one option. The WS-* on the other hand only requires five decisions, but yields more than ten alternatives for these decisions. It is argued that REST offers a freedom of choice leading to substantial design and development efforts, while the choices for WS-* are within strict conceptual boundaries, as well as easier to implement, due to high degree of standardization.

Comparison of technology decisions: The comparison of technology is concerned with possible *Transport Protocols*, *Payload Formats*, possibilities of *Service Identification*, *Service Description*, *Reliability*, *Security*, *Service Composition*, *Service Discovery*, and *Implementation Technology*. Substantial differences are in possibilities of transport protocols, which is basically only HTTP for REST, and more than seven protocols for WS-*. The opposite situation comes to play concerning the payload format where WS-* is restricted to SOAP, while REST is rich in possibilities. All ten topics can be addressed by WS-* and REST, but WS-* has significantly more architectural alternatives (REST 17, WS-* 25).

Vendor asset-level comparison: In this section concrete tools are evaluated such as web browsers and web servers implementation of HTTP and existing SOAP engines and WSDL tools. This part however is not included in the paper.

It is concluded that REST as well as WS-* have similar quantitative characteristics. On the architectural level the WS-* requires less decisions, but has more available alternatives. On the technology level REST and WS-* require an equal number of decisions, with less alternatives for REST to be considered. The WS-* stack requires a number of decisions related to different layers of the stack that add additional complexity compared to REST. However advanced functionality is delivered through WS-* standards that would require substantial effort to be realized for RESTful services. It is also argued, that architectural decisions required for RESTful services can lead to significant development efforts and technical risk, such as the design of exact specification of resources and their URI addressing scheme. When using the same technologies for both integration styles (XML/HTTP and SOAP/HTTP) the two seem rather similar. Without a need for enterprise features of WS-*, it is concluded, key decision drivers are degree of flexibility and control, in which REST scores better. Other important aspects are development efforts and technical risk (implementation design, development, and maintenance), degree of open source and vendor tool support, and programming interface convenience. On the one hand, WS-* has a better tool support and programming interface convenience, on the other hand, this also introduces dependencies to vendors and open source projects. The main conclusion drawn is to use “RESTful services for tactical, ad hoc integration over the Web (à la Mashup) and to prefer WS-* Web services in professional enterprise application integration scenar- ios with a longer lifespan and advanced QoS requirements”.

2.2.1. Relevancy to code generation

Especially the before mentioned *programming interface convenience* of the WS-* comes to play when using a contract first approach to implementing web services. The contract first approach means that the interface is defined *before* a line of code is written. The development strategy then focusses on the specification of interfaces. In the context of this thesis a WSDL defined interface is a model that can be used to generate the required server and client stubs. This generation of code is common and widely used in development of web services and available for many platforms. The difference to REST is significant with respect to the concepts and terms used in modeling. While WSDL is primarily about operations and their input and output messages, REST is about resources, representations and links to other resources. For most developers the notion of operations and input/output is natural, since there are similar concepts in many programming languages such as methods and their parameters. This can be shown with a simple ToDo list service as example. The service shall be capable to retrieve a shortened list of upcoming ToDos, retrieve details for a specific ToDo and to add new ToDos to the list. In the case of a web service this would result in the three operations shown in listing 2.2. The listing shows a method to retrieve all ToDos (line 2), one method to add a ToDo to the list (line 5) and one method to retrieve the particular ToDo with the given *todoId* (line 8)

Listing 2.2 Java method declarations for the ToDo list service

```
1 //retrieves all ToDos
2 public ArrayList<ToDo> getToDos();
3
4 //adds a ToDo to the list
5 public void addToDo(ToDo toDo);
6
7 //retrieves the ToDo with todoId
8 public void getToDo(String todoId);
```

Of course one needs to define the data types and bindings, but these abstractions suffice to realize the web service. The developer only has to implement the three functions after the WSDL is defined.

To realize this service with REST, one first has to identify the resources in the system, and how they interact. Most likely a developer adept with the REST principles would use two classes of resources: A *ListOfToDos* resource class and a *ToDo* resource class. The second task is to provide a URI scheme for the resources. ListOfTodo is a resource that exists only once in this service so its URI would be static to the base URI of the server, such as *http://localhost:8080/listoftodos*. There can be many ToDos in the system so each ToDo needs its own URI, best achieved through the use of a dynamic URI part with an identifier, such as *http://localhost:8080/listoftodos/{todoId}*. The ListOfToDos requires a representation that

contains information on all ToDo resources, as well as a hyperlink to each resource. The developer now has to implement all HTTP verbs for each resource separately. Also the developer must decide whether the ListOfToDos supports the POST method with a ToDo representation in the request body to create new ToDo resources, or if the ToDo resource implementation allows the clients to create new ToDos by sending a HTTP PUT request to a *empty* ToDo resource (the todoId is not assigned yet) with a new ToDo representation in the request body.
On the client side those two implementations of essentially the same functionality behave differently. When consuming the WSDL implementation the client would maybe call the getToDos operation to view all upcoming ToDos. Then the client would have to extract the identifier from a ToDo and send another request with this identifier to view the ToDo. The REST scenario for this use-case begins similar, as the client sends a GET request to the ListOfToDos to receive a representation of all ToDos. To examine a certain ToDo the client simply has to follow the offered hyperlink, using another GET request to retrieve the desired information. This small example gives an impression on the differences and similarities of the WSDL and REST integration styles. The differences in REST require a different modeling approach, which results in more complex code generation from a model since the terms and concepts of REST are different from programming languages, unlike the terms and concepts of WSDL.

2.3. Model Driven Software Development and Architecture

The book *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management* [SVEH07] introduces *Model Driven Software Development* with a definition as follows:

"Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen."[SVEH07]

Translated to english: "Model Driven Software Development (MDSD) is a genus for techniques to automatically create runnable software from formal models."

This definition introduces three aspects:

Formal Models: A formal model in the MDSD sense is a model, that completely covers a certain aspect of a software. A formal model does not describe everything. But it has to be clear what is described by a model and what isn't.

Creation of Runnable Software: Models are often used for documentation purposes, to give an overview to developers, or even as close-grained specifications for manual implementation. This however is not MDSD. The ultimate goal of MDSD is to create

runnable software. So the main purpose of models used in MDSD is to be used as a basis for code generation. There are two possible concepts to create runnable software from models. An interpreter is a piece of code that uses models as input at runtime and performs actions based on the given models. For this thesis the relevant concept is that of a generator. A generator is a piece of software that also uses models as input, but unlike a interpreter generates source code in a particular programming language.

Automation: The transition from a formal model to runnable software should happen automatically by a software, in this case a generator. The idea is, that models become almost equivalent to source code, so that changes in the model ultimately result in changed source code. In the previous chapter it was described that WSDL is often used to generate server and client stubs. These stubs are generated once and then extended through manual implementation. This is explicitly not the goal of MDSD. MDSD aims to generate code, that also changes with changed models. So generated code is not intended to be modified through other means than automatic code generation. The overall system of course, contains generated and manually implemented code.

Model Driven Architecture *Model Driven Architecture (MDA)* is a software development approach proposed and standardized by the Object Management Group (OMG) [omg]. According to [SVEH07] the MDA is a standardization initiative from OMG towards the topic of MDSD. MDSD and MDA use the same basic terminology for their approach. The principle of MDA and MDSD is the use of models and their transformation to other models. Key models are *Platform Independent Models (PIM)* and *Platform Specific Models (PSM)*.
A *PIM* is a model of a software that is completely independent from specific platforms and doesn't provide any implementational details. Although a *PIM* might represent a complete system it is not sufficient to generate runnable application code. A good example for a PIM would be a entity relationship diagram, that shows entities, their relationships to each other and maybe even their capabilities. Still without further information it would not be possible to generate running application code. For example if resources of a REST service were modeled with an entity relationship diagram it would still not be possible to use that as a basis for code generation since a lot of vital information is missing, for example how resources can be implemented, since there is no information in the model about the platform.
A *PSM* however contains all information necessary to generate runnable software as it also has information about the target platform and implementational details. An example for a PSM would be a detailed class diagram, containing all necessary information to generate runnable source code the model would have to contain all helper classes and their relationships as well.
A *PSM* can be created through model-to-model transformation by combining a platform model (provides platform specific information) and a *PIM* and thereby transforming the *PIM* to a *PSM*.
A model-to-text transformation is a special kind of model transformation, where *PSM* is transformed to source code. An illustration of this process is shown in figure 2.3. MDSD aims to improve the quality of software, the reusability, and to increase the efficiency of software development. MDSD tries to automate recurring software development tasks that are error prone. There are several important reasons to use MDSD:

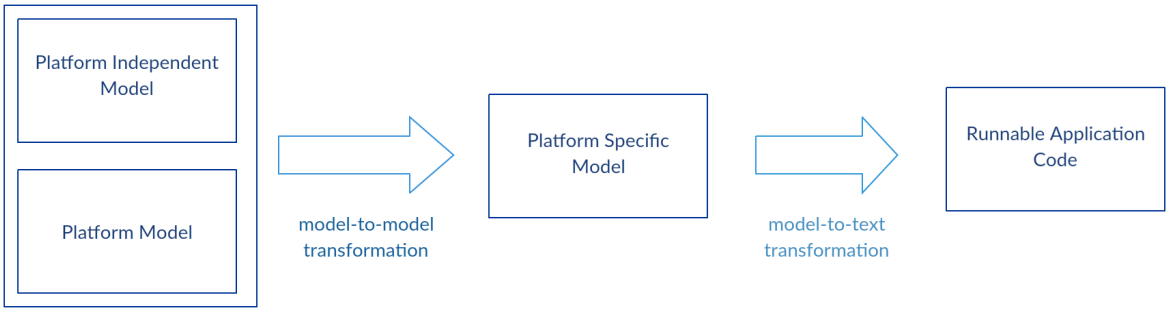


Figure 2.3.: Model Transformation Process

Abstraction: An important reason to use MDSD is that MDSD enables software development on a higher level of abstraction. Instead of developing in terms and concepts of a certain programming language, the developers can model the system based on concepts of the system (e.g. a developer models resources instead of writing Java classes). A lot of the (technical) complexity is then hidden from the developer because it is weaved into the transformation from model to code. The technical complexity has only to be dealt with once, when developing the transformation instead of every time, a developer has to extend the system.

Uniform Architecture: The transformation from models to code is realized by a set of rules and mappings from elements of the model to the target language. So by definition of the transformation all components of the models (which are of the same type) are constructed the same way. This brings several advantages: The system resembles a homogeneous collection of components all constructed by the same principles so the system doesn't contain components dealing with the same conceptual issue, but using different solutions. Expert knowledge of developers can better be used, since they can be assigned to different tasks. Some responsible for the development and improvement of the model-to-text transformation and some to deal with specific issues, that can not be automated.

Speed of development: MDSD does not necessarily result in the faster development of new systems. Generators and transformations have to adapted and extended. Models have to be developed and maintained. But the development speed increases when a system has grown and has to be maintained. For if a large number of system components uses constructs that become deprecated, in manually developed systems this would require to search for these components, understand what they are doing, and then fix them. In generated classes this becomes easier. The transformation from the model to code has to be adapted once, and all affected classes can be generated with the next transformation cycle.

Reusability: Architectures, modeling languages and generators can be resused for the development of other systems. Especially when the developed systems show a number of similarities.

Interoperability and Platform Independence: The use of platform independent models enables the reuse for other platforms. This is especially true when platform specific models can be used to generate code for another platform, but the same programming language. The problem however is that the models are seldom generic enough for several platforms, because the target platform is often reflected in the models. But it is certainly easier to convert a system to a completely different platform when there are models present that describe the systems.

Quality of Software: The main reason for improved software quality is that uniform systems can be tested easier. Also the specific transformation rules can be developed by experts (developers and their skills are heterogeneous). The overall quality of software can improve but is of course dependent on the quality of the generators and transformations used.

Unfortunately there are also downsides to MDSD:

High Initial Effort: When starting with a *blank slate*, the initial effort to develop a system is very high. Besides architectural decisions concerning the system, there are also architectural decisions to be made that address issues of MDSD. Then there is a lot of "meta development" to be done. Modeling languages and transformations have to be defined and developed, before the actual system can be developed.

Possible Loss of Code: The generator is intended to overwrite generated classes with new versions. So the generator needs to be adapted to recognize manually implemented code. Otherwise it is possible that the code generator might overwrite manual code. A good practice would be that developers comply with the rule that no generated files may be touched. But it is understandable that a quick fix might have to be applied to a generated class in case an angry customer calls.

Rigidity: Using code generators takes a lot of choices from programmers but also introduces rigidity. Programmers in a MDSD environment have less freedoms regarding their style of implementation. An issue is that problems in development are only solved, if they have been thought of before they actually occur, so they can be taken into account for the transformation rules. If not, situations can arise in which developers have to violate the rule of not modifying generated files.

Increased technical complexity: Using MDSD often results not only in the development of a system but also in the development of MDSD tools. This can multiply the number of technologies that have to be dealt with since MDSD requires to define, process, and transform models. Since MDSD tools also have to be maintained this introduces further possible sources for errors and bugs.

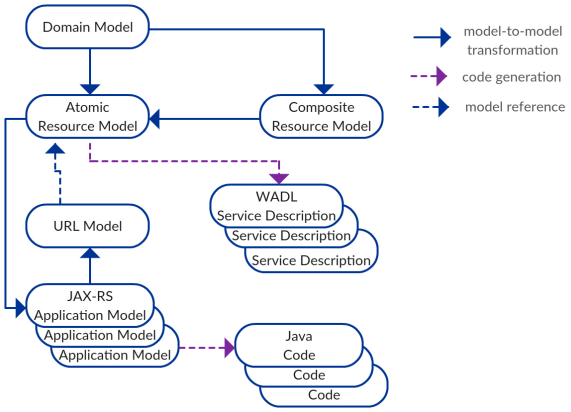


Figure 2.4.: Layered meta-model for REST applications in accordance to [HLP15]

The use of MDSD has to be considered for each project individually. A project to develop a prototype or projects to test the feasibility of an architecture are better and faster done without MDSD. However the development of a system that will be used for a long time and has to be maintained may benefit greatly from MDSD.

2.4. Modeling Tool

In their paper *A Model-Driven Approach for REST Compliant Services* [HKLS14] Haupt et. al. propose a multi layered model for REST APIs that partially enforces the generation of REST compliant application code. A detailed view on the resource meta model is given in [HLP15]. Haupt et. al. provide a prototypical implementation of the approach which is also the basis for this thesis. The complete stack of layered meta models can be seen in figure 2.4.
The *Domain Model* enables the modeling of an application independent from REST. The meta model of the domain model can adapted to serve the needs of any specific application domain. This enables domain experts to use their knowledge about the domain, to express their knowledge in concepts they are already familiar with (e.g. entity relationship diagrams). The next step is the transformation of the domain model to a resource model, either an *Atomic Resource Model* or a *Composite Resource Model*. This step requires a non trivial effort because it involves the definition of a mapping of concepts natural to the domain model towards the resource model. This *impedance mismatch* (originally the mismatch between relational data bases and object orientation) means that two concepts are not mutually compatible. In this case the concept of an impedance mismatch becomes visible when mapping non resource oriented meta models to resource oriented meta models. Figure 2.5 shows a *Book* entity that has *title* and *loanPeriod* as attributes, and the method *setLoanPeriod(...)* to update the loanPeriod. While in an object oriented environment, there is an obvious way to implement

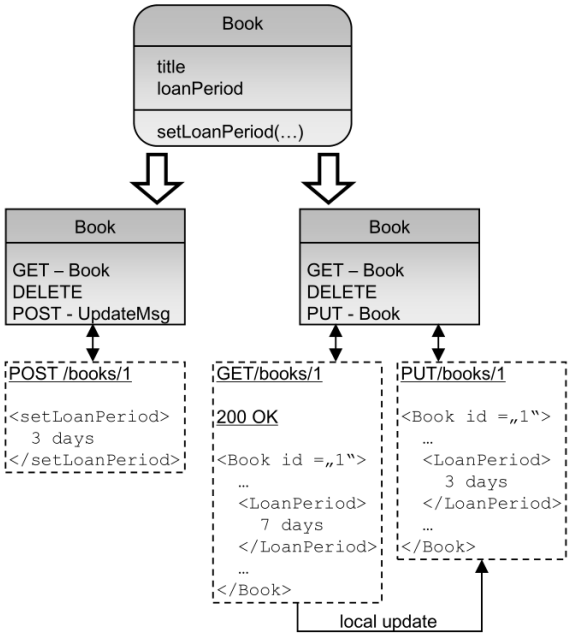


Figure 2.5.: Possible impedance mismatch [HKLS14]

the Book entity as a class, in REST this issue needs further examination. There are two possible strategies to update the loanPeriod attribute of a *Book resource*. One strategy is the use of the POST method and only transmitting the part to be updated. The other strategy is via the use of GET and PUT methods. A client retrieves a representation of the book resource, modifies it, and uses a PUT request to update the current state of the resource. Using GET and PUT, both idempotent methods, allows the requests to be resubmitted in case of network failures. Disadvantages are, that this approach requires two messages, which can result in concurrency issues such as lost updates: Two clients retrieve the resource simultaneously, perform a local update, and both sent a PUT request to update the resource. The PUT request reaching the resource first will be futile, as it is overwritten by the second request. The POST method strategy requires only one message (with even less payload). The POST method, however, is not defined as idempotent and therefore can't be resubmitted. This issue is addressed by the notion of *marking* the model to "prefer safe operations" through a separate model.
The *Atomic Resource Model* is the core model to specify a REST application. The meta model for the atomic resource model provides constructs to model the application in terms of resources, their relationships to each other, and their individual interfaces. The *Composite Resource Model* is an extension of the atomic resource model allowing to group several atomic resources together to form a composite resource. The atomic resource model and the composite resource model are further refined in [HLP15] towards a more conversation centric approach. A conversation is a number of communication activities between two or more participants. As

defined by Haupt et.al there are two types of participants in *RESTful Conversations*: *Clients* interacting with an API, and *Resources* interacting with clients. The *communication primitives* are given by the uniform interface constraint. In case an API uses HTTP, which is common, each communication is initiated by the client through a request, which is answered with a response message (synchronous communication). Each message includes the HTTP verb, the resource identifier and is stateless. The *state of the conversation* is kept entirely on the client. There are four examples given for RESTful conversations: *Redirecting*, *Accessing Collections of Resources*, *Try-Confirm-Cancel*, and *Long running Requests*. These four types of conversations are explained in detail in [HLP15]. All of these conversation types have in common that they have more than two participants respectively their participants consist of one client and several resources.
The atomic resource model is described through the *interaction centric meta model* since interactions only require single resources. The *conversation centric meta model* (based on the composite resource model) extends the interaction centric meta model with new elements (for the before mentioned conversation types) that are more specific than the generic elements in the interaction centric meta model. The composite resource model is used to group resources (and their relationships), participating in a particular conversation type.
So far, none of the introduced models contains a means to associate resources with their identifier. This is addressed by the *URL Model* which can be used to define URL schemes for each resource in the atomic resource model. The atomic resource model and the URL model are then sufficient, to generate an application model. Application models are dependent on the target platform. This thesis will provide a platform specific meta model for *JAX-RS*[5] for the existing prototype as well as the relevant transformation from the atomic resource model and URL model to the JAX-RS model and the generation of application code from the JAX-RS model. The versions of the atomic resource model and URL model used as basis for this thesis are introduced in 2.5 and 2.6.

2.5. Resource Metamodel

This chapter introduces the metamodel of the atomic resource model that will be used for this thesis. It is the primary source model for the model transformation and can be edited directly with a graphical editor. Figure 2.6 shows an UML class diagram of the complete *Resource* metamodel.

The basic modeling elements are *Resource*, *Method*, and *Link*:

***Resource*:** The *Resource* element is the main modeling element since it is the direct representation of a resource within a REST API. It can have a list of *EntityAttributes* to model the information structure of the underlying REST Resource. So the *ResourceModel*

[5] see 5.3.1 for information about JAX-RS

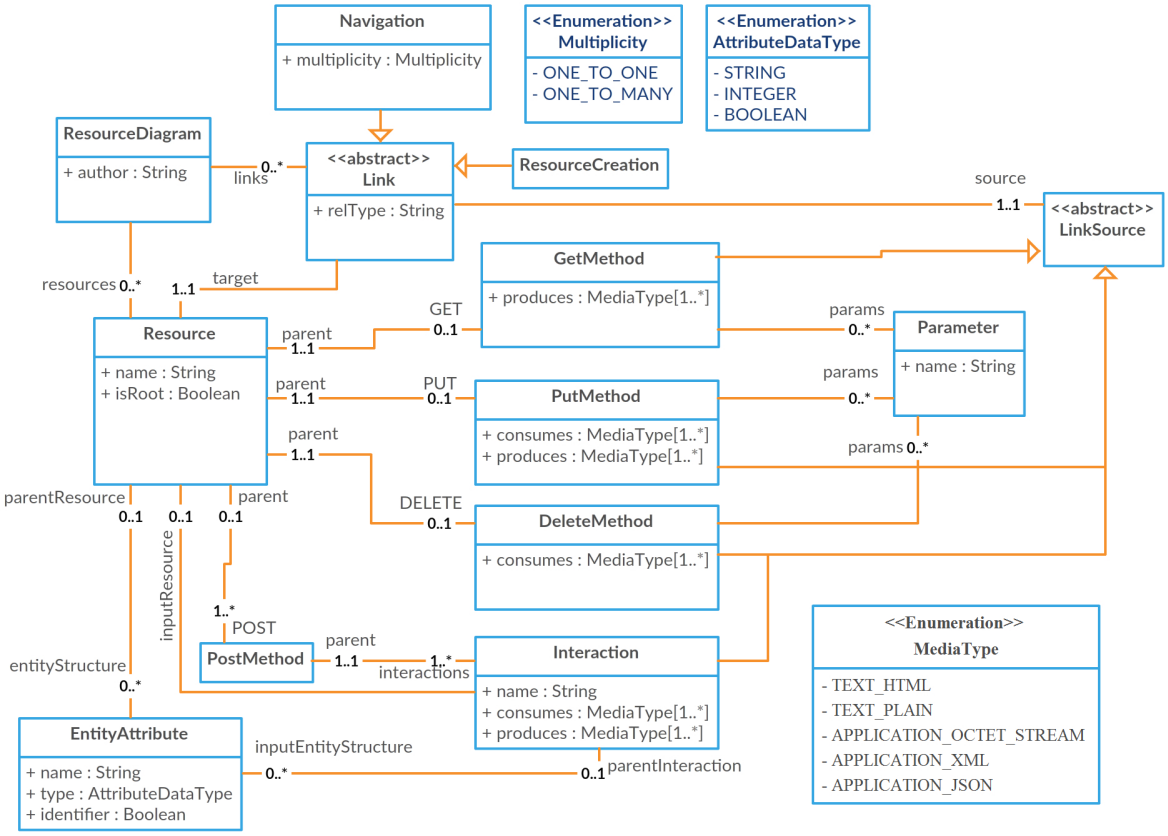


Figure 2.6.: Resource meta model

not only contains infrastructual information, but can also be used to define a domain specific information model through the use of *EntityAttributes*, which define data fields with a name and a type. This concept could be improved further by enabling the use of schema information.
The *Resource* element also holds separate attributes for each of the four HTTP Methods: GET, PUT, POST, and DELETE.

Method: The *Method* element is a supertype for the four HTTP methods (it is not depicted in figure 2.6 because it has neither relations to other elements besides the four methods, nor any attributes). On the one hand *GetMethod*, *PutMethod*, and *DeleteMethod* are very similar, only differing in whether or not they have a list of media types for producing and consuming. The *PostMethod* on the other hand has a list of *Interaction* elements, each one having its own method name and separate lists of media types for producing and consuming, since it is possible to have several different POST methods for a resource, each with different semantics.

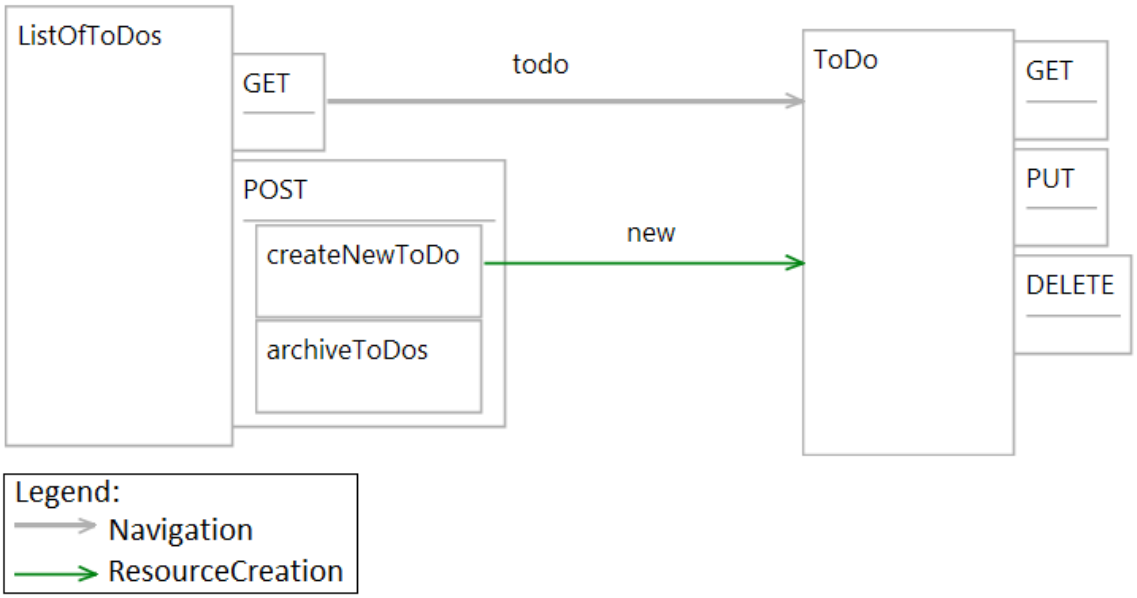


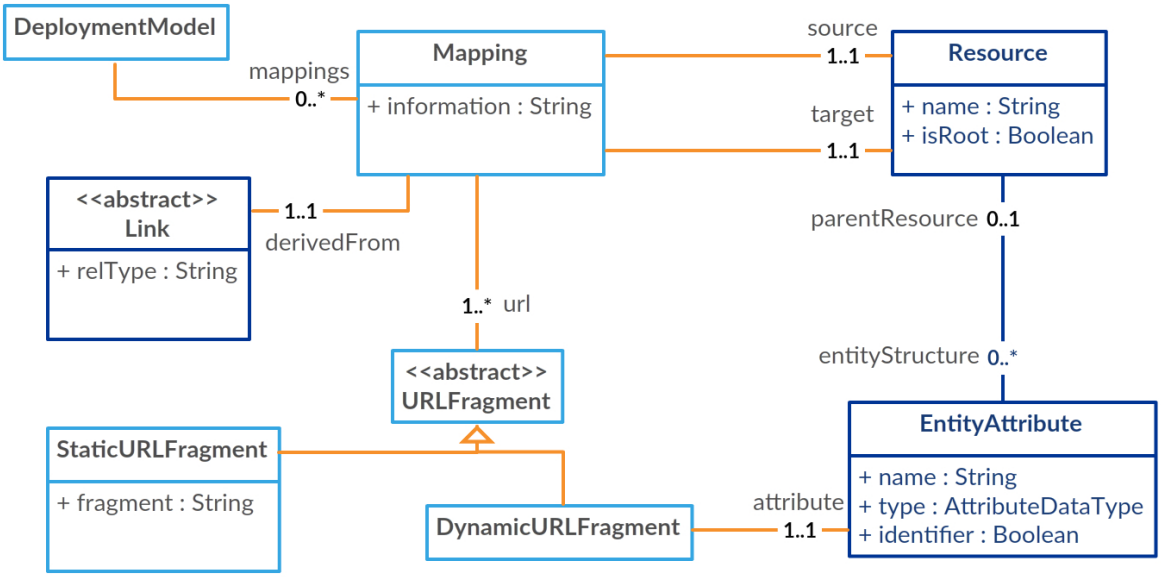
Figure 2.7.: ResourceModel example

Link: The *Link* element defines the relationship between two resources and comes in two different types, *ResourceCreation* and *Navigation*. They can originate from *GetMethod*, *PutMethod*, *DeleteMethod*, and from each of the *PostMethods Interactions*. Links are directed and always point towards a *Resource*. A *ResourceCreation* indicates that the invocation of its source will create a new instance of the target *Resource*. The *Navigation* element is the main modeling tool to realize HATEOAS. A *Navigation* from *Method* A to *Resource* B means that a link to B will be embedded in the Response to a call of A.

Figure 2.7 is a visual representation of an exemplary *ResourceModel*. It depicts the ToDo Service introduced in 2.2.1 extended in some points. The ListOfToDos resource uses the HTTP GET and POST methods. GET is used to retrieve a representation of the resource. The POST method has two interactions defined. *createNewToDo* can be used to create a new ToDo resource, indicated through the green arrow, labeled *new*. The other interaction *archiveToDos* starts a process that transfers finished ToDos to a permanent storage location. The arrow from the GET method of ListOfToDos to the ToDo resource indicates that the representation retrieved with this GET method contains links to particular ToDo resources. The ToDo resource features the GET, PUT, and DELETE methods of HTTP to retrieve, update, and delete the ToDo resource.

2.6. Deployment Metamodel

This chapter introduces the metamodel of the URL model that will be used for this thesis. The Uniform Interface constraint of REST is partly defined through the identification of resources. The *ResourceModel* allows to only specify links and relationships between resources. This linking of resources allows to model how clients can navigate through the API and thereby realizes the HATEOAS constraint of REST. HATEOAS is an important feature to reach the goal of loose coupling between client and server in a RESTful environment. The *ResourceModel* is sufficient to generate client documentation, but to generate an executable Java application also the identification of resources has to be defined. The identification of resources is a vital aspect of the uniform interface constraint. Unique identification of resources is a prerequisite for HATEOAS since it is impossible to guide a client to certain resources if those resources can't be identified. Figure 2.8 shows an UML class diagram of the complete *Resource* metamodel.



Dark blue items are imported from the Resource meta model.

Figure 2.8.: Deployment meta model.

The deployment model enables the association of resources with URIs. An important feature is that the deployment model allows to define the URI of a resource not only relative to the base URI (of the service) but also relative to other resources. The deployment model contains a list of *Mapping* elements:

Mapping: A *Mapping* contains a source and a target, both of which are *Resource* elements from the resource model. Also the *Mapping* has a ordered list of *URLFragments*.

URLFragment: A *URLFragment* can either be a *StaticURLFragment* or a *DynamicURLFragment*. The *StaticURLFragment* contains a String indicating a static path element, while the *DynamicURLFragment* contains a reference to an *EntityAttribute* of a resource model. *StaticURLFragments* in figure 2.9 are for example *domain*, *admin*, and *control*. *DynamicURLFragments* are {*offerId*}, {*invoiceId*}, and {*processId*}.

This design enables the creation of dynamic URL patterns with subresources. Figure 2.9 shows an exemplary deployment model. This deployment model resembles a tree. The tree structure is in general not mandatory, but for this thesis it is assumed that a deployment model always resembles a tree. This guarantees two properties: There are no cycles in the deployment model and there is exactly one path in the resource model for each resource. So each resource can be identified with one distinct URI scheme. The reasons for this are discussed in chapter 8.4: Limitations.
The *RootResource* is the starting point from which all other resources are subresources. Assuming the root is at "localhost:8080" the *AdminResource* would have its addres at "localhost:8080/admin". Since the *URLFragments* are ordered (from to right in this example) the *OfferResource* could be accessed at "localhost:8080/domain/offers/{offerId}". Curly brackets indicate, that the *URLFragment* is a *DynamicURLFragment* referencing the OfferResources identifying attribute *offerId*.

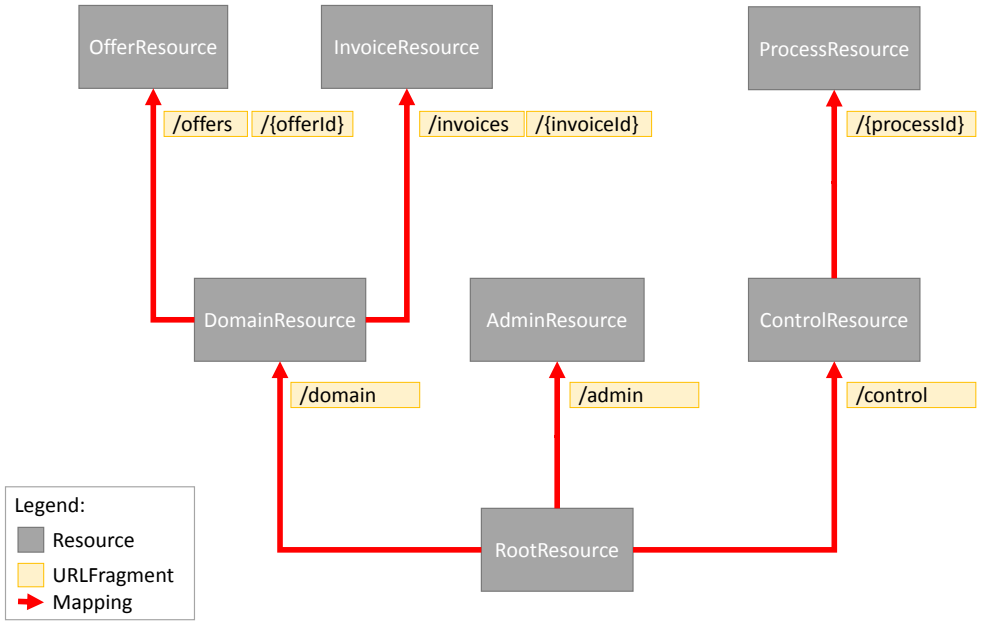


Figure 2.9.: A deployment model.

3. Solution Approach

This chapter describes the task of the thesis and how the solution was approached. It gives an overview over the different stages and their associated tasks, goals, difficulties, and motivations.

3.1. Task Description

The main task of this work is to design and realize the generation of application code from a given metamodel for REST APIs. This includes the development of a platform specific metamodel as well as the corresponding model-to-model and model-to-text transformations. Figure 3.1 gives a short impression about the task. The generated application code shall provide the complete HTTP and resource infrastructure. This means the generated code shall realize all needed functionality to expose resources via HTTP, so the generators user only has to add domain specific code. It is required that the domain specific code can be added to the application in a loosely coupled manner without modifying any of the generated classes. The developed solution has to be integrated in the existing modeling tool described in 2.4.

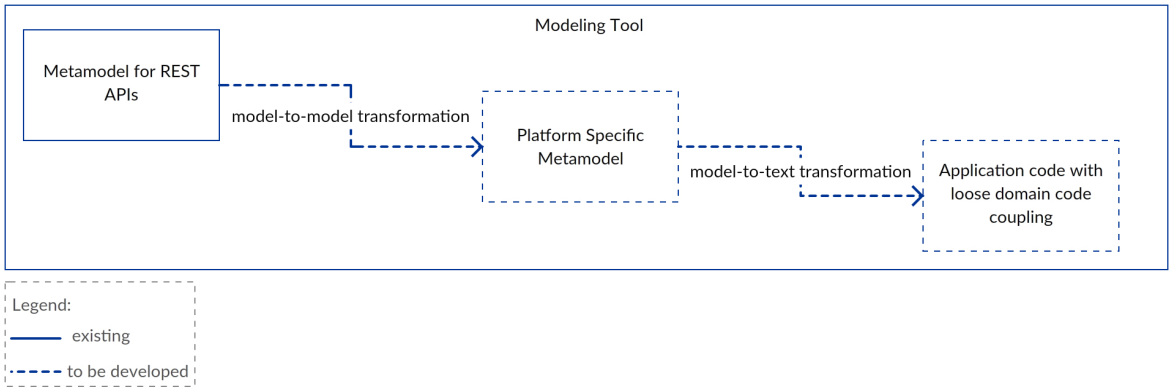


Figure 3.1.: Task of this thesis.

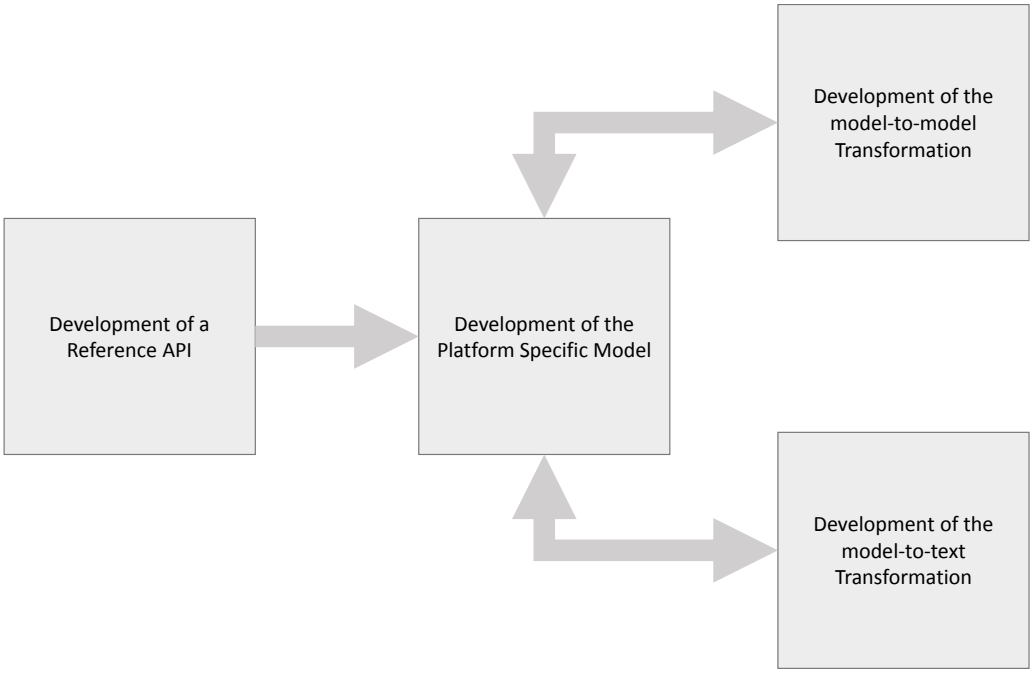


Figure 3.2.: The structure of the Approach

3.2. Description

The approach is structured into three stages: The development of a reference API, the development of a platform specific model and the development of transformations. As depicted in figure 3.2, the third stage consists of the development of two distinct transformations. The model-to-model transformation from *ResourceModel* and *DeploymentModel* to the *PSM*, and the model-to-text transformation from the *PSM* to application code. The following chapters give an overview over the described steps.

Development of a reference API

The first stage is the *development of a reference API*. The goal is to develop a typical REST compliant API to later extract code templates for the model-to-text generation and derive requirements for the platform specific model.
The first step was to find or develop a suitable REST API. This REST API is required to be a typical REST API, compliant to the REST constraints with focus on the facilitation of HATEOAS. The reference API shall also contain all concepts relevant to REST APIs, since the goal is to generate code for all possible constellations. HATEOAS is part of the Uniform Interface constraint and is often either realized in a wrong fashion or not at all. The book “Rest in

Practice Hypermedia and Systems Architecture"[WPR10] uses an application called *Restbucks* as the running example to show how a hypermedia system can be designed and implemented. *Restbucks* was chosen as a reference application because it is an example that is not trivial, as well as not too complex to be implemented as part of this thesis. *Restbucks* will be introduced more detailed in chapter 5.
The second step in the development of the reference API was to manually implement the application. A requirement for the implementation was the use of Dropwizard, a bundle of Java libraries helping in the implementation of a REST API, see chapter 8.1. The challenge of this stage was to implement code that can serve as a template for the generation of application code. As a consequence, this code was allowed to have similar constructs in different classes instead of using superclasses with complex methods. The code had to be structured and consistent throughout all different types of implemented classes. Another important task was the design of a loosely coupled way to integrate domain logic, so that none of the generated classes have to be modified in order to integrate the domain specific code. This reduces the complexity of the code generator since there is no need to scan already generated classes to recognize manually implemented code. This constraint enables that all generated classes can be overwritten safely, but also introduces some rigidity since the developer is no longer free to modify every piece of code in the system.

Development of the Platform Specific Model

After the development of the reference application the second stage was the development of a platform specific model as basis for later code generation. The platform specific model was derived from the reference application to later provide the code generator with all information needed to generate Java code. The platform specific model will be explained in chapter 6. Since the solution has to be integrated into an existing application, first developed by Benjamin Schroth [Sch13] and adapted by Jens Petersohn [Pet14] it was a requirement for the new model to be provided as Ecore model, see 8.1.

Development of the model transformations

The last step to accomplish the task was to develop the model transformations from the provided resource metamodel to the platform specific model, and from the platform specific model to runnable Java code. The existing modeling tool uses the *Epsilon Transformation Language* (see 8.1) to realize model-to-model transformations. Since this technology is well integrated into the existing tool and there is sufficient integration with the development environment (eclipse), such as highlighted syntax and static analysis, the modeling tool did not have to be adapted for the model-to-model transformation. For model-to-text transformation some new technology was introduced to the prototype, see 8.1. During this stage, the platform

specific model also had to be adapted until the complete model generation was realized, due to new requirements discovered during the development of the model-to-text transformation.

4. Related Work

This chapter introduces works related to this thesis. [VP09] proposes a general model to describe REST APIs, while [Sch11] provides a model to not only model the structure, but also the behaviour of a REST API. [LSS09] describes a model driven process to extract REST service interfaces from functional specifications.

4.1. “Modeling RESTful applications”

In *Modeling RESTful applications* [Sch11] Silvia Schreier proposes a meta model to describe REST applications. Schreier defines a structural meta model suited to describe resources and their relations, as well as a behavioural model to describe how different *ResourceTypes* behave. Schreier introduces several different *ResourceTypes* such as *primary resource*, *list resource*,and *paging resource*. Figure 4.1 shows the proposed structural model that also contains *Link*

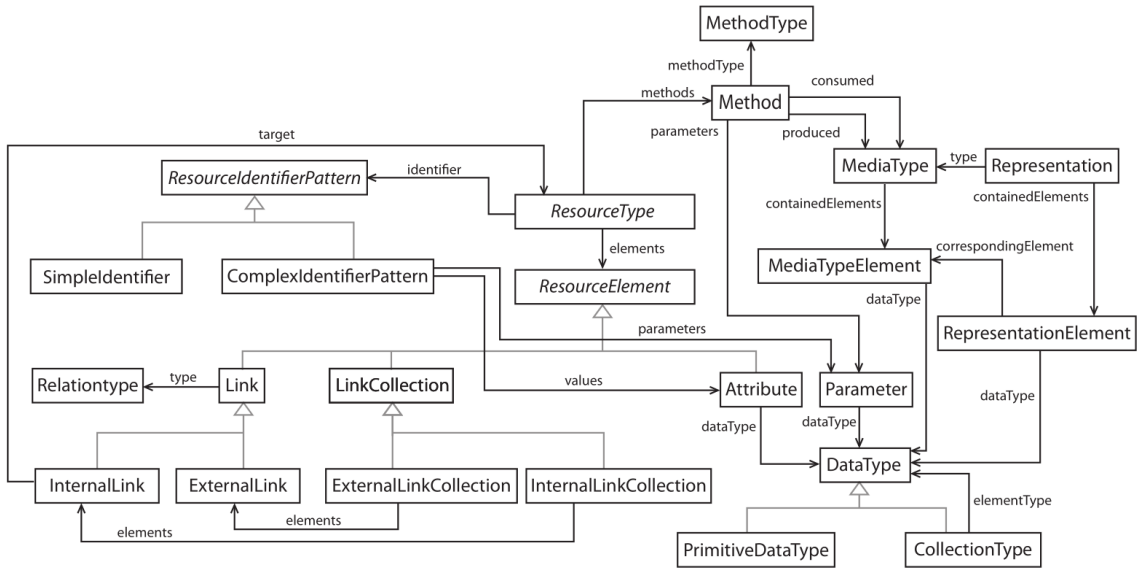


Figure 4.1.: Proposed structural meta model from [Sch11]

elements to model HATEOAS. The structural model also contains other elements like *Method*,

MethodType and *Parameter*, to define the interface, as well as *MediaTypes* and *Representation*. An important difference to the proposed meta model in 2.5 is the use of *ResourceTypes*. Figure

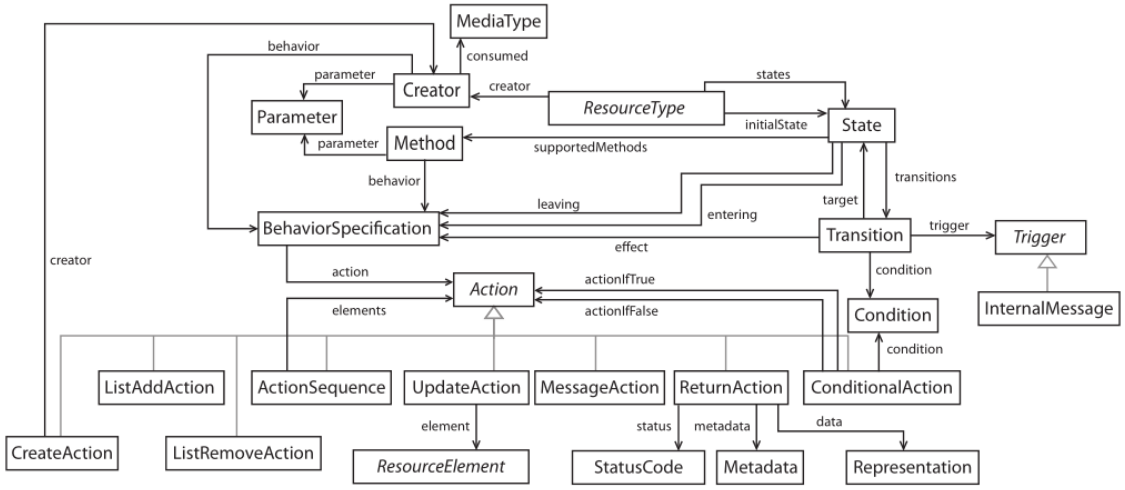


Figure 4.2.: Proposed behavioural model from [Sch11]

4.2 shows the behavioural model that enables the modeling of behaviour for ResourceTypes. Each ResourceType defines a set of *States* with one of the states being the initial state. A state can have several outgoing *Transitions*. Each of the transitions has exactly one target state. Each Transition defines *Triggers* and optionally a *Condition*. The Trigger can be an *InternalMessage* for example. There is not much information about *Conditions*, since they are not modeled yet, but it can be assumed that a transition can only be executed if a specified condition is met. States can also be associated with methods to define which method is supported while in a certain state. To not only change states, but also add behaviour to the model, Schreier also describes *BehaviourSpecifications* that can be associated with states to be enacted upon entering or leaving a certain state. Also the BehaviourSpecification can be associated with a transition, so a behaviour is triggered upon the execution of a certain transition.
The behaviour of *Methods* is also specified through BehaviourSpecifications. The BehaviourSpecification is mainly specified through *Actions* and *ActionSequences*. The latter is an ordered set of *Actions*.
The structual meta model is similar to the proposed meta model in chapter 2.5, however the proposed structural meta model of Schreier proposes eight specific resource types, each with its own set of *States* and *Transitions* from the behavioural model. Additionally, a behavioural model is introduced to describe specific resource types. This enables the modeling of a REST API including its intended behaviour. However Schreiers model is not yet finished and only described in a coarse grained fashion. The model is suitable for modeling REST APIs, but the introduced complexity makes it hard to use as a starting point for model driven development, especially since the model isn't fully specified.

4.2. “Towards a Model-Driven Process for Designing ReSTful Web Services”

In their paper *Towards a Model-Driven Process for Designing ReSTful Web Services* [LSS09] Laitkorpi et. al. describe a model driven process to transform functional specifications, expressed as arbitrary actions, to a resource oriented API. Laitkorpi et. al. introduce REST in a brief fashion highlighting benefits on architectural properties, like interoperability, evolvability, and scalability, gained through compliance with the REST constraints. A focus lies on the Uniform interface constraint that is described through three dimensions: resources and links, uniform operations to manipulate resource state information, and data types to represent the state information. It is stated that many existing services that are called “RESTful” don’t really embrace the REST constraints. They claim that major deviations from the REST principles occur during the service design when functionality is mapped to concrete API elements, like resources. The design of RESTful APIs requires to identify resources and their interconnecting links, their identification through URIs, the selection of suitable HTTP operations, and the definition of data formats and corresponding MIME media types.To emphasize the difficulties in this approach figure 4.3 shows the design gap between a functional specification and the resulting REST API. The question is raised how to best add seats to a flight by using domain specific concepts as resources. The left side shows a functional specification (as UML sequence diagram), the right-hand shows a possible REST API to add seats to a flight and retrieve the new price. In the following, the model driven process to refine functional specifications to a

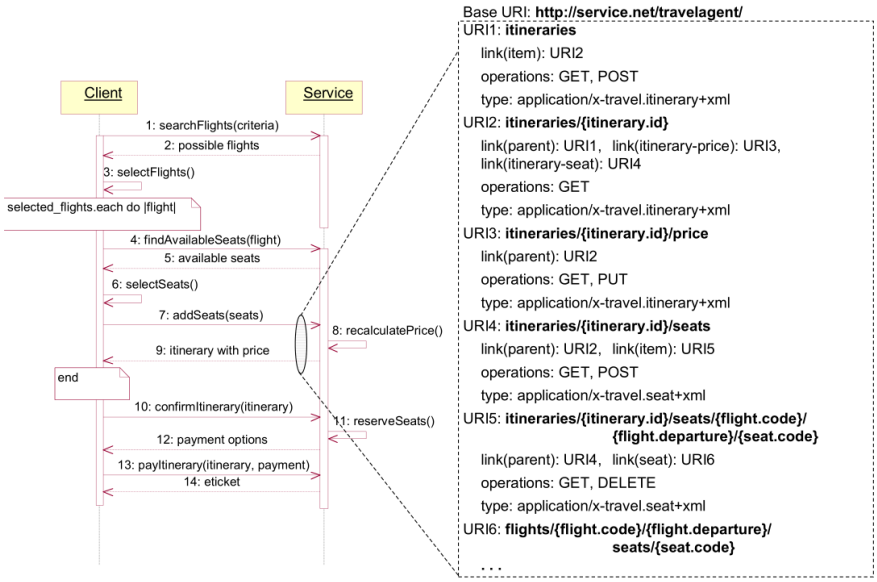


Figure 4.3.: Design gap between functional specification and corresponding REST API [LSS09]

resource oriented API is described through five phases.

1. **Analysis:** The Analysis phase uses textual use cases and other requirement specifications to provide a UML sequence diagram. The resulting diagram is assumed to cover all requirements of the API provided as functional specification consisting of top-level interactions between a client and the API expressed as UML sequence diagram. The specification also provides business states and high-level classes representing the domain vocabulary. The Analysis phase aims to capture interactions between the service and to identify side effects that are relevant to the client. It is reasoned that a high amount of domain concepts embedded in the functional model leads to a process that is less dependent on human effort.

2. **Behavioural canonicalization:** During this phase top-level interaction are analysed and each instructive operation is generalized to a uniformly expressed state manipulation. The idea is to use an approach based on speech to identify the intent of the specified operations and other relevant information that is broken down to the following concepts:

   Listeners are the target for an interaction and *primary holders* of state information that is accessed or manipulated. Listeners are modeled as «addressee» classes.

   Bystanders are secondary state holders that provide additional context for «addressee» classes. Each «addressee» can have zero or more bystanders for each interaction. They are modeled as «bystander».

   Relationships between «addressee» and «bystander» are modeled as composite or directed associations with multiplicity. Relationships are modeled as «owns» or «knows».

   Qualifiers for relationships between listeners are used to specify under which conditions listeners are relevant for an interaction in terms of selected attributes. Attributes identifying a concept (e.g. a listener) can be marked with «id», e.g. the id attribute of an Itinerary.

   Intention is used to define the purpose of a message. Intentions can be «stateInquiry» (retrieval) or «stateChange

   Effect of a message defines the result of an interaction. Effects are expressed as state manipulation primitive on the «addressee». For example there is no effect for a «stateInquiry». Effects are modeled as «inspect» (retrieval), «create» (creation of a listener), «replace» (updating with the content), or «remove» (deleting).

   Content of a message is modeled as attributes from «addressee» classes and can be marked as «input» or «output» .

   The result of the behavioural canonicalization is a information model that covers state information. The transformation in this phase of the process is aided by supplementary questionnaires.

3. **Structural canonicalization:** This phase transforms the information model to a resource model that contains interconnected resource entities. Concepts of structural canonicalization are:

   Item , modeled as «item». Each «addressee» or «bystander» results in an item.

   Container is a collection of «item» instances. It is modeled as «container».

   Projection is an additional class that is neither «item» nor «container». A «container» filtered with certain criteria is a «projection» of the «container».

   Sub-resource associations reflect the hierarchy of state information. Derived from «owns» relationships, modeled as «sub».

   Reference associations reflect the «knows» relationships and are modeled as «ref». Each «projection» has a «ref» association to the «item» contained by the projected «container».

   The result of this phase is a resource model that represents a layer on the information model representing resources and their relationships.

4. **Service translation:** This phase uses the resource model as input and produces an output that can be used by implementations tools. The outcome of the service translation described in the paper includes an URI structure, links between resources, HTTP methods for each URI and MIME types as representations. A *Resource hierarchy* is identified through navigating the «sub» associations in the resource model. «id» attributes are used as identifier segments in the URI. «sub» associations become navigable links representing hierarchical relationships for example */itineraries/id/price*: *itineraries* and *price* are generated from «sub» relationships, while {*id*} is derived from the «id» of the itinerary «item». The HTTP methods are derived from the modeled effects: «inspect» maps to GET, «create» maps to POST, «replace» maps to PUT, and «remove» maps to DELETE. MIME types are generated based on the selected resource data content as attributes in the resource classes. The output of this phase is a complete service specification.

5. **Code generation:** The last phase of the process is the transformation of the result of the previous phase to software artifacts. It is assumed that the service specification is in a machine processable format, but the actual code generation was omitted in the paper.

The paper uses an ongoing example and shows the results for each step based on the example shown in figure 4.3. The proposed approach uses UML sequence diagrams as input and uses several model-to-model transformations to refine a service specification that can be used for implementation or source code generation. The complete approach is visualised in figure 4.4. During the analysis the first formal model is defined as a high-level sequence diagram, which is transformed to a information model that covers state information. The information model is then transformed to a resource model that covers the structure of the REST API, which is then

transformed to a complete service specification. In terms of the proposed approach this thesis would start with the resource model. Although the proposed approach mentions the generation of application code, the code generation is omitted in the paper.

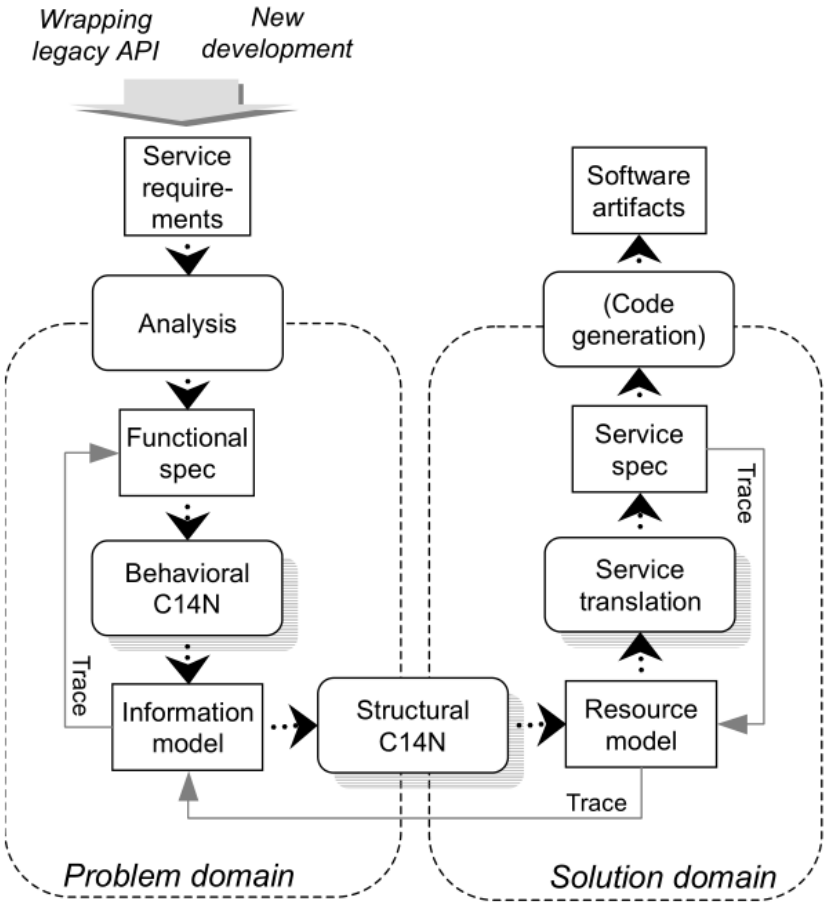
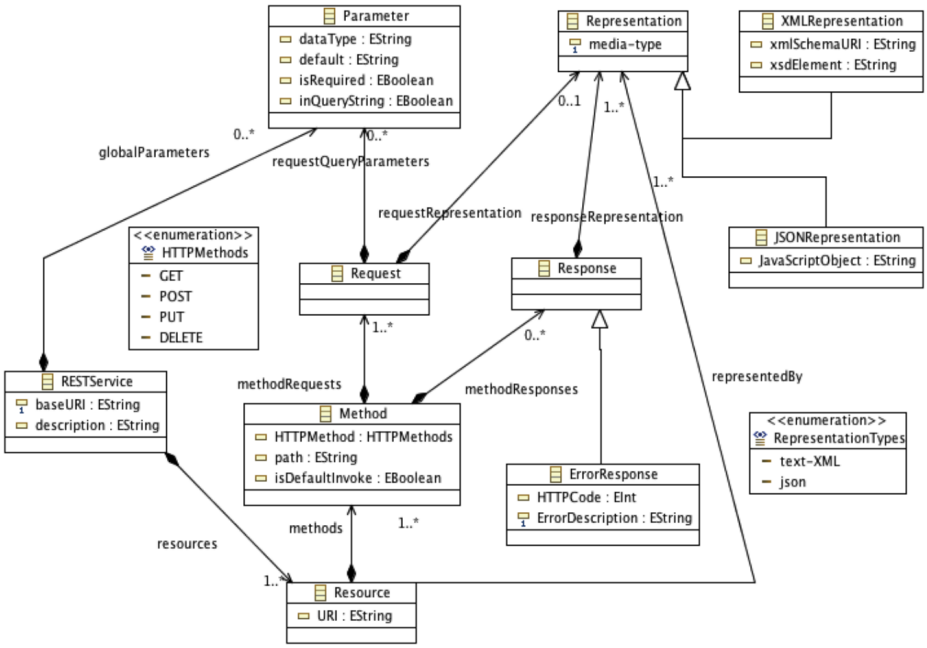


Figure 4.4.: Overview of the proposed approach [LSS09]

4.3. "Dealing with REST Services in Model-driven Web Engineering Methods"

In their article *Dealing with REST Services in Model-driven Web Engineering Methods* [VP09] Valverde and Pastor propose a meta model to describe REST Services for use in model driven development. Valverde and Pastor follow two goals with the definitions of a meta model for REST applications. One is to provide abstraction from technical complexity for analysts, e.g.

domain experts do not have to deal with technical issues. The second goal is to provide a human readable notation to deal with REST Services within web application development. According to them, previously proposed approaches to use model-driven engineering in development of web services rely on the formal specification of those services through WSDL. Figure 4.5 shows a graphical representation of the proposed meta model. As can be seen the meta model provides a generic set of elements suited to describe RESTServices. An instance of this meta model is basically a set of *Resources* associated with *Methods* and *Representations* and a possibly empty set of global parameters. The meta model seems to be suitable to describe a RESTService however it lacks the capabilities and expressiveness to show how resources are interconnected and is therefore not suited to approach HATEOAS.



»

Figure 4.5.: Proposed REST Metamodel from [VP09]

5. Reference Application

This chapter introduces the reference application developed in this thesis. It gives an overview over the design, modeling, and implementation of the reference application and shows the concept for domain logic integration.

5.1. Introducing Restbucks

Restbucks is a small application used as a running example in *Rest in Practice* [WPR10] to show best practices in the development of REST applications. *Restbucks* is a small coffee shop that digitalizes the whole process of ordering, paying, and receiving a coffee. For this thesis, *Restbucks* was expanded and adapted to capture additional REST style interactions such as a long running task or the use of query parameters. The basic idea is to place an order, pay it, and receive a receipt. So there are three different entities that must be linked through hypermedia. *Order*, *Payment*, and *Receipt*. The hyperlink relationships are depicted in figure 5.1. *CoffeeShop* is the root resource and provides hyperlinks to *OrderList*, *Payment*, and *Receipt*. *Orderlist* points via hyperlink to *Order*. *Order* points to *Payment*, which points to *Receipt*.

5.2. Model

Restbucks is composed of five distinct resources, three of which having their own data structure, one serves as a starting point and root resource, and one serves as a list with sorting and paging capabilities. *Restbucks* was adapted to show the capabilities of the modeling tool. The exact model can be seen in figure 5.2. The three resources with domain entities are *Order*, *Receipt*, and *Payment*. An *Order* can be placed via the *OrderList* resource. Each *Order* is associated with exactly one *Payment*. To complete an *Order* two steps have to be performed. The *Payment* corresponding to an *Order* has to be updated (aka. paid) and then the *Receipt* associated with the *Payment* has to be deleted (aka. received). This process will be guided by hyperlinks embedded in the resources representations. Figure 5.3 shows the ordering process in detail as UML sequence diagram anlisting 5.1 shows the contents of the messages.

CoffeeShop: The *CoffeeShop* resource is the root resource of the application. It serves to give an entry point and an overview. It has three *Navigations*. The *Navigation* to *OrderList* is *ONE_TO_ONE*, the *Navigations* to *Payment* and *Receipt* are *ONE_TO_MANY*. This means

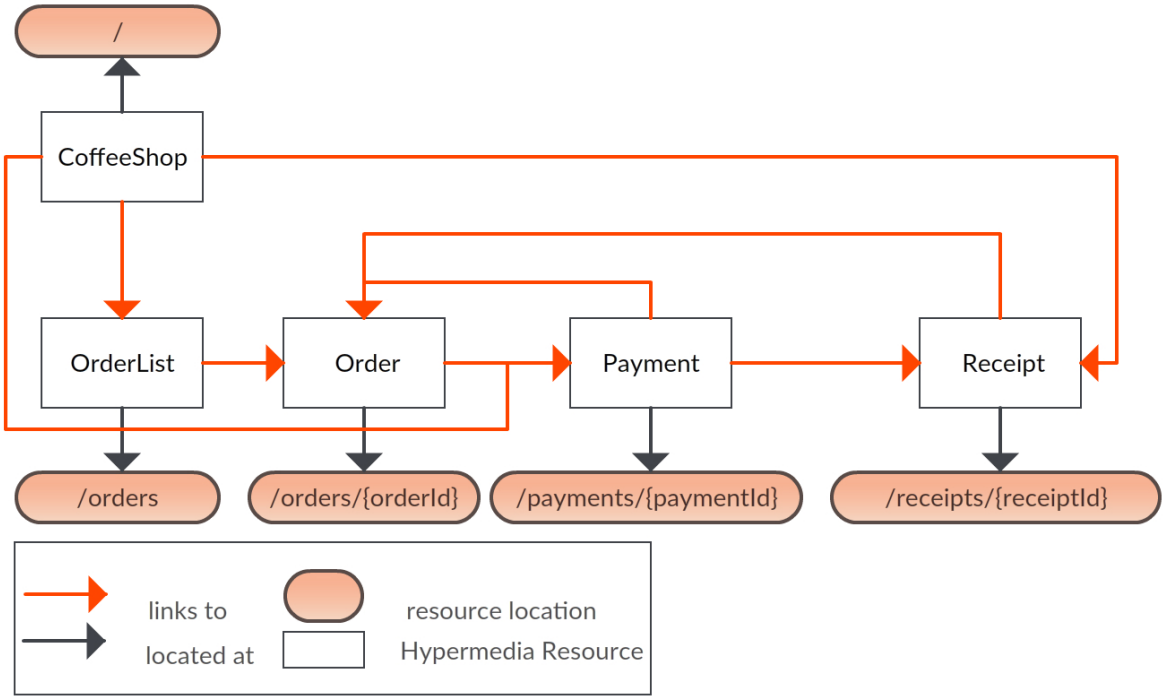


Figure 5.1.: Depiction of the relationships, adapted from [WPR10, figure 5-9]

that the representation of *CoffeeShop* will have one link to *OrderList* and several links to the corresponding instances of *Payment* and *Receipt*.

OrderList: The *OrderList* resource is intended to provide a list of *Orders*. There are five *Navigations* originating from the *GET* method of the *OrderList* resource, four of which are self referencing. The self referencing Navigations are intended to provide paging capabilities. For this, there are some query parameters defined for the *GET* method. Each self referencing navigation will provide its own set of values for the query parameters, for example the *nextPage* can be realized by calculating the offset (starting index of a sorted list of *Orders*). Since *Navigations* are relationships, the *OrderList* is presented as a list of hyperlinks to particular orders. The *Navigation allOrders* provides links to all *Orders* and realizes the connection between *OrderList* resource and *Order* resource. The *POST* method comes with two *Interactions*: *archiveList* emulates a long running task, and *createOrder* is supposed to serve as controller to instantiate new *Orders* as indicated by the green arrow (*ResourceCreation*).

Order: The *Order* resource represents an actual domain object and has an internal data structure. It is a CRUD style resource with *GET*, *PUT*, and *DELETE*. It has one *Navigation* pointing towards the *Payment* resource. This link exists as soon as the *order Order* is

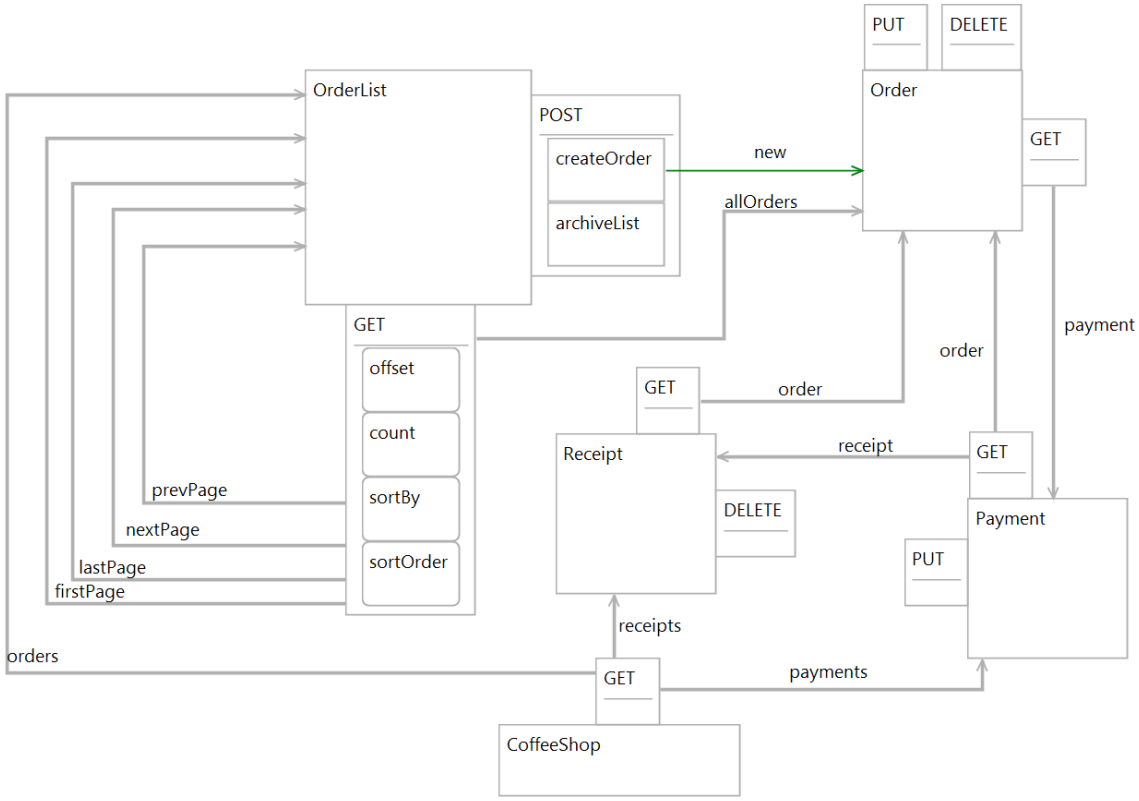


Figure 5.2.: *Restbucks* as modeled in the modeling tool.

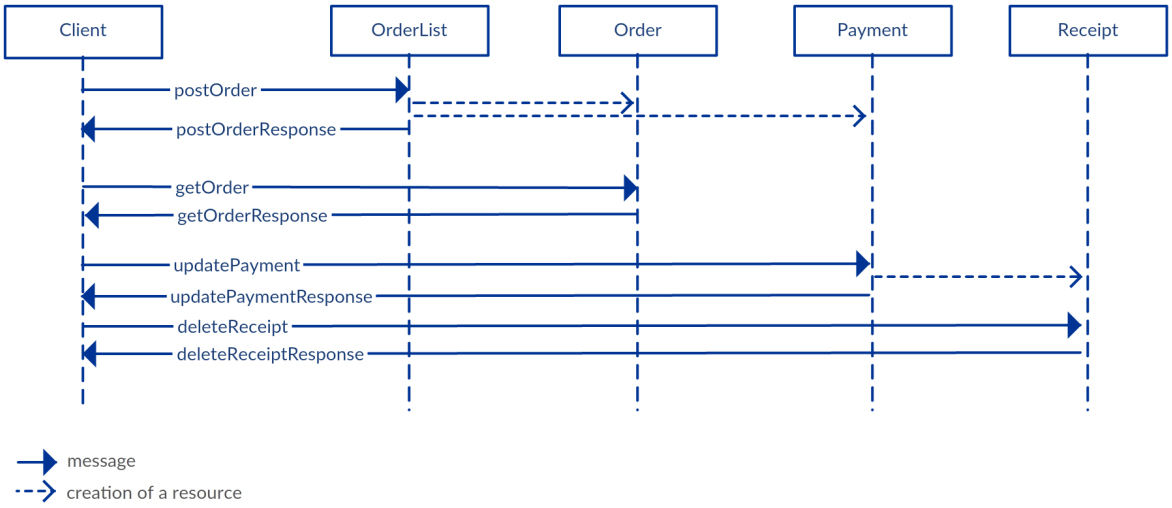
created since the *Order* and *Payment* resources are created through the same request to the *OrderList*, see 5.3.

Payment: The *Payment* resource has an internal data structure and provides a link to its *Order* and to its *Receipt*. As shown in figure 5.3 the *Receipt* resource is created on updating the *Payment* resource to the status of being paid. The *Payment* resource has a *GET* method to safely explore it and a *PUT* method to update it (i.e. to make a payment).

Receipt: The *Receipt* resource has an internal data structure and provides a link to its *Order*. The *Receipt* resource has a *GET* method to safely explore it and a *DELETE* method to conclude the process.

5.3. Design of the Restbucks Implementation

The design of the implementation of *Restbucks* is shown in figure 5.4. The implementation is composed of five major building blocks. The *Application*, the *Resources*, the *Model*, the



Message content is listed in listing 5.1

Figure 5.3.: Ordering process in restbucks.

DomainLogicProviderRegistry, and the *DomainLogicProvider Implementations*. Except for the *DomainLogicProvider Implementations* all architectural units are supposed to be generated by the modeling tool. The *DomainLogicProvider Implementations* are supposed to be provided by the *User* of the modeling tool. The term *User*, that will be used in the following, describes the developer using the modeling tool to design and realize a REST API.

5.3.1. Brief introduction of JAX-RS

JAX-RS [jaxa] is short for *Java API for RESTful Web Services*. JAX-RS is a specification of an API for the Java programming language that enables and standardizes the development of RESTful web services. JAX-RS specifies annotations and associated classes/interfaces that can be used with Java classes to expose them as REST resources. It is assumed that HTTP is used as network protocol, so JAX-RS provides high level support for HTTP concepts. There are several implementations of the JAX-RS specification. This thesis uses Jersey [jer], which is also the reference implementation of JAX-RS. The specified annotations most relevant for this thesis are the following:

@Path: The @Path annotations specifies a path for a resource class. JAX-RS considers Java classes as resource classes when they have at least one method annotated with @Path or a request method designator (see next). The @Path annotation can be used on class level to specify a root resource class. In JAX-RS terms root resource classes provide roots

Listing 5.1 Restbucks messages and answers

postOrder	POST /orders HTTP/1.1
	`<Order>`
	`...`
	`</Order>`

postOrderResponse	HTTP/1.1 201 Created Location:/orders/1
getOrder	GET /orders/1 HTTP/1.1

getOrderResponse	HTTP/1.1 200 OK
	`<Order>`
	`<Link href="/payments/1" title="payment">`
	`</Order>`

updatePayment	PUT /payments/1 HTTP/1.1
	`<Payment>`
	`<Status>paid</Status>`
	`</Payment>`

updatePaymentResponse	HTTP/1.1 200 OK
	`<Payment>`
	`<Status>paid</Status>`
	`<Link href="/payments/1" title="payment">`
	`</Payment>`

deleteReceipt	DELETE /receipts/1 HTTP/1.1
deleteReceiptResponse	HTTP/1.1 204 No Content

of resource class trees, which means there can be several root resource classes. When @Path is used on a method it identifies a sub-resource method or locator.

@GET, @PUT, @POST, @DELETE: These are request method designator defined by JAX-RS. They are used to map HTTP requests to the methods that handle specific requests, e.g. @GET handles HTTP GET requests, etc.

@Produces, @Consumes: These annotations can be applied to a resource method (a method annotated with a resource method designator). They can be used to specify the media types that are supported by the method. @Consumes is used to specify which media types the resource accepts and @Produces is used to specify which media types the resource can provide. These annotations are assisted by the *MessageBodyReader* and

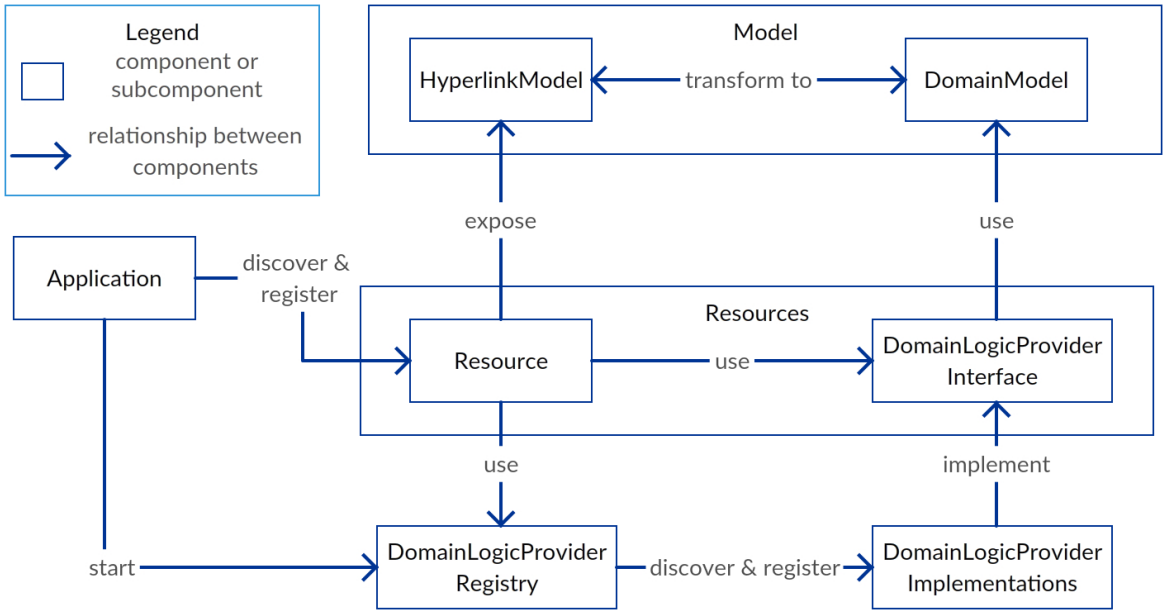


Figure 5.4.: Design of *Restbucks*

MessageBodyWriter interfaces, that have to be implemented for each specified media type. Jersey provides implementations for many of the most common media types.

@Context: The @Context annotation can be used to inject objects of a certain type into resource class fields or method parameters. Types that can be injected include instances of *UriInfo*, which provide access to the static and dynamic parts of the requested URI, and *HttpHeaders*, which provides access to HTTP request header information.

@PathParam, @QueryParam: These annotations can be used to extract certain path parameters or query parameters from the requested URI and inject them into class fields or method parameters. Assuming a class is annotated as follows:

```
@Path("orders/{orderId}")
```

Then the injection of the dynamic URI part *orderId* can be realized as follows:

```
@PathParam("orderId") String id;
```

This declaration can be used either as a class field or as a method parameter.

5.3.2. Model

The *Model* of *Restbucks* is separated into two parts. The *DomainModel* contains simple Java classes and the *HyperlinkModel* provides HATEOAS capabilities. The basic idea of the

separation is to encapsulate ideas and concepts of REST from the Java environment (also see impedance mismatch discussed in chapter 2.4) . In the REST metamodel, *Navigations* represent the relations between resources. Relationships are realized by creating fields in Java with the other *DomainModel* classes as data type according to the *Navigations*. The user only needs to work with *DomainModel* classes and associate them with each other. The *HyperlinkModel* classes use those fields to create the hyperlinks to be embedded in their representations.

DomainModel: The *DomainModel* is a set of POJOs (Plain Old Java Object). It is directly derived from the resources, their data structure, and *Navigations*. Each *Navigation* results in a field that can be used to show the associations between objects. The *DomainModel* is intended to provide domain specific classes for the user of the modeling tool to work with. It is used by the *DomainLogicProviderInterfaces* as input and return values, and by the *HyperlinkModel* to construct hyperlinks for representation.

HyperlinkModel: The *HyperlinkModel* is the major unit for the realization of HATEOAS. Each POJO of the *DomainModel* has a corresponding class in the *HyperlinkModel*. Instances of *HyperlinkClasses* can only be instantiated with an instance of the corresponding *DomainClass* and information about the address of the *Resource*, that was called. During instantiation the *HyperlinkClass* calculates the proper links for display. *HyperlinkClasses* are directly used to generate resource representations.

Examples for a *DomainModel* class and a corresponding *HyperlinkModel* class can be seen in listing 5.2. The *Order* has fields for its data structure and relationships. Instances of *Order* can be constructed with values or with an instance of the corresponding *OrderHyperlink* class. The *OrderHyperlink* class only has fields for the data structure. Relationships to other resources are extracted during the construction of its instances and stored as *RESTHyperLink* objects. Another important difference is the presence of *Annotations* used for automatic (de-)serialization. The information about what path segments have to be used for the links are extracted from the *DeploymentModel*.

5.3.3. Resources

The *Resources* module is the main module to provide JAX-RS related logic. The *Resources* module contains all *Resource* classes and provides all HTTP functionality. The *Resource* classes expose the *HyperlinkModel* classes to the client. The other package of the *Resources* module are the *DomainLogicProviderInterfaces*. Every *Resource* class has a corresponding *DomainLogicProviderInterface* that provides the *Resource* with *DomainModel* objects. Those objects are transformed by the *Resource* to their associated *HyperlinkModel* object. The complete process from a HTTP request to the sending of the response is illustrated in figure 5.5. The payload of the request message is a *HyperlinkObject* in serialized form. It is deserialized by the *Resource* on arrival and then transformed into a *DomainObject*. This

Listing 5.2 Example for manually developed domain and hyperlink model classes

```
1  public class Order {
2    // data structure
3    public String orderId;
4    public String customerName;
5    // relationships
6    public Payment payment;
7    // queryparameter
8      //this class doesn't use query parameters
9    // constructors
10   public Order(String orderId, String customerName, Payment payment) { ... }
11   public Order(OrderHyperlink order) { ... }
12 }
13
14 @XmlRootElement(name = "Order")
15 public class OrderHyperlink {
16   //Link representations for HATEOAS
17   @XmlElements(value = { @XmlElement(name = "Link") })
18   public List<RESTHyperLink> restHyperLinks = new ArrayList<RESTHyperLink>();
19   // data structure
20   @XmlElement
21   public String orderId;
22   @XmlElement
23   public String customerName;
24   // The constructor adds links to related resources, if they are present.
25   public OrderHyperlink(Order order, UriInfo ui) {
26     this.orderId = order.getOrderId();
27     this.customerName = order.getCustomerName();
28     String baseURI = ui.getBaseUri().toString();
29     if (order.getPayment()!= null) {
30       this.addRESTHyperLink(new RESTHyperLink(baseURI + "payments/" +
           order.getPayment().getPaymentId(), "payment"));
31     }
32     this.addRESTHyperLink(new RESTHyperLink(baseURI + "orders/" + this.uniqueId,
         "self"));
33   }
34 }
```

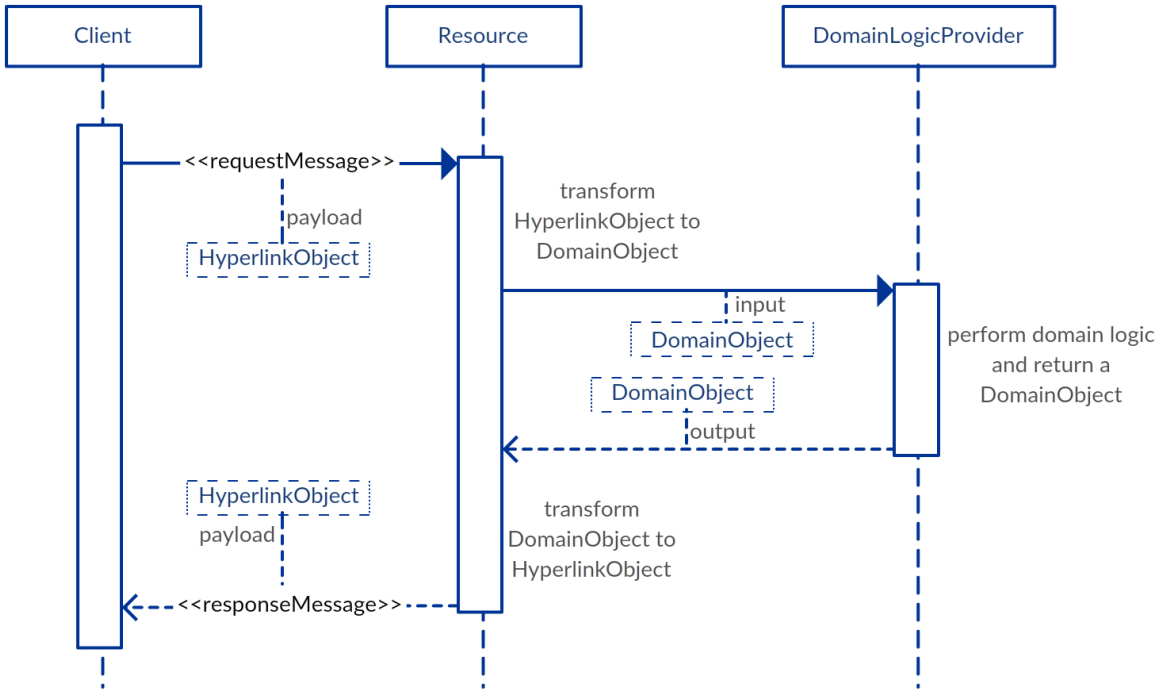


Figure 5.5.: Processing of a HTTP Request with request payload and response payload

DomainObject is given to a *DomainLogicProvider* which performs the domain logic and returns a *DomainObject* to the *Resource*. The *Resource* transforms this *DomainObject* to a *HyperlinkObject* and serializes it as payload for the response message.

5.3.4. Other modules

Besides the *Model* and the *Resources* there are three smaller modules in the architecture of the implementation. The *Application*, the *DomainLogicProviderRegistry*, and the *DomainLogicProviderImplementations*.

Application: The *Application* module is the basis of any application generated with the modeling tool. The *Application* scans the project for *Resources* and registers them with the HTTP server.

DomainLogicProviderRegistry: The *DomainLogicProviderRegistry* is the main module to integrate domain specific logic. It scans the classpath for implementations of *ResourceInterfaces* and instantiates them. Also the *DomainLogicProviderRegistry* is responsible for delivering those instances to the *Resources*. A detailed description of the integration mechanism is given in chapter 5.4.

DomainLogicProviderImplementations: This module comprises all domain specific logic and is the only module that has to be provided by the user of the modeling tool. The integration of domain logic is explained in chapter 5.4.

5.4. Domain Logic Integration

In chapter 3.1 the task of this thesis was described in detail. One requirement of the thesis is, that the solution must provide a loosely coupled way of integrating user provided domain logic into the generated applications.
The intention is, that no generated code has to be modified for the domain logic to be integrated into the resulting application. This brings several advantages:

- The user doesn't need to understand the generated code. The user only needs to understand the interfaces necessary for the integration of the manually written code. As long as these interfaces remain stable the generated code can be improved at will without additional work of adapting manually written classes.
- Without any handmade modifications on generated code, there is no risk that the user might break anything. This requires that the interfaces are designed this way. This can be achieved by following the principle of *information hiding* which means that the user is only provided with information relevant to domain logic. The user should not be given access to objects that could affect the intended behaviour of the generated code.
- There is no need for merging while generating code. All generated classes can simply be overwritten.
- Generated code and user code can be managed separately.

Figure 5.4 shows that the *DomainLogicProvider Implementations* are connected to the *DomainLogicProviderInterfaces* and the *DomainLogicProviderRegistry*. Every *Resource* in the *Resources* module specifies a specific *DomainLogicProviderInterface* that is used to provide the domain logic for the *Resource*. After the generated application is started, the *Application* module also starts the *DomainLogicProviderRegistry*. The *DomainLogicProviderRegistry* then scans the classpath of the application via reflections and instantiates at least one implementation of any *DomainLogicProviderInterface* it can find. This mechanism allows that the user of the modeling tool only has to implement the *DomainLogicProviderInterface*.

6. JAX-RS PSM Metamodel

This chapter introduces the platform specific meta model that has been developed to generate the modeled application. The platform specific JAX-RS meta model is provided as an Ecore model and its instances are the only source model for code generation. JAX-RS models are constructed by transforming a deployment model, a resource model and some user input. In the following, the most important model elements are explained. Figure 6.1 shows an UML diagram of the complete JAX-RS PSM metamodel. The detailed listing of all element description can be found in appendix A.1. The JAX-RS PSM Metamodel features several similar constructs to the resource meta model. There are different ways to approach this issue. The JAX-RS metamodel already references elements that are defined in the resource model. So the JAX-RS metamodel could also reference other elements defined by the resource model and extend them to fulfil the platform specific requirements. Another approach would be to define wrapper classes that can reference to actual elements in the resource model, which would have the advantage that the transformation from the resource model to the JAX-RS model would require less effort to copy information by simply referencing other elements. Both of these approaches were dismissed in favour of a third possibility. Constructs similar to those of the resource model are defined in the JAX-RS meta model tailored to suit its needs. This introduces an overhead in copying information between models, but also introduces independence between the models. The resource model is in a state where it is highly unlikely that there won't be any changes in the future. Changes in the resource model also require that the transformation from resource model to JAX-RS model have to be checked if they are still valid. But if the JAX-RS model were to reference or inherit from changed constructs it would require a significantly higher effort. The JAX-RS PSM metamodel would have to be checked if the semantics of the changed elements are still valid. The model-to-model transformation would have to be checked and most importantly the model-to-text transformation would have to be adapted. This way the model-to-text transformation is more stable and can be improved independent from changes to the resource model.

6.1. JaxrsModel

The *JaxrsModel* element is the root element of all JAX-RS PSMs. It has several project related attributes like *projectName*, that have to be provided by the user and can not be derived from the *ResourceModel* or the *DeploymentModel*. Most importantly, the *JaxrsModel* holds two lists.

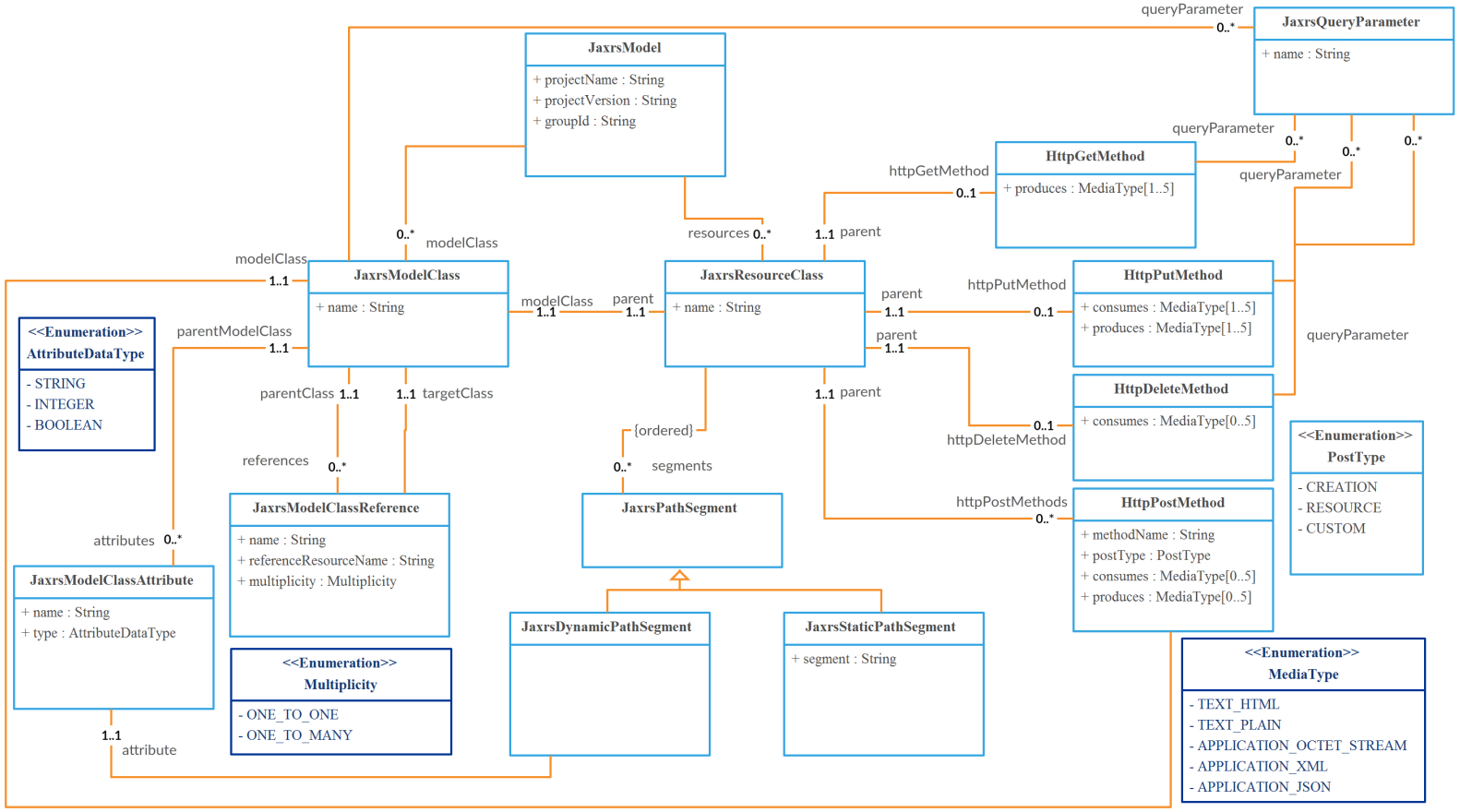
One list with all *JaxrsResourceClass* elements and one list with all *JaxrsModelClass* elements that belong to the PSM. So the *JaxrsModel* element holds all information necessary to generate application code.

6.2. JaxrsResourceClass

The *JaxrsResourceClass* element is the basic element for later generation of resource classes. This element also knows its relative address realized through an ordered list of *JaxrsPathSegments*. Each *JaxrsResourceClass* element has a reference to a *JaxrsModelClass* element, which represents its data model. *JaxrsResourceClass* elements also hold all information about offered HTTP methods . A *JaxrsResourceClass* can have one of each *HttpPutMethod*, *HttpGetMethod*, *HttpDeleteMethod*, and several *HttpPostMethod* elements. Each of those elements stores its consumed and produced mediatypes. All method elements except for the *HttpPostMethod* element can also have *JaxrsQueryParameter*. The *HttpPostMethod* additionally references a *JaxrsModelClass* class, because *HttpPostMethod* elements do not necessarily work with the same data as its parent *JaxrsResourceClass*. An example for this is the *OrderList* resource class described in chapter 5.2. Tranformed to a *JaxrsResourceClass*, the *OrderList* resource will have two *HttpPostMethod*s. One is designated to create new instances of the Order resource class, so this *HttpPostMethod* references the *JaxrsModelClass* directly derived from the Order resource. The *archiveList HttpPostMethod* references a *JaxrsModelClass* specifically generated for this *HttpPostMethod*.

6.3. JaxrsModelClass

JaxrsModelClass elements are the core elements to generate the model classes of the application code. *JaxrsModelClass* elements also reference to a parent *JaxrsResourceClass* but are not necessarily the *JaxrsModelClass* element referenced by the *JaxrsResourceClass*. There may be more *JaxrsModelClass* elements than *JaxrsResourceClass* elements since each *HttpPostMethod* in the PSM can have its own instance of *JaxrsModelClass*. *JaxrsModelClass* elements can have several *JaxrsQueryParameter* elements which are derived from the associated *JaxrsResourceClass* elements methods. Furthermore all relations between resources are reflected through *JaxrsModelClassReference* elements, that represent those relations through a *parentClass* and a *targetClass*. *JaxrsModelClassAttribute* elements are used to realize the data structure and represent the attributes of the resulting application code classes. Also the *JaxrsModelClassAttribute* elements are referenced by instances of *JaxrsDynamicPathSegment* to indicate identifying attributes of the *JaxrsModelClass* elements.



The enumerations *AttributeDataType*, *Multiplicity*, and *MediaType* are imported from the resource meta model.

Figure 6.1.: UML diagram of the JAX-RS PSM meta model

7. Model Transformations

This chapter shows the complete process of generating application code from the point where the DeploymentModel (see 2.6) and the ResourceModel (see 2.5) are provided. The process is depicted in figure 7.1. The DeploymentModel and the ResourceModel are transformed into a JAX-RS PSM, which is enhanced by user input and finally transformed into application code.

7.1. Transformation to PSM

This chapter describes how the elements of the JAX-RS PSM are transformed from elements of the DeploymentModel and the ResourceModel. This chapter gives an overview over the model-to-model transformation. A detailed description can be found in appendix A.2. The

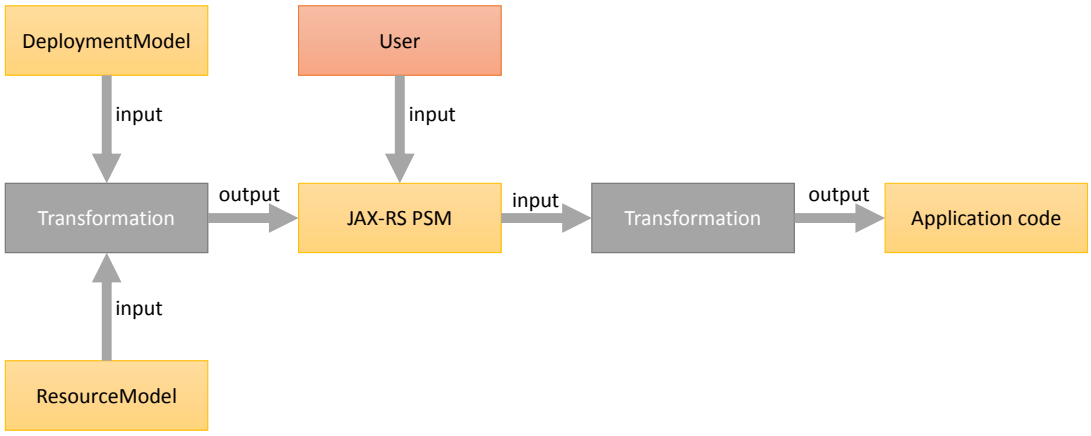


Figure 7.1.: The complete transformation sequence of this work.

transformation assumes that all requirements listed below are met.
Requirements:

- The DeploymentModel must resemble a tree, originating from the root resource.
- The DeploymentModel and the ResourceModel must be valid. This means that they have to be defined in the given meta models (chapters 2.6 and 2.5). Also they must comply to validation rules defined by previous works.

The root element *JaxrsModel* of the JAX-RS PSM is transformed from its counterpart of the *ResourceModel*. During the transformation process the user is prompted to provide project specific information like the *projectName*, or the *projectVersion*. It is possible to make changes to the generated JAX-RS PSM via a graphical editor.

7.1.1. Transformation to JaxrsResourceClass

The *JaxrsResourceClass* is derived from the *Resource* element of the *ResourceModel* and from *Mapping* elements of the *DeploymentModel*. Each *Resource* in the *ResourceModel* results in one *JaxrsResourceClass*. To calculate the appropriate URI for a *JaxrsResourceClass* (through ordered *URLFragments*) the *Mappings* have to be traced towards the root from the *Mapping* where the corresponding *Resource* is the target. The result is an ordered list of *URLFragments* that directly represent the relative URI towards the root resource. Also the different *Method* elements of each *Resource* in the *ResourceModel* are transformed to their counterparts in the *JaxrsResourceClass*. Note that the *Resource* elements can only have one *PostMethod* with several *Interactions*. Those *Interactions* are transformed to *HttpPostMethod* elements within the *JaxrsResourceClass*. Each *JaxrsResourceClass* has a distinct *JaxrsModelClass* element, while each *HttpPostMethod* can either have its own *JaxrsModelClass* or point to the *JaxrsModelClass* element of any *JaxrsResourceClass*. So there might be more *JaxrsModelClass* elements than *JaxrsResourceClass* elements in the PSM.

7.1.2. Transformation to JaxrsModelClass

The *JaxrsModelClass* elements can originate from two different elements of the *ResourceModel*. Either from the *Resource* element or from the *Interaction* Element. If an *Interaction* has an own *entityStructure*, a new *JaxrsModelClass* element is produced. Otherwise the *Interaction* points to a *Resource* and will use its associated *JaxrsModelClass* element. The *JaxrsAttribute* elements are transformed from the respective *entityStructure*. *Navigations* are the base elements to become *JaxrsModelClassReferences* in the JAX-RS PSM. *JaxrsModelClass* instances transformed directly from *Resources* use all *Navigations* originating from the GET, PUT, and DELETE Method of the *Resource* as well as the *Navigations* that originate from any *Interaction* that references the *Resource*. *JaxrsModelClass* instances transformed directly from *Interaction* use all *Navigations* originating from that *Interaction*. The *JaxrsQueryParameter* are constructed from

all *queryParameter* of the GET, PUT, and DELETE methods of the *Resource*. Since *Interactions* don't have any *Parameters* there won't be any *JaxrsQueryParameter* in a *JaxrsModelClass* constructed from an *Interaction*. The *parent* of the *JaxrsModelClass* is either the *JaxrsResourceClass* that is transformed from the same *Resource*, or the *JaxrsResourceClass* transformed from the *Resource* associated with the *Interaction* (via *PostMethod*).

7.2. Transformation to Application Code

The transformation from the JAX-RS PSM to Java code is basically the transformation of *JaxrsResourceClasses* and *JaxrsModelClasses*. The generated application is designed correspondent to the design of *Restbucks* and provides code according to the JAX-RS specification [jaxa] (introduced in 5.3.1). The architecture of *Restbucks* is depicted in figure 5.4 and the module descriptions can be found in chapter 5.3. To illustrate the transformated code examples are shown, that were generated from the *Restbucks* model shown in 5.2. In the following a *DomainLogicProvider* implementation is abbreviated with *DLP*.

7.2.1. Transforming the JaxrsModelClass

Each *JaxrsModelClass* is transformed into two distinct classes. A domain model class and a hyperlink model class.

- The *name* attribute will be used to generate the actual class names of the resulting classes.
- The *attributes* and *references* will be realized as fields with their respective type.
- The *queryParameter* will be defined as constants in the domain class.

Listings 7.1, 7.2, and 7.3 show the main aspects of generated hyperlink and domain classes. There are two important differences between hyperlink and domain classes. The first difference is that the hyperlink classes are annotated with JAXB-Annotations [jaxb] for automatic (de-)serialization. The second difference is that hyperlink classes don't have any fields corresponding to the *JaxrsModelClassReferences* from its source model. Instead the hyperlink classes extract links from the domain classes and offer them via *XML/JSON*.

Listing 7.1 shows the structure of domain classes. Fields are declared in an ordered fashion: The first fields are the JaxrsQueryParameter names defined as constants (line 3-6). The second group of fields are the JaxrsModelClassAttributes (line 22-23) followed by fields derived from the JaxrsModelClassReferences (line 8-12, 25). If there are JaxrsQueryParameters present an additional *HashMap* is declared to store values for queryParameters, that can be used either by the corresponding hyperlink class or by the user. Line 8-11 show four fields declared with OrderListDomain as type. These four fields result from the navigations pointing to the

Listing 7.1 Generated OrderListDomain and OrderDomain from figure 5.2, (manual implementation in listing 5.2)

```
1  public class OrderListDomain {
2    //JaxrsQueryParameter names as constants
3    public static final String OFFSET = "offset";
4    public static final String COUNT = "count";
5    public static final String SORTBY = "sortby";
6    public static final String SORTORDER = "sortorder";
7    //JaxrsModelClassReferences
8    private OrderListDomain nextPage;
9    private OrderListDomain prevPage;
10   private OrderListDomain firstPage;
11   private OrderListDomain lastPage;
12   private ArrayList<OrderDomain> allOrders = new ArrayList<OrderDomain>();
13   //to store queryParameterValues
14   public HashMap<String, String> queryParameterMap = new HashMap<String, String>();
15   public OrderListDomain() {}
16   public OrderListDomain(OrderListDomain nextPage, OrderListDomain prevPage,
         OrderListDomain firstPage, OrderListDomain lastPage, ArrayList<OrderDomain>
         allOrders) { ... }
17   public OrderListDomain(OrderListHyperlink orderListHyperlink) { ... }
18 }
19
20 public class OrderDomain {
21   //JaxrsModelClassAttributes
22   private String orderId;
23   private String customerName;
24   //JaxrsModelClassReferences
25   private PaymentDomain payment;
26   //constructors
27   public OrderDomain() {}
28   public OrderDomain(String orderId, String customerName, PaymentDomain payment) { ... }
29   public OrderDomain(OrderHyperlink orderHyperlink) { ... }
30 }
```

OrderList and originate from the GET method of OrderList defined in figure 5.2, later they are called *self references*. The intention is that the user provides these four objects with different sets of queryParameter values to be transformed to links embedded in the representation of OrderList resource. Line 12 shows how a ONE_TO_MANY relationship results in an ArrayList of objects. Moreover the examples show each three constructors. One without any arguments, one with one parameter per field (not static final), and one with the corresponding hyperlink class for an argument. Getters and setters are generated but were omitted for the examples.

Listing 7.2 Generated OrderListHyperlink from figure 5.2

```
1  @XmlRootElement(name = "OrderList")
2  public class OrderListHyperlink {
3    //Wrapper for Links
4    @XmlElements(value = { @XmlElement(name = "Link") })
5    public ArrayList<RESTHyperLink> restHyperLinks = new ArrayList<RESTHyperLink>();
6    //Empty Constructor for automatic serialization
7    public OrderListHyperlink() {}
8    //Constructor with domain model object and URI as arguments to realize HATEOAS
9    public OrderListHyperlink(OrderListDomain orderListDomain, UriInfo ui) {
10     String baseURI = ui.getBaseUri().toString();
11     String queryParametersNextPage =
           extractQueryParameters(orderListDomain.getNextPage());
12     // links, self referencing, ONE_TO_ONE multiplicity
13     if (!queryParametersNextPage.isEmpty()) {
14       this.addRESTHyperLink(new RESTHyperLink(baseURI + "orderlist" + "?" +
             queryParametersNextPage, "nextPage"));}
15     ...
16     // links to allOrders, ONE_TO_MANY multiplicity
17     for (OrderDomain orderDomainLoop : orderListDomain.getAllOrders()) {
18       this.addRESTHyperLink(new RESTHyperLink(baseURI + "orderlist" + "/" +
             orderDomainLoop.getOrderId().toString(), "allOrders"));
19     }
20     //link to this particular resource
21     String myQueryParameters = extractQueryParameters(orderListDomain);
22     if (!myQueryParameters.isEmpty()) {
23       myQueryParameters = "?" + myQueryParameters;
24       this.addRESTHyperLink(new RESTHyperLink(baseURI + this.getRelativeSelfPath() +
             myQueryParameters, "self"));
25     } else {
26       this.addRESTHyperLink(new RESTHyperLink(baseURI + this.getRelativeSelfPath(),
             "self"));
27     }
28   }
29   public String getRelativeSelfPath() {
30     return "orderlist";
31   }
32   ...
33 }
```

Listings 7.2 and 7.3 show the hyperlink classes corresponding to the domain classes shown in listing 7.1. The first line of each listing shows the annotation *@XmlRootElement* that is needed for automatic (de-)serialization. Lines 4 and 5 show the declaration of an ArrayList intended to serve as wrapper for objects of the *RESTHyperLink* class, a simple JAXB annotated class with

Listing 7.3 Generated OrderHyperlink from figure 5.2

```
1  @XmlRootElement(name = "Order")
2  public class OrderHyperlink {
3    //Wrapper for Links
4    @XmlElements(value = { @XmlElement(name = "Link") })
5    public ArrayList<RESTHyperLink> restHyperLinks = new ArrayList<RESTHyperLink>();
6    //JaxrsModelClassAttributes
7    private String orderId;
8    private String customerName;
9    //Empty Constructor for automatic serialization
10   public OrderHyperlink() {}
11   //Constructor with domain model object and URI as arguments to realize HATEOAS
12   public OrderHyperlink(OrderDomain orderDomain, UriInfo ui) {
13     setOrderId(orderDomain.getOrderId());
14     setCustomerName(orderDomain.getCustomerName());
15     String baseURI = ui.getBaseUri().toString();
16     //link to payment, ONE_TO_ONE multiplicity
17     if (orderDomain.getPayment() != null) {
18       this.addRESTHyperLink(new RESTHyperLink(baseURI + "payments" + "/" +
           orderDomain.getPayment().getPaymentid().toString(), "payment"));
19     }
20     this.addRESTHyperLink(new RESTHyperLink(baseURI + this.getRelativeSelfPath(),
         "self"));
21   }
22   //only the getters are marked as XmlElements
23   @XmlElement
24   public String getOrderId() { ... }
25   @XmlElement
26   public String getCustomerName() { ... }
27   ...
28   public String getRelativeSelfPath() {
29     return "orderlist" + "/" + this.getOrderId().toString();
30   }
31 }
```

fields for a link description (*href* for the URI and *rel* for the type). The main HATEOAS engine is realized in the constructor that uses a domain object and an object of type UriInfo (see 5.3.1) as arguments (lst 7.2: line 9, lst 7.3: line 12). Each field in the domain class that is derived from a JaxrsModelClassReference is accessed to extract a link. As can be seen in listing 7.2, fields resulting from self references are used to extract query parameters for constructing a hyperlink (line 11, the other extractions were omitted because they are syntactically identical). Line 17 shows how links are extracted from an ArrayList of objects (ONE_TO_MANY multiplicity). Lines 21-27 show how a hyperlink to the current resource

Listing 7.4 Example @Path annotations from *Restbucks* (see figure 5.2)

```
1 @Path("")//address of the root resource
2 public class CoffeeShopResource extends AbstractResource { ... }
3 @Path("/orderlist")//static address
4 public class OrderListResource extends AbstractResource { ... }
5 @Path("payments/{paymentid}")//address with a static and a dynamic path fragment
6 public class PaymentResource extends AbstractResource { ... }
```

(domain object given as argument) is extracted. Listing 7.3 shows that all attributes are copied to the hyperlink object (lines 13-14) and how a hyperlink is created from a ONE_TO_ONE relationship. Additional to the OrderListHyperlink class from listing 7.2 the OrderHyperlink class has also fields derived from JaxrsModelAttributes that can also be (de-)serialized through annotated getters (lines 23-26).

7.2.2. Transforming the JaxrsResourceClass

Each *JaxrsResourceClass* results in a resource class and the corresponding interface.

- The *name* attribute will be used to generate the actual class names of the resulting classes.
- The *segments* are used to generate code concerned with addressing the resource classes. Detailed explanation follows in 7.2.2 : Resource identification.
- The *httpGetMethod* is used to generate the GET method, see (7.2.5 : HttpGetMethod).
- The *httpGetMethod* is used to generate the PUT method, see (7.2.5 : HttpPutMethod).
- The *httpGetMethod* is used to generate the DELETE method, see (7.2.5 : HttpDeleteMethod).
- The *httpGetMethod* is used to generate the POST methods, see (7.2.5 : HttpPostMethod).

Resource identification

Since REST is based on resources and their identification each resource class needs its own address. JAX-RS solves this by *@Path* annotations either at the class declaration or at methods. For simplicity, this work solely uses the annotations at the class declaration level, but hierarchical resource classes could be a point of future extension. Some examples of the @Path annotations are shown in listing 7.4. Line 1 declares a @Path annotation with an empty string. This means that the resource is the root resource. Line 2 shows an example for a static address and line 5 shows an address with a static part and a dynamic part. The @Path

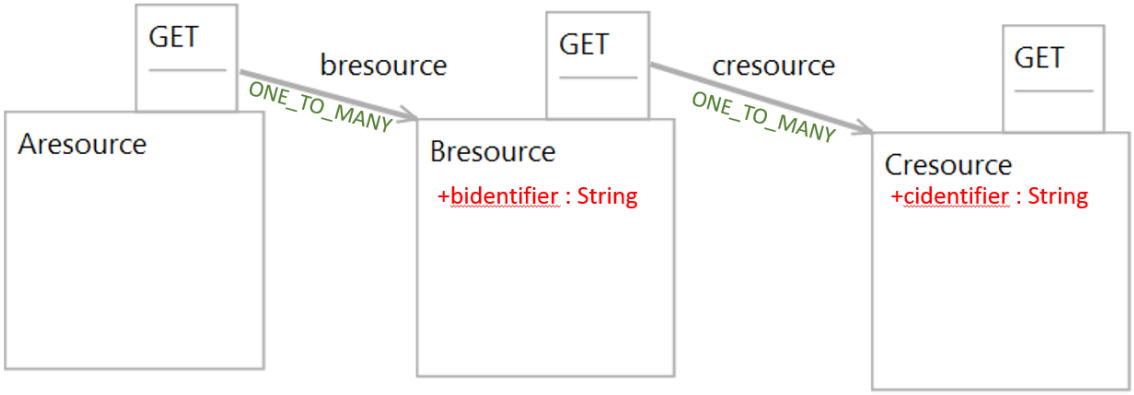


Figure 7.2.: JAX-RS PSM resulting in an address with multiple dynamic parts.

Listing 7.5 Generated domain model class with multiple identifiers.

```
1 public class CresourceDomain {
2   // this domain class uses an identifier originating fromanother domain model class
3   //the identifier of Cresource
4   private String cindentifier;
5   //the identifier that was originally from Bresource
6   private String bresource_bidentifier;
7   ...}
```

annotation does not represent the absolute URI of a resource. The @Path annotation identifies a resource relative to the address of the server the resource is hosted on. An address can have multiple dynamic parts, each of which can originate from a different Resource. To keep the convention that DLP only have to deal with one DomainObject all dynamic parts of a particular resource are added as fields to the model object classes of the resource. An example is shown in figure 7.2, where the *Navigations bresource* and *cresource* are both *ONE_TO_MANY*. The corresponding declaration of *Cresource* domain model class is shown in listing 7.5. The CresourceDomain class contains two fields. The *cindentifier* originates from the Cresource. The *bresource_bidentifier* is derived from *Bresource*.

7.2.3. DomainLogicProviderInterface

Each resource class is generated along with a *DomainLogicProviderInterface*. The purpose of the *DomainLogicProviderInterface* is to integrate domain logic into the resource class, respectively its methods. Each HTTP method of the resource class generates one method in the *DomainLogicProviderInterface*. An example from *Restbucks* is shown in listing 7.6. The listing

Listing 7.6 *DomainLogicProviderInterface* generated for *OrderResource* of *Restbucks* (5.2).

```
1  public interface OrderResourceInterface extends AbstractResourceInterface {
2    //used by the GET method
3    public OrderDomain get(String orderId);
4    //used by the PUT method
5    public OrderDomain put(String orderId, OrderDomain orderDomain);
6    //used by the DELETE method
7    public boolean delete(String orderId);
8  }
9
10 public interface OrderListResourceInterface extends AbstractResourceInterface {
11   //used by the GET method
12   public OrderListDomain get(String oFFSET, String cOUNT, String sORTBY, String
         sORTORDER);
13   //used by a POST method
14   public OrderDomain createOrder(OrderDomain orderDomain);
15   //used by a POST method
16   public Boolean archiveList(ArchiveListDomain archiveListDomain);
17 }
```

shows the domain logic provider interfaces used by the OrderResource and by the OrderListResource. The OrderResource is identified through a URI that contains a dynamic segment. This is reflected by the fact that each of the interface methods uses the *orderId* as an argument. The get and the put (lines 3 and 5) method both require an *OrderDomain* object as return value that will be used for the response body. The put method also provides the received OrderDomain object to update the resource. The delete method (line 7) only needs a boolean value to indicate if the deletion was a success (return status code 204) or not (return status code 500 - server error). The second interface is used by the OrderListResource. Since the OrderListResource can be identified statically there is no identifier provided for the methods. On answering a GET request, the user is provided with the values for all defined query parameters (line 12). The method *createOrder(...)* is used to create new instances of the OrderResource. The return value is used to create the location header and therefore the user must update the orderId of the given OrderDomain. The archiveList(...) method requires a Boolean return value from the user to indicate the success of the request. The user of the modeling tool has to implement the interfaces to provide domain logic, see 5.4.

7.2.4. Domain Logic Provider

Each generated resource class needs an object that handles incoming domain objects and provides requested domain objects. A *DomainLogicProvider* provides this services for the resources. Each resource has a private method *getDLP()* with the corresponding

DomainLogicProviderInterface as return. This method utilizes the *DomainLogicProviderRegistry* to retrieve an implementation of its interface. The DomainLogicProviderRegistry is introduced in 5.3.4: *Other modules* and 5.4: *Domain Logic Integration*.

7.2.5. Method Generation

The basic approach of JAX-RS (see 5.3.1) is, to offer Java methods as HTTP methods via annotations. Also the content-negotiation is done by the JAX-RS implementation. The *MediaTypes* for consumption and producing are defined by annotations at method declaration level: @produces and @consumes. To make use of eventual dynamic url parts or query parameters JAX-RS provides the *@PathParam*, and *@QueryParam* annotations. JAX-RS also enables access to the current uri of the resource with the *@Context* annotation. *@PathParam*, *@QueryParam* and *@Context* are used in the method header. The basic pattern for the generated methods is the following (also illustrated in figure 5.5):

- Transform any incoming object (hyperlink) to a domain object.
- Call the DLP with the domain object as argument.
- Transform the returned domain object to a hyperlink object.
- Create a response with the hyperlink object as entity.

Of course this basic template can not be realized for all methods since *GET* usually has no request entity and *DELETE* has no response entity. Every generated method has the capability of answering with a HTTP status 500 (Internal Server Error) if an exception occurred during the processing of a request. Also if a resource has dynamic URL parts each method might answer with 404 (Not Found) if the DLP returns *null*. It is assumed that a resource without dynamic url parts never gets a null value returned by the DLP. The JAX-RS implementation used, deals with a lot of other issues concerning the response status, such as recognizing if a method is allowed (405 Method Not Allowed) or content negotiation (406 Not Acceptable).

Listing 7.7 shows several generated methods as examples, that where generated from the Restbucks model (figure 5.2). The first method is a PUT method from the OrderResource. It uses both the @Consumes and @Produces annotations to specify which media types the method accepts and provides. A well formatted request body is automatically transformed to an OrderHyperlink object. The UriInfo as well as a PathParam are also injected (line 4). Line 6 shows how an domain object is created and given to the domain logic provider (DLP) in line 7. Depending on the return value of the DLP a response is constructed. If the response entity is *null* it is assumed that the resource didn't exist and a status code 404 (not found) is sent. Otherwise the responseEntity is transformed to a hyperlink object (line 10) and sent as response body with status code 200 (OK) in line 11. All methods have in common that the complete method body is enclosed by a try/catch block to capture possible exceptions. If an exception occurs, a response status 500 (internal server error) is sent with a

Listing 7.7 Several generated methods from restbucks (figure 5.2)

```
@PUT
@Consumes({"application/xml", "application/json"})
@Produces({"application/xml", "application/json"})
public Response put(@Context UriInfo ui, @PathParam("orderId") String orderId,
    OrderHyperlink orderHyperlink) {
  try {
    OrderDomain requestEntity = new OrderDomain(orderHyperlink);
    OrderDomain responseEntity = getDLP().put(orderId, requestEntity);
    if (responseEntity != null) {
    // transform to linked object
      OrderHyperlink responseEntityLinked = new OrderHyperlink(responseEntity, ui);
      return Response.ok().entity(responseEntityLinked).build();
    } else {return Response.status(404).build();}
  } catch (Exception e) {
    return Response.status(500).entity(new RESTServerException(e)).build();
}}
@POST
@Consumes({"application/archiveList+xml", "application/archiveList+json"})
public Response archiveList(@Context UriInfo ui, ArchiveListHyperlink
    archiveListHyperlink) {
  try {
    ArchiveListDomain requestEntity = new ArchiveListDomain(archiveListHyperlink);
    Boolean success = getDLP().archiveList(requestEntity);
    //noProduces indicates no responseEntity--true/false
    if (success) {return Response.ok().build();
    } else {return Response.status(500).build();}
  } catch (Exception e) {
    return Response.status(500).entity(new RESTServerException(e)).build();
}}
@DELETE
public Response delete(@Context UriInfo ui, @PathParam("receiptId") String receiptId)
    {
  try {
    Boolean success = getDLP().delete(receiptId);
    if (success != null) {
      if (success) {
        return Response.noContent().build();
      } else {return Response.status(500).build();}
    } else {return Response.status(404).build();}
  } catch (Exception e) {
    return Response.status(500).entity(new RESTServerException(e)).build();
}}
```

RESTServerException as body. The RESTServerException contains the full stacktrace delivered by the exception.
The second method is a POST method which is only annotated with a @Consumes annotation (line 17). The accepted media types differ from the specified media types in the model, which were application/xml and application/json. The reason for this is that there are two post methods in the OrderListResource from which it is generated. It is not possible to have more than one method accepting and producing the same set of media types. Instead the media types are extended to also reflect the method name in the media type. Automated (de-) serialization is still provided through the JAX-RS implementation. Since the post method has no @Produces annotation there is no response entity. Instead the DLP returns a boolean (line 21) indicating whether the request was accepted (line 23), to respond with status code 200, or not accepted (line 24) in which case a status code 500 is sent.
The third method is a DELETE method without support for a request or response entity. This method is from the ReceiptResource. Since the ReceiptResource has dynamic segments in its URL the DLP is called with the *receiptId* as argument. The delete method has three possible outcomes: The resource was not found (the return value is null) which results in a response status code 404, the request was accepted (respond with status 204 no content), or the request was not successful (respond with status 500).

HttpGetMethod

The *HttpGetMethod* method is the simplest of the generated methods. Since GET doesn't consume any request entities the DLP is only called with eventual *@PathParam* or *@QueryParam* There is only the choice whether or not the method might need to return a status 404 in case the resource has dynamic url parts. If not the method simply returns status 200 (OK) and the entity delivered by the DLP.

HttpPutMethod

The *HttpPutMethod* is basically the same as the *HttpGetMethod* with one important difference. The *HttpPutMethod* not only produces, but also consumes an entity. This means that the received entity (hyperlink model object) has to be transformed to a domain object before the DLP can be called. Other than that a *HttpPutMethod* is the same as a *HttpGetMethod*.

HttpDeleteMethod

Contrary to *HttpGetMethod* and *HttpPutMethod*, the *HttpDeleteMethod* does not respond with an entity, since the ResourceModel of the modeling tool does not provide MediaTypes for producing. If it has MediaTypes for consuming, the *HttpDeleteMethod* transforms the incoming object and gives it as an argument to the DLP. The *HttpDeleteMethod* either answers with 204

(No Content) which acknowledges deleting the resource, or with status 500 (Internal Server Error) to indicate, that the resource could not be deleted by the server. In the case of DELETE the DLP returns a simple boolean to show weather or not the resource was deleted.

HttpPostMethod

The *HttpPostMethod* is the most complex to generate, since it does not have defined semantics. It can consume and produce entities. Depending whether the method produces an entity, the DLP returns a boolean value to indicate if the request was successful, or the DLP returns an entity. A third aspect to determine the structure is the postType (CREATION, RESOURCE, or CUSTOM) of the HttpPostMethod. CREATION indicates that the method creates another resource, RESOURCE indicates that the method uses the JaxrsModelClas associated with another JaxrsResourceClass, and CUSTOM indicates that the method uses its own JaxrsModelClass. The method answers as follows:

- **No response entity:**
  - DLP returns true: successful processing, answer with 200 (OK)
  - DLP returns false: processing unsuccessful, answer with 500 (Internal Server Error)
  - DLP returns null: answer with 404 (Not Found). This can only happen if the resource has dynamic path segments, e.g. a certain processing resource was not found.
- **Response entity expected:**
  - DLP returns entity:
    - type is *CREATION*: answer with 201 (Created) and add a link header as well as the response entity.
    - type is *CUSTOM* or *RESOURCE*: answer with 200 (OK) and attach response entity.
  - DLP returns null:
    - type is *RESOURCE*: answer with 404 (Not Found)
    - type is *CUSTOM*: answer with 500 (Internal Server Error) because the server couldn't process the request.

7.2.6. Other Generated Classes and Artifacts

Besides the model and resource classes there are other classes and artifacts that have to be provided for the application to function properly. There is the DomainLogicProviderRegistry, the RESTServerException class, the RESTHyperlink class, and abstract classes. Also there are several artifacts needed for the Dropwizard framework : an Application class, a Configuration class, a pom.xml, and a configuration YAML file. Dropwizard will be introduced in the implementation chapter, see 8.1.3.

DomainLogicProviderRegistry: The *DomainLogicProviderRegistry* is a singleton [GHJV95] and the provider of DLPs for the resource classes. It searches via reflections for implementations of every *DomainLogicProviderInterface* and instantiates them. After instantiation, the *DomainLogicProviderRegistry* registers them and delivers them to the resources.

RESTServerException: *RESTServerException* is a class annotated for automatic XML/JSON (de-)serialization and is used to provide exception stacktraces to the client.

RESTHyperlink: *RESTHyperlink* is a class annotated for automatic XML/JSON (de-)serialization and is used to embed links into hyperlink model objects.

Application: The *Application* class is the starting point of the Dropwizard application. It is responsible to register all resources with the server.

Configuration: The *Configuration* is a class to provide configuration items to the *Application*.

pom.xml: Since the generated applications use Maven to import dependencies (especially to import Dropwizard), the *pom.xml* is a necessary requisite. It is an XML file that contains configuration information used by Maven to build the application [mav].

configuration YAML file: A file that is used to configure the server and loggers. It can also be used in combination with the *Configuration* class.

8. Implementation

This chapter describes the modeling tool, introduced in chapter 2.4, and how the solution developed in this thesis was integrated. The chapter introduces new technologies used for the solution, describes the architecture of the modeling tool and how the solution was integrated.

8.1. Technologies

Since the modeling tool was developed by Benjamin Schroth [Sch13], and enhanced by Jens Petersohn [Pet14] this chapter will only cover the technologies that were introduced into the modeling tool by this work. The modeling tool is based on *Eclipse Epsilon* [eps]. It uses the Eclipse Modeling Framework (EMF [emf]) to define ecore meta models and EuGENia to generate Java source code that can be used to generate graphical editors with the Graphical Modeling Framework (GMF [gmf]). The Epsilon Transformation Language (EVL) is used to validate instances of the defined meta models and the Epsilon Transformation Language (ETL) is used to realize transformations between instances of the defined meta models. The before mentioned technologies were already introduced in [Sch13], so this chapter focusses on Xtend, Maven, and Dropwizard. The use of the Dropwizard (8.1.3) framework is a mandatory requirement for the implementation of this thesis. Dropwizard provides necessary Java libraries for REST application development like Jersey (JAX-RS) and Jetty (HTTP Server).

8.1.1. Xtend

For model-to-text transformation the modeling tool used Java Emitter Templates (JET) [jeta]. JETs latest release is from 2011 and there haven't been any contributions in the last 12 months. Also there is no more tooling for JET, which means that there are no editors with syntax highlighting, validation, or other development features. So it was decided to realize the code generation with *Xtend*.
Xtend is a programming language that has its syntactical and semantical roots in the Java programming language. Instead to byte code, *Xtend* translates to Java source code, so it is completely interoperable with Java and the JVM. *Xtend* was chosen for this work because it provides features that are of great use for the model-to-text transformation.

Listing 8.1 Definition and usage of an *Extension Method*.

```
1 //declaration, the extended class is the input
2 def String getXtendGetMethodInterface(JaxrsResource resource) {
3   ...
4   return "getMethodInterface"}
5 //usage
6 var resource = new JaxrsResource()
7 //the 'get' can be ommitted
8 System.out.println(resource.xtendGetMethodInterface)
```

Extension Methods: The JAX-RS PSM meta model is defined as ecor emodel. This definition is used to generate Java classes, which are used by the generator to transform a JAX-RS PSM into Java source code. Since the model classes are regenerated with each adaption of the meta model, it is not an option to alter these generated classes. Extension Methods allow to add new functionality to existing classes without modifying the original classes. The new methods can be used with the same syntax as one would use methods of the original class. An example can be seen in listing 8.1. The JaxrsResource used as input in line 2 is an imported class from the generated Java code of the JAX-RS PSM meta model. The "traditional" way to extend this class would be to create a new class and inherit from JaxrsResource and define new methods. Xtend allows to declare so called *extension methods* in any class without the need to declare this inheritance (line 2). It is not necessary to use the *extends* keyword in the class definition. Within the class a extension method is declared, it is possible to use the extension method on objects of the original class, in this case JaxrsResource objects, in a manner as if the methods would have been declared in the original class. The syntax for the declared method, with the JaxrsResource object *resource*, would be *resource.getXtendGetMethodInterface()*. Furthermore Xtend allows to omit the a part of the method name (the "get") so the extension method can be used directly on JaxrsResource with the syntax as depicted in line 8.

Template Expressions: Generating code often requires to repeatedly use the same fragments of strings, or insert several smaller strings into a bigger template. *Template Expressions* allow to define multi line expressions, inbetween triple single quotes (''').The templates facilitate the usage of expressions and control structures within the template through the insertion of french quotes (guillemets: «»). Within the guillemets a developer can use IF conditions, LOOPs and normal Xtend code. Also the templates enable a comfortable whitespace handling to produce well formatted code. A small example of a template with an IF control structure inside is shown in listing 8.2. If the condition returns true, the complete string is "*text conditional text*" otherwise the string is "*text text*".

Listing 8.2 Example for a *Template Expression*

```
def String templateExample(){
//the template can be escaped via french quotes
return ”’text «IF condition»conditional «ENDIF»text”’
}
```

Other features of *Xtend* aim to shorten the written code, such as automatic type inference (so the developers don't need to declare variables with their type)[xte].

8.1.2. Apache Maven

Apache Maven is a project management tool that provides a uniform build system to build and manage Java based projects. The basic thought behind Maven is "Convention over Configurations". Each Maven project has a *project object model* (POM),which is stored as pom.xml and holds all information regarding the project. All Maven projects have the same directory structure, unless not specified otherwise in the pom.xml. Software dependencies are declared in the pom.xml and will be resolved by Maven when the project is built using local or public Maven-respositories. Also Maven is well integrated with Eclipse by a plugin. It was chosen because this work was required to generate Java applications using Dropwizard, which is recommended to use with Maven [mav].

8.1.3. Dropwizard

[dw] Dropwizard is a Java framework for developing RESTful web services. The main idea of Dropwizard is to provide Java libraries concerning the development of RESTful web services, so the developer can focus on developing the application instead of piecing together all needed libraries before even one line of code is written. Dropwizard also deals with the integration of those libraries and provides an easy way for developers to create runnable *Fat JARs* complete with HTTP Server and the developed application deployed. The most important libraries for this project delivered by Dropwizard are Jetty, Jersey, and Jackson.

Jetty: The Jetty library is used by the Dropwizard application to embed an HTTP server to the project. Dropwizard projects start the HTTP server with their main() method.[jetb]

Jersey: Jersey is the JAX-RS reference implementation that provides a lot of functionality regarding RESTful web services.[jer]

Jackson: Jackson is a JSON library for Java used to provide simple automatic (de-)serialization via annotations (Jackson annotations or JAXB annotations) to XML and JSON.[jac]

8.2. Architecture

This chapter shows the architecture of the existing modeling tool and explains the individual modules the modeling tool is comprised of. The modeling tool is realized as an eclipse based application, so it can only be used within eclipse. The tool is made up of four distinct modules:

Meta Models: The *Meta Models* module is a collection of all meta models used by the modeling tool and is realized through several projects. All meta models are defined as Ecore models to enable the generation of usable Java code for the *Generator*, and the generation of graphical editors for users of the modeling tool.

ModelTransformations: The *ModelTransformations* is a set of ETL scripts that handle the transformations between the models. Each transformation is offered to the user via the context menu of eclipse.

ModelValidator: The *ModelValidator* is a set of EVL scripts to validate instances of the the meta models provided by the *Models* module.

Generator: The *Generator* is the component that handles the transformation of JAX-RS PSMs to Dropwizard projects. It is realized using the programming language *Xtend* that integrates well with EMF and provides useful features for code generation (8.1.1).

An overview over the architecture is shown in figure 8.1.

8.3. Integration

The task of this work was to design and realize the generation of application code. Also the solution was required to be integrated into the modeling tool. As described in [Sch13] and [Pet14] the modeling tool already provided the capabilities to realize the generation of application code, but in a rather simple fashion. The existing approach also didn't have the capability to cover all possible resource models. The new JAX-RS PSM meta model was defined as an Ecore model and added to a project that already provided a platform specific model, which is now deprecated. To transform the *ResourceModel* and the *DeploymentModel* into the newly defined JAX-RS PSM, a new ETL script was added to the *ModelTransformations* component.
An important aspect of this work was the development of a new *Generator*. This was realized using the programming language *Xtend*. The modeling tool uses OSGi ([osg]) to integrate all necessary projects. So the *Generator* was integrated by declaring dependencies in the projects manifest file. The generation of application code can be accessed via eclipses context menu by right-clicking on a JAX-RS PSM file.
Figure 8.3 shows the situation after modeling an application as a resource model and corresponding deployment model. The context menu offers the generation of a JAX-RS PSM. Necessary information is prompted from the user (see figure 8.2 (a)) and a JAX-RS PSM is

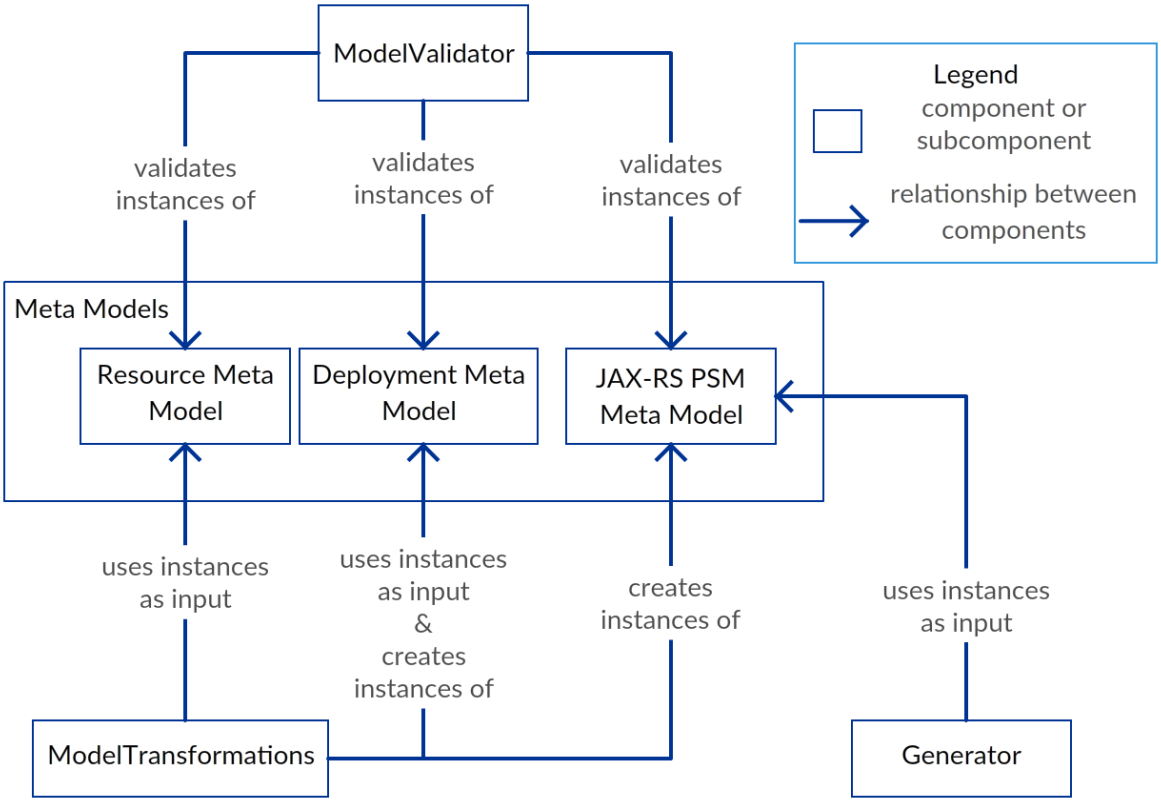


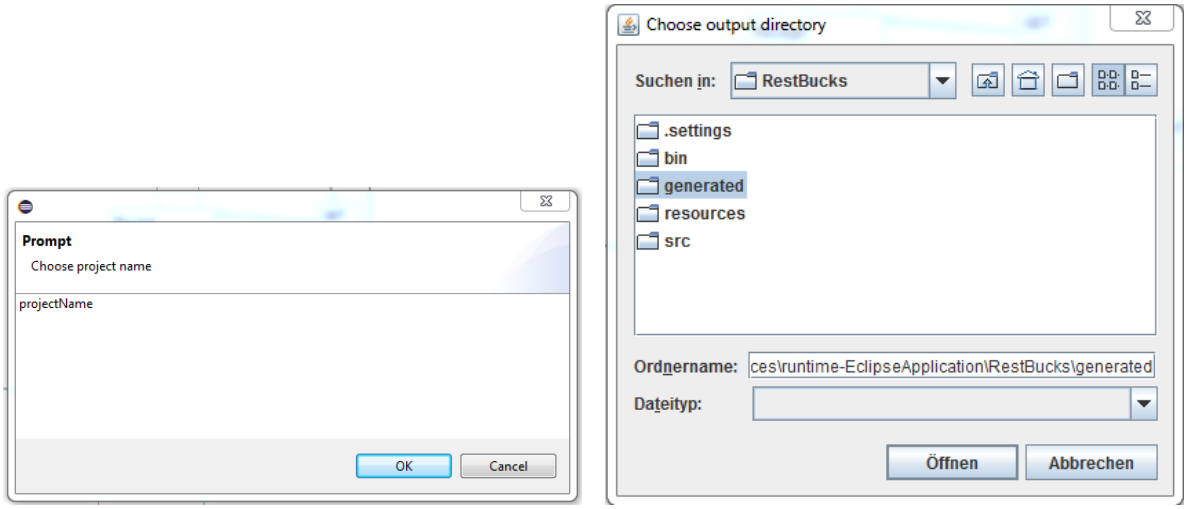
Figure 8.1.: Architecture of the modeling tool.

generated. The JAX-RS PSM now offers the generation of a JAX-RS Maven project through the context menu on the PSM. When started, the user is asked to provide the output directory. In this case the original eclipse project (RestBucks in figure 8.3) was created as Java project and the folder *generated* is a source code folder intended to serve as output directory, see figure 8.2 (b). The result of the code generation can be seen in figure 8.4.

8.4. Limitations

This chapter describes the technical limitations of the current solution. The limitations are as follows:

MediaTypes: It is currently only possible to generate valid Java code for the MediaTypes *application/json* and *application/xml*. The resource meta model provides three additional media types: text/plain, text/html and application/octet-stream. This limitation could



(a) User is prompted to provide the project name.

(b) User is prompted to choose the output directory.

Figure 8.2.: User dialogs.

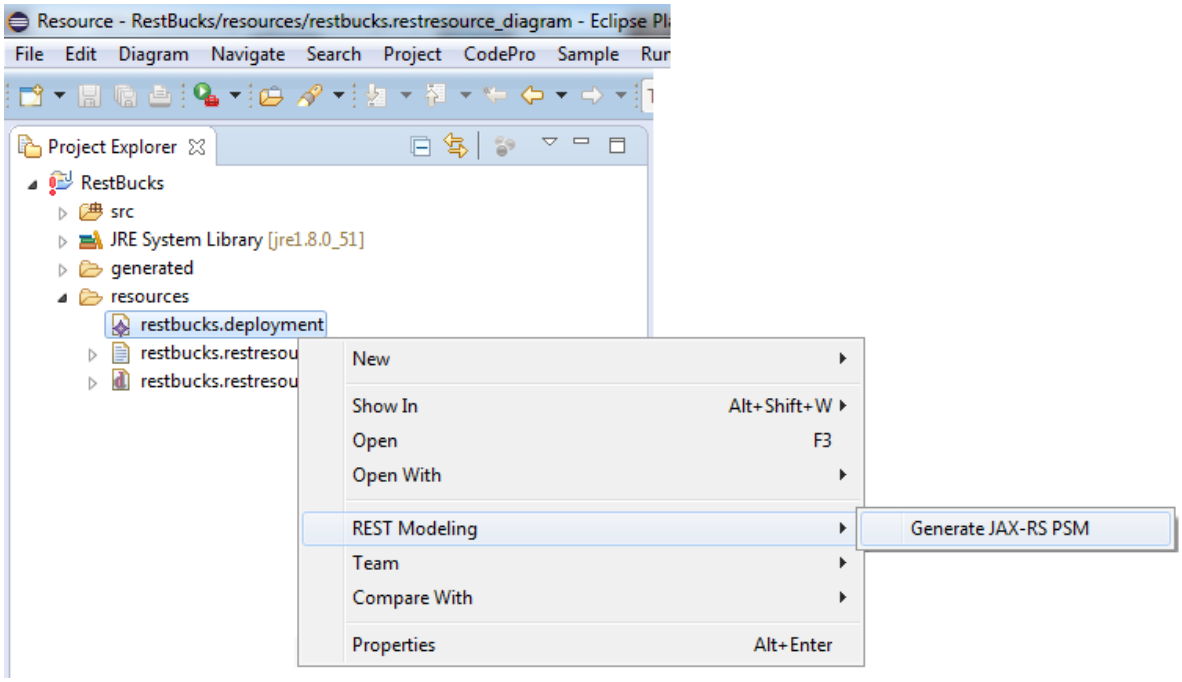


Figure 8.3.: Context menu on deployment model (several items of the context menu were omitted).

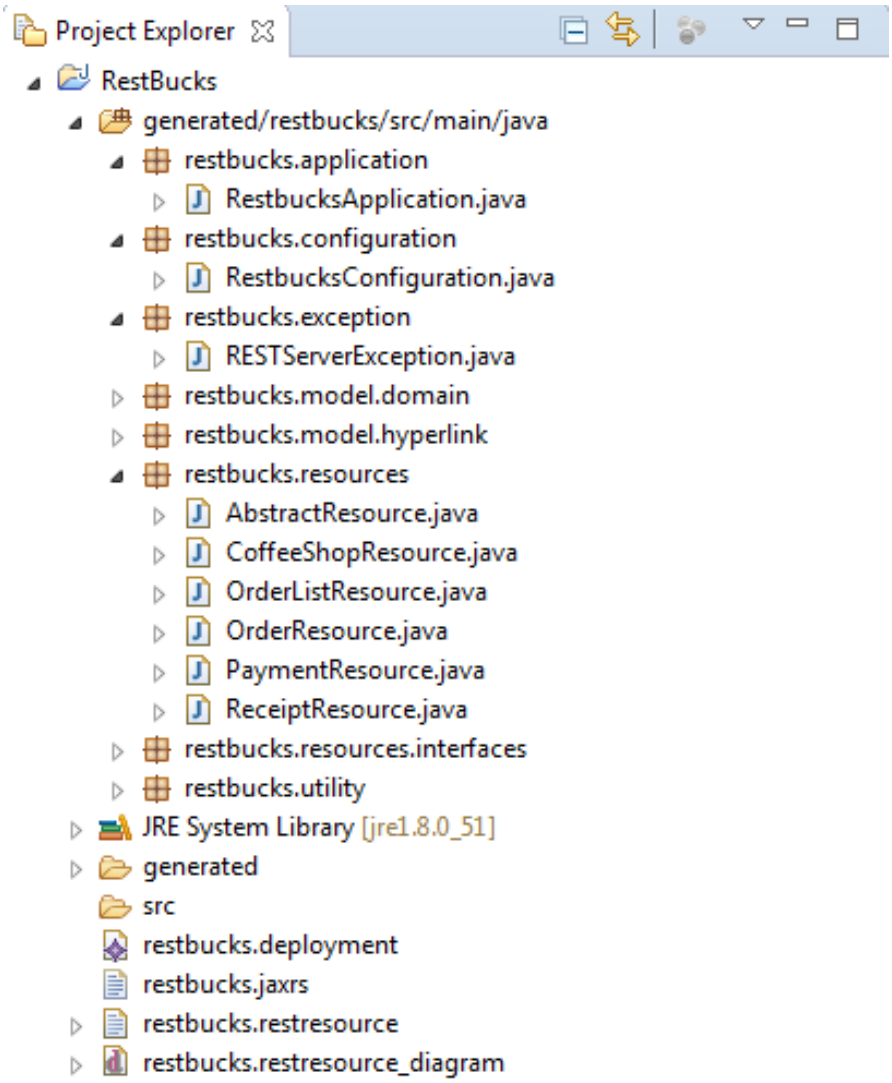


Figure 8.4.: The result of the code generation.

be addressed by generating MessageBodyWriter and MessageBodyReader interfaces for these media types to be implemented by the user.

Maven artifacts: The *pom.xml* is currently generated every time code is generated. Any changes by the user will be lost. This limitation could be addressed by introducing an additional model that allows the user to make additions to the pom.xml that will be reflected in the generated file.

Dropwizard artifacts: The *.yml file* is currently generated every time code is generated. Any changes by the user will be lost. This issue can be addressed through an addition to the JAX-RS PSM or through an additional model that focusses on the Dropwizard configuration file.

HTTP DELETE: It is currently not possible to provide a response body for HTTP DELETE method. To address this limitation the resource meta model as well as the JAX-RS PSM meta model have to be adapted.

URIs: It is currently only possible to associate one resource with one URI template. This constraint was introduced to use the @Path element at class level (requires exactly one URI template per resource class) in order to reduce the complexity of code generation for this thesis. If a REST API provides access to a data structure with directories and files there is a need for cyclic behaviour. It is possible to realize this behaviour through @Path annotations at method level in JAX-RS resource classes and can be a point of future extension to the prototype.

9. Summary

This work addresses the generation of source code for applications compliant to the constraints of REST. The existing modeling tool described in 2.4 provides a formal model to fully describe a REST API and the corresponding addressing of resources. The goal of the thesis was the development of a platform specific model to later generate REST compliant code using the Dropwizard (8.1.3) framework.
An important aspect was the development of a reference application (chapter 5) to identify programming patterns that could serve as template for the resulting Java classes. The developed application also served as a means to derive a platform specific model covering all aspects to fully describe a Dropwizard application. Another important task was the development of a concept to integrate manually written and automatically generated code. The developed concept uses reflections to search the classpath of an application to discover interface implementations at starting time of the application that can be used by the generated code.
The platform specific model (chapter 6) and the generation of code (chapter 7.2) can be used independently from the existing modeling tool. So the most important part of the integration into the existing tool was the development of a model transformation that uses the resource and the deployment model as input and transforms them into the platform specific model. This transformation is described in 7.1. The modeling tool is implemented using the Eclipse Modeling Framework, Eclipse Epsilon and the Graphical Modeling Framework. To integrate this work into the framework the developed model transformation was realized using the Eclipse Transformation Language so that the transformation could be added to the already existing *Model Transformations* component of the modeling tool. The *Generator* was written using *Xtend* (see 8.1.1) and added to the modeling tool as a plugin.

Conclusion

An important aspect of this work is the generation of source code for applications compliant to the REST constraints. Every resource modeled with the modeling tool will ultimately result in three Java classes. A resource class, a hyperlink class, and a domain class. A hyperlink class is a means to provide at least two different representation formats for any resource. The resource identification is derived from the deployment model and realized through *@Path* annotations in Java. The usage of Jersey ensures that there are no ambiguous address

templates for resources in a running application. REST also calls for standard methods that can now be easily realized through modeling. The resource model supports all HTTP methods relevant to realize a CRUD-like (Create Retrieve Update Delete) interface for any resource. The resource model also enables the modeling of relationships between resources via navigations. This navigations are used by the solution to automatically embed appropriate links into resource representations so HATEOAS can be realized by developers with ease. REST demands stateless communication between client and server meaning that the server side should not keep or use any session state for processing requests. This is enforced by not giving control of the communication protocol to the developer of the application. The manually implemented code is integrated through the manual implementation of generated interfaces. None of these interface methods have any input other than identifiers or domain objects. So the developer can not rely on context information from the server. This measures are all designed to ease the development of applications compliant to the REST constraints and to enforce several principles defined by REST. The developed solution provides an easy way to develop applications compliant to the REST constraints without the need to manually implement source code related to the technical aspects of HTTP based REST APIs and allows developers to focus on the development of domain specific logic.

Prospect

There are several possibilities to improve the modeling tool described in this work. This chapter provides several suggestions how the tool can be enhanced.

At the moment there is one JAX-RS PSM meta model that is sufficient to generate all artifacts necessary for a Dropwizard application. Dropwizard uses the JAX-RS implementation *Jersey*. There are many other ways to use classes annotated with JAX-RS annotations. The generated classes can also be used for REST applications realized as WAR files (Web Application Archive) or other formats. Currently the JAX-RS PSM contains information needed for Dropwizard and Maven. This information could be moved to other models, so that the JAX-RS PSM only contains information relevant to JAX-RS, while information on the target application could be held in separate models. Possible new models could be a Dropwizard model, a WAR model, and a maven model. Figure 9.1 shows the current model and transformation structure, as well as the suggested.

Dropwizard model: A Dropwizard application needs a configuration file to start. The generator currently creates a minimal version of this configuration file. The configuration file of Dropwizard allows the configuration of the server, logging, application metrics, and database access. This can currently only be done manually with the risk, that the configuration file is overwritten during the next generation cycle.

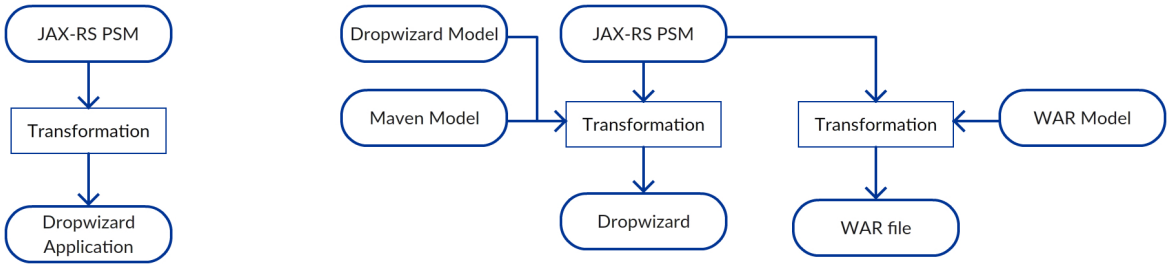


Figure 9.1.: Current (left) and suggested (right) model and transformation structure

Dropwizard also provides capabilities for *HealthChecks*[1] which can currently not be created automatically.

Maven model: When developing an application it is often the case that additional libraries are needed. Maven allows to declare dependencies to import libraries to the project. This feature is currently used by generated applications, but it is not really available to the application developer. As with the Dropwizard configuration file, the maven pom is also generated in a minimal fashion. So a new Maven model could be used to provide additional elements to be added to the generated pom.

Other suggestions to enhance the modeling tool are the following:

HATEOAS: Currently the *RESTHyperlink* class is used to create embedded links (see 7.2.6). This could be enhanced so that the application developer can provide custom formats.

DomainClasses: Domain classes are currently modeled through the use of navigations and attributes, which can only have three different attribute types. This mechanism could be enhanced to allow a more sophisticated definition of domain classes through the use of XML Schema.

MediaTypes: There are five predefined media types that can be modeled. There is currently no possibility to provide a custom media type to a generated application.

URIs: Currently there is the requirement that there is exactly one URI template for each resource. In some cases it is necessary that there can be multiple URI templates for resource classes.

Deprecation: Sometimes resources might become deprecated. This fact is currently ignored by the generator. So the deletion of resources in the model doesn't result in the deletion

[1] "Health checks give you a way of adding small tests to your application to allow you to verify that your application is functioning correctly in production"[dw]

of artifacts generated from the deleted resource. One possible way, is to *mark* those resources as deprecated and delete all marked resources during the generation process.

A. Appendix

A.1. JAX-RS PSM Metamodel

JaxrsModel: The root element for all JAX-RS PSMs. It holds all information necessary to generate a rest application (except for domain logic).

Attributes

String projectName: The name of the project, non optional.
String projectVersion: The version number, non optional.
String groupId: The groupId for the maven *pom.xml*, non optional.

Associations

JaxrsResourceClass[*] resources: A collection of all *JaxrsResourceClasses* in the model.
JaxrsModelClass[*] modelClasses: A collection of all *JaxrsModelClasses* in the model.

JaxrsResourceClass: Represents the resource classes that will later be generated.

Attributes

String name: The name of the resource class.

Associations

JaxrsPathSegment[*] segments: A ordered collection of *JaxrsPathSegments* that describes the complete url path for this resource.
JaxrsModelClass modelClass: The primary *JaxrsModelClass* that represents the entity associated with the resource.
HttpGetMethod httpGetMethod: The GET method of the resource. Can be empty.
HttpPutMethod httpPutMethod: The PUT method of the resource. Can be empty.
HttpDeleteMethod httpDeleteMethod: The DELETE method of the resource. Can be empty.
HttpPostMethod[*] httpPostMethods: Collection of POST methods of the resource. Can be empty.

JaxrsPathSegment: An abstract class to group static and dynamic JaxrsPathSegments.

JaxrsStaticPathSegment: The static version of the *JaxrsPathSegment*. It represents a static url path segment.

Attributes

String segment: A url path segment.

JaxrsDynamicPathSegment: The dynamic version of the JaxrsPathSegment. It points to a *JaxrsRentityAttribute*.

Associations

JaxrsModelClassAttribute attribute: The *JaxrsRentityAttribute* that will be used as dynamic url path segment.

HttpMethod: An abstract class to group all HttpMethods.

HttpGetMethod: The representation of a GET method.

Attributes

MediaType[1..5] produces: A collection of *MediaTypes* the method can produce.

Associations

JaxrsQueryParameter[*] queryParameter: A collection of *JaxrsQueryParameter* the method uses.

JaxrsResourceClass parent: The resource that offers the method.

HttpPutMethod: The representation of a PUT method.

Attributes

MediaType[1..5] produces: A collection of *MediaTypes* the method can produce.

MediaType[1..5] consumes: A collection of *MediaTypes* the method can consume.

Associations

JaxrsQueryParameter[*] queryParameter: A collection of *JaxrsQueryParameter* the method uses.

JaxrsResourceClass parent: The resource that offers the method.

HttpDeleteMethod: The representation of a DELETE method.

Attributes

MediaType[0..5] consumes: A collection of *MediaTypes* the method can consume. Can be empty.

Associations

JaxrsQueryParameter[*] queryParameter: A collection of *JaxrsQueryParameter* the method uses.

JaxrsResourceClass parent: The resource that offers the method.

HttpPostMethod: The representation of a POST method

Attributes

String methodName: The name of the method.

PostType postType: The type of the method.
MediaType[0..5] produces: A collection of *MediaTypes* the method can produce. Can be empty.
MediaType[0..5] consumes: A collection of *MediaTypes* the method can consume. Can be empty.

Associations

JaxrsModelClass modelClass: The *JaxrsModelClass* that is used by this method. This can either be the *JaxrsModelClass* of the parent *JaxrsResourceClass* or a separate *JaxrsModelClass* used only by this method.
JaxrsResourceClass parent: The resource that offers the method.

JaxrsModelClass: This class provides all information to generate *DomainClasses* and *HyperlinkClasses* (see chapter 5.2).

Attributes

String name: The basic name of the resulting classes. Will be extended by "Hyperlink" or "Domain".

Associations

JaxrsModelClassAttribute[*] attributes: A collection of *JaxrsModelClassAttributes* that define the data structure of the model classes. Can be empty.
JaxrsModelClassReference[*] references: A collection of *JaxrsModelClassReferences*. To show the associations to other *JaxrsModelClasses*.
JaxrsQueryParameter[*] queryParameter: A collection of *JaxrsQueryParameter*. Contains all *JaxrsQueryParameter* of all methods of the parent resource to define them as static fields in the class declaration.
JaxrsResourceClass parent: The main resource where the *JaxrsModelClass* is used.

JaxrsModelClassAttribute: Used to model the data structure or *JaxrsModelClasses*.

Attributes

String name: The name of the attribute.
AttributeDataType type: The type of the attribute.

Associations

JaxrsModelClass parentModelClass: The *JaxrsModelClass* associated with the attribute.

JaxrsModelClassReference: Used to model the associations between the resulting *DomainClasses*.

Attributes

String name: The name of the reference.

String referencedResourceName: The *name* of the *JaxrsResourceClass* the reference is associated with. Mainly used during transformation.

Multiplicity multiplicity: An enumeration to differentiate between *ONE_TO_ONE* and *ONE_TO_MANY* relationships.

Associations

JaxrsModelClass targetClass: The *JaxrsModelClass* the *parentClass* is associated with.

JaxrsModelClass parentClass: The *JaxrsModelClass* that is the owner of the reference.

JaxrsQueryParameter: A simple class to model *QueryParameters*.

Attributes

String name: The name of the *JaxrsQueryParameter*.

PostType: An enumeration to differentiate between different types of POST methods.

CREATION: Indicates that the POST method creates an instance of of a particular resource (its *JaxrsModelClass*).

RESOURCE: Indicates that the POST method uses the *JaxrsModelClass* of a resource method.

CUSTOM: Indicates that the POST method uses its own *JaxrsModelClass*, that is not a primary *JaxrsModelClass* of a resource.

A.2. Model-To-Model Transformation Rules

A.2.1. **JaxrsModel Transformation**

input: ResourceDiagram

- projectName: user input, non optional.
- projectVersion: user input, non optional.
- groupId: user input.
- resources: transformed from *ResourceDiagram resources* (A.2.2).
- modelClasses: transforment from *ResourceDiagram resources* (A.2.9).

A.2.2. **JaxrsResourceClass Transformation**

input: Resource, DeploymentModel
initiatedBy: **JaxrsModel Transformation**

- name: derived from *Resource name*.
- segments: derived from the *DeploymentModel*. The input *Resource* is searched within the DeploymentModel and then traced backwards to the root.
- modelClass: the *JaxrsModelClass* is simultaneously created from the *Resource* (A.2.9).
- httpGetMethod: transformed from *Resource GET* (A.2.5)
- httpPutMethod: transformed from *Resource PUT* (A.2.6)
- httpDeleteMethod: transformed from *Resource DELETE* (A.2.7)
- httpPostMethods: transformed from *Resource POST*s *Interactions*. (A.2.8)

A.2.3. **JaxrsStaticPathSegment Transformation**

input: StaticURLFragment
initiatedBy: **JaxrsResourceClass Transformation**

- segment: derived from *StaticURLFragment fragment*.

A.2.4. **JaxrsDynamicPathSegment Transformation**

input: DynamicURLFragment
initiatedBy: **JaxrsResourceClass Transformation**

- attribute: transformed from *DynamicURLFragment attribute* (A.2.11).

A.2.5. **HttpGetMethod Transformation**

input: GetMethod
initiatedBy: **JaxrsResourceClass Transformation**

- produces: derived from *GetMethod produces*.
- queryParameter: derived from *GetMethod params*.
- parent: the initiating *JaxrsResourceClass*

A.2.6. **HttpPutMethod Transformation**

input: PutMethod
initiatedBy: **JaxrsResourceClass Transformation**

- produces: derived from *PutMethod produces*.
- consumes: derived from *PutMethod consumes*.
- queryParameter: derived from *PutMethod params*.
- parent: the initiating *JaxrsResourceClass*

A.2.7. **HttpDeleteMethod Transformation**

input: DeleteMethod
initiatedBy: **JaxrsResourceClass Transformation**

- consumes: derived from *DeleteMethod consumes*.
- queryParameter: derived from *DeleteMethod params*.
- parent: the initiating *JaxrsResourceClass*

A.2.8. **HttpPostMethod Transformation**

input: Interaction
initiatedBy: **JaxrsResourceClass Transformation**

- methodName: derived from *Interaction name*.
- postType: calculated from the *Interaction*. CREATION if the *Interaction* is a *LinkSource* of *ResourceCreation*, RESOURCE if the *Interaction* has an *inputResource*, and CUSTOM if the *Interaction* has its own *entityStructure*.
- produces: derived from *Interaction produces*.
- consumes: derived from *Interaction consumes*.
- modelClass: either the modelClass of another *JaxrsResourceClass* (postType is CREATION or RESOURCE), or a new modelClass is created (postType is CUSTOM, A.2.10).
- parent: the initiating *JaxrsResourceClass*

A.2.9. **JaxrsModelClass Transformation from Resource**

input:Resource
initiatedBy: **JaxrsResourceClass Transformation**

- name: derived from *Resource name*.
- attributes: transformed from *Resource entityStructure* (A.2.11).
- references: transformed from the *Navigations* of the *Resource GET, PUT, DELETE* (A.2.12).
- queryParameter: transformed from the *queryParameter* of *Resource GET, PUT, DELETE* (A.2.13).
- parent: the initiating JaxrsResourceClass.

A.2.10. **JaxrsModelClass Transformation from Interaction**

input: Interaction
initiatedBy: **HttpPostMethod Transformation**

- name: derived from *Interaction name*.
- attributes: transformed from *Interaction entityStructure* (A.2.11).
- references: transformend from the *Navigations* of the *Interaction* (A.2.12).
- queryParameter: The *Resource POST* does not have any QueryParameter. So this will stay empty.
- parent: The parent *Resource* of the associated *PostMethod*.

A.2.11. **JaxrsModelClassAttribute Transformation**

input: EntityAttribute
initiatedBy: **JaxrsModelClass Transformation from Resource** or **JaxrsModelClass Transformation from Interaction**

- name: derived from *EntityAttribute name*.
- type: derived from *EntityAttribute type*.
- parentModelClass: the initiating *JaxrsModelClass*.

A.2.12. **JaxrsModelClassReference Transformation**

input: Navigation
initiatedBy: **JaxrsModelClass Transformation from Resource** or **JaxrsModelClass Transformation from Interaction**

- name: derived from *Navigation type*.
- referencedResourceName: derived from *Navigation source parent*.
- targetClass: derived from *Navigation target*.
- parentClass: the initiating *JaxrsModelClass*.
- multiplicity: derived from *Navigation multiplicity*.

A.2.13. **JaxrsQueryParameter Transformation**

input: Parameter
initiatedBy: **HttpGetMethod Transformation** or **HttpPutMethod Transformation** or **HttpDeleteMethod Transformation**

- name: derived from *Parameter name*.

Bibliography

[dw] Dropwizard. URL http://www.dropwizard.io/. (Cited on pages 79 and 87)

[emf] Eclipse Modeling Framework. URL https://eclipse.org/modeling/emf/. (Cited on page 77)

[eps] Eclipse Epsilon. URL http://www.eclipse.org/epsilon/. (Cited on page 77)

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, 2000. (Cited on pages 9 and 13)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Cited on page 76)

[gmf] Graphical Modeling Framework. URL http://www.eclipse.org/modeling/gmp/. (Cited on page 77)

[HFK+14] F. Haupt, M. Fischer, D. Karastoyanova, F. Leymann, K. Vukojevic-Haupt. Service Composition for REST. In *Proceedings of the 18th IEEE International EDOC Conference (EDOC 2014)*. IEEE, 2014. (Cited on page 21)

[HKLS14] F. Haupt, D. Karastoyanova, F. Leymann, B. Schroth. A Model-Driven Approach for REST Compliant Services. In *Proceedings of the IEEE International Conference on Web Services (ICWS 2014)*, pp. 129 – 136. IEEE, 2014. doi:10.1109/ICWS.2014.30. (Cited on pages 7, 30 and 31)

[HLP15] F. Haupt, F. Leymann, C. Pautasso. A conversation based approach for modeling REST APIs. In *12th Working IEEE / IFIP Conference on Software Architecture - WICSA 2015*. IEEE Computer Society, 2015. (Cited on pages 7, 30, 31 and 32)

[htt99] Hypertext Transfer Protocol – HTTP/1.1, 1999. URL https://tools.ietf.org/html/rfc2616. (Cited on pages 15, 16 and 22)

[jac] Jackson JSON Processor. URL http://wiki.fasterxml.com/JacksonHome. (Cited on page 79)

[jaxa] Java API for RESTful Services (JAX-RS). URL https://jax-rs-spec.java.net/. (Cited on pages 52 and 65)

[jaxb] Java Architecture for XML Binding (JAXB). URL https://jcp.org/en/jsr/detail?id=222. (Cited on page 65)

[jer] Jersey. URL https://jersey.java.net/. (Cited on pages 52 and 79)

[jeta] Java Emitter Templates. URL https://eclipse.org/modeling/m2t/?project=jet. (Cited on page 77)

[jetb] Jetty. URL http://www.eclipse.org/jetty/. (Cited on page 79)

[LSS09] M. Laitkorpi, P. Selonen, T. Systa. Towards a Model-Driven Process for Designing ReSTful Web Services. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pp. 173–180. 2009. (Cited on pages 7, 41, 43 and 46)

[mav] Apache Maven. URL https://maven.apache.org/. (Cited on pages 76 and 79)

[mim96] Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types, 1996. URL https://tools.ietf.org/html/rfc2046. (Cited on page 16)

[Mus12] J. Musser. Open APIs, what's hot, what's not, 2012. (Cited on pages 7 and 10)

[omg] Object Management Group. URL http://www.omg.org. (Cited on page 27)

[ope] URL http://www.opengroup.org/. (Cited on page 19)

[osg] OSGi. URL http://www.osgi.org. (Cited on page 80)

[Pau09] C. Pautasso. Some REST Design Patterns (and Anti-Patterns), 2009. (Cited on page 9)

[Pet14] J. Petersohn. A multilayered model for REST applications, 2014. (Cited on pages 39, 77 and 80)

[PZL08] C. Pautasso, O. Zimmermann, F. Leymann. Restful Web Services vs. "Big"' Web Services: Making the Right Architectural Decision. In *Proceedings of the 17th International Conference on World Wide Web*, WWW '08. ACM, 2008. (Cited on pages 21 and 22)

[rmm] Richardson Maturity Model. URL http://www.crummy.com/writing/speaking/2008-QCon/act3.html. (Cited on page 22)

[Sch11] S. Schreier. Modeling RESTful Applications. In *Proceedings of the Second International Workshop on RESTful Design*, WS-REST '11. ACM, 2011. (Cited on pages 7, 41 and 42)

[Sch13] B. Schroth. Entwurf und Realisierung von REST - Anwendungen nach Prinzipien der modellgetriebenen Softwareentwicklung, 2013. (Cited on pages 39, 77 and 80)

[SVEH07] T. Stahl, M. Völter, S. Efftinge, A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt, 2 edition, 2007. (Cited on pages 26 and 27)

[Til08] S. Tilkov, 2008. URL http://www.infoq.com/articles/rest-anti-patterns. (Cited on page 9)

[uri05] Uniform Resource Identifier (URI): Generic Syntax, 2005. URL https://tools.ietf.org/html/rfc3986. (Cited on page 17)

[url94] Uniform Resource Locators (URL), 1994. URL https://tools.ietf.org/html/rfc1738. (Cited on page 17)

[Vit10] T. Vitvar, 2010. URL http://www.programmableweb.com/news/api-anti-patterns-how-to-avoid-common-rest-mistakes/2010/08/13. (Cited on page 9)

[VP09] F. Valverde, O. Pastor. Dealing with REST Services in Model-driven Web Engineering Methods. *V Jornadas Científico-Técnicas en Servicios Web y SOA, JSWEB*, 2009. (Cited on pages 7, 41, 46 and 47)

[w3c] World Wide Web Consortium. URL http://www.w3.org. (Cited on page 19)

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005. (Cited on pages 7 and 20)

[WPR10] J. Webber, S. Parastatidis, I. Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 1st edition, 2010. (Cited on pages 7, 39, 49 and 50)

[wsd] Web Service Description Language. URL http://www.w3.org/TR/wsdl. (Cited on pages 8, 20 and 21)

[xte] Xtend. URL http://www.eclipse.org/xtend/. (Cited on page 79)

All links were last followed on Oktober 16, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature