Model-driven Round-trip Engineering of REST APIs

Hamza Ed-douibi

A thesis submitted for the degree of Doctor of Philosophy in Network & Information Technologies

Department of Computer Science, Multimedia and Telecommunications

Open University of Catalonia

Supervised by
Jordi Cabot and Javier Luis Cánovas Izquierdo

May 2019

SUPERVISORS

- Dr. Jordi Cabot – *Internet Interdisciplinary Institute (IN3) - ICREA, Barcelona, Spain*
- Dr. Javier Luis Cánovas Izquierdo – *Internet Interdisciplinary Institute (IN3) - Universitat Oberta de Catalunya (UOC), Barcelona, Spain*

THESIS DEFENSE COMMITTEE MEMBERS

- Dr. Francis Bordeleau – *École de technologie supérieure, Université du Québec, Montreal, Canada*
- Dr. Marco Brambilla – *Politecnico di Milano, Milan, Italy*
- Dr. Robert Clarisó – *Universitat Oberta de Catalunya (UOC), Barcelona, Spain*

HAMZA ED-DOUIBI, *Model-driven Round-trip Engineering of REST APIs*. PhD Thesis, Network and Information Technologies Doctoral Programme, Universitat Oberta de Catalunya (UOC), 2019.

'Something else an academic education will do for you. If you go along with it any considerable distance, it'll begin to give you an idea what size mind you have. [...] After a while, you'll have an idea what kind of thoughts your particular size of mind should be wearing. [...] You'll begin to know your true measurements and dress your mind accordingly.'

— J. D. Salinger, The Catcher in the Rye (1951)

Acknowledgments

This PhD would not have been possible without the help and support of many people. The following lines are dedicated to those who contributed to this achievement.

First of all, I would like to express my sincere gratitude to my supervisors, Jordi Cabot and Javier Luis Cánovas Izquierdo. Jordi, who fought to get this PhD started, always believed in me and supported me. His direction and good advice helped me achieve this thesis. Javi, who is wearing two hats: a supervisor and a friend, has always been there to mentor me, help me technically, and support me emotionally. Jordi, Javi, I will always be indebted to you.

In addition, I would like to thank the other members of SOM research team for providing such a loving working environment. In SOM research, I have not only found colleagues but also good friends. I owe countless thanks to Abel Gómez who gave me many advices and helped me fix several technical issues. My sincere thanks to Gwendal Daniel for joining me during the break times and providing me technical support. I would like to thank Lola Burgueño who offered me help and encouraged me during the writing of this thesis.

There are also a few people in particular whom I would like to thank. Aljoscha Gruler, who I met at UOC and became a good friend and companion by sharing similar experiences as PhD student. Amine Benelallam, who is always there to listen to my problems and to support me. Ayoub Chakib, my good friend from Morocco, who always encourages me and welcomes me each time I go back home.

Finally, I am thankful to my family for their love and continued support. In particular, I would like to thank my parents, Latifa and Ahmed, for their selfless sacrifices, and supporting me to travel abroad to pursue my own path in life. I'm thankful to my brother and sisters, Anas, Hanae and Yousra, for always making smile. Last but not least, I would like to thank my better half, my wife, Míriam, who has always been by my side to celebrate with me after a success, comfort me after a failure, and keep me moving forward.

Abstract

Web APIs have increasingly becoming a key asset for businesses, thus boosting their implementation and integration in companies' daily activities. Such importance is reflected by the growing number of available public Web APIs, listed in a number of catalogs (e.g., APIS.GURU with more than 800 APIs or PROGRAMMABLEWEB with more 19000 APIs). In practice, most of these Web APIs are "REST-like", meaning that they adhere partially to the Representational State Transfer (REST) architectural style. REST is a technical description which outlines the principles, properties, and constraints to build Internet-scale distributed hypermedia systems. Indeed, REST is a design paradigm and does not propose any standard. Thus, both developing and consuming REST APIs are challenging and time-consuming tasks for API providers and clients, respectively. In fact, API providers should have a sound API strategy fostering ease of use while respecting REST constraints. On the other hand, writing applications to consume these APIs typically requires sending HTTP requests and using JSON or XML to represent data. This task is fully manual and developers are having a hard time integrating Web APIs to their applications due to the lack of machine-readable definitions. In fact, Web APIs adopted a human-oriented approach based on informal textual descriptions explaining what they propose. Recently, and aiming at standardizing the way to describe REST APIs, a consortium of major actors in the API market has launched the OpenAPI Initiative (OAI). This initiative has the objective of creating a vendor-neutral, portable, and open specification for describing REST APIs. OAI has succeeded in attracting major companies and the OpenAPI specifi-

cation has become the choice of reference to describe REST APIs. On the other hand, Open Data Protocol (OData), is an emerging specification for Web APIs which is specially useful to expose and query data sources as REST APIs. The current version of OData (version 4.0) has been approved as an OASIS standard.

The objective of this thesis is to facilitate the design, implementation, composition and consumption of REST APIs, targeting specially the OpenAPI specification and OData protocol, by relying on Model-Driven Engineering (MDE). MDE is a methodology that promotes the use of models and transformations to raise the level of abstraction and automation in software development, respectively. This thesis proposes the following contributions: (i) EMF-REST, an approach to generate REST APIs for models, thus promoting model management in distributed environments; (ii) APIDISCOVERER, an example-based approach to automatically infer OpenAPI specifications for REST APIs, thus helping developers increase the exposure of their APIs without fully writing API specifications; (iii) APITESTER, an approach to generate test cases for REST APIs relying on their OpenAPI specifications to assess that the behavior of an API conforms to its specification; (iv) APIGENERATOR, a model-driven approach to automate the generation of ready-to-deploy OData REST APIs from conceptual models; and (v) APICOMPOSER, a lightweight model-based approach to automatically compose REST APIs based on their data models. These contributions constitute an ecosystem which advances the state of the art of automated software engineering for REST APIs development and consumption. We believe such contributions to be of a great value to Web APIs developers who want to bring more agility to their development tasks.

Resumen

Las API Web se han convertido en una pieza fundamental para un gran número de compañías, que han promovido su implementación e integración en las actividades cotidianas del negocio. Tal importancia se refleja en el creciente número de API Web públicas, recogidas en catálogos como APIS.GURU (con más de 800 API) o PROGRAMMABLEWEB (con más de 19000 API). En la práctica, estas API Web son "REST-like", lo que significa que se adhieren parcialmente al estilo arquitectónico conocido como *Transferencia de Estado Representacional* (*Representational State Transfer*, REST en inglés). REST es una descripción técnica que define los principios, propiedades y restricciones para construir sistemas hipermedia distribuidos en Internet. De hecho, REST es un paradigma de diseño y no propone ningún estándar. Por ello, tanto el desarrollo como el consumo de API REST son tareas difíciles y que demandan mucho tiempo de los proveedores y los clientes de API. En realidad, los proveedores de API deben tener una estrategia de API sólida que promueva la facilidad de uso al mismo tiempo que respete las limitaciones de REST. Por otra parte, implementar aplicaciones para consumir estas API normalmente requiere del envío de mensajes HTTP y el uso de JSON o XML para representar datos. Este proceso es completamente manual y los desarrolladores tienen una gran dificultad para integrar API Web con sus aplicaciones debido a la falta de definiciones legibles por máquinas. Así, las API Web adoptaron una aproximación basada en descripciones informales en lenguaje natural explicando lo que proponen. Recientemente, con el objetivo de estandarizar la manera de describir API REST, un consorcio formado por los representantes prin-

cipales del mercado ha lanzado la *Iniciativa OpenAPI* (*OpenAPI Initative*, OAI en inglés). Esta iniciativa tiene como objetivo crear una especificación neutral, portable y abierta para describir API REST. La OAI ha logrado atraer a grandes compañías y la especificación OpenAPI se ha convertido en la elección de referencia para la descripción de API REST. Por otra parte, OData (del inglés *Open Data Protocol*) ha emergido como una especificación para API Web que es especialmente útil para mostrar y consultar fuentes de datos como API REST. La versión actual de OData (versión 4.0) ha sido aprobada como una estándar OASIS.

El objetivo de esta tesis es facilitar el diseño, implementación, composición y consumo de API REST, enfocándose especialmente a la especificación OpenAPI y el protocolo OData, y apoyándose en el del *Desarrollo de Software Dirigido por Modelos* (DSDM) . El DSDM es una metodología que promueve el uso de modelos y transformaciones para elevar el nivel de abstracción y automatización en el desarrollo de software, respectivamente. Esta tesis propone las siguientes contribuciones: (i) EMF-REST, una aproximación para generar API REST para modelos con el objetivo de promover el manejo de modelos en ambientes distribuidos; (ii) APIDISCOVERER, una aproximación para inferir automáticamente especificaciones OpenAPI a partir de ejemplos de llamadas de API REST, ayudando a los desarrolladores a facilitar el uso de sus API sin escribir completamente las especificaciones API; (iii) APITESTER, una aproximación para generar casos de prueba para API REST apoyándose en sus especificaciones OpenAPI, permitiendo evaluar si el comportamiento de una API conforma a su especificación; (iv) APIGENERATOR, una aproximación dirigida por modelos para automatizar la generación de API REST OData a partir de modelos conceptuales; and (v) APICOMPOSER, una aproximación basada en modelos para componer automáticamente API REST dados sus modelos de datos. Estas contribuciones constituyen un ecosistema que avanza el estado del arte en el área de la ingeniería del software referida a la automatización de las tareas relacionadas con el desarrollo y consumo de API REST. Además, creemos que estas contribuciones aportan un alto nivel de agilidad en el desarrollo de API Web.

Resum

Les API web s'han convertit cada vegada més en un actiu clau per a les empreses, que han promogut la seva implementació i integració en les seves activitats quotidianes. Aquesta importància es reflecteix en la creixent quantitat d'API web públiques disponibles, que figuren a diversos catàlegs (p. ex., APIS.GURU amb més de 800 API o PROGRAMABLEWEB amb més de 19 000 API). A la pràctica, la majoria d'aquestes API web són "REST-like"', el que significa que s'adhereixen parcialment a l'estil arquitectònic conegut com *Transferència d'Estat Representacional* (*Representational State Transfer*, REST en anglés). REST és una descripció tècnica que descriu els principis, propietats i restriccions per construir sistemes d'hipermèdia distribuïts a Internet. De fet, REST és un paradigma de disseny i no proposa cap estàndard. Com a conseqüència, tant desenvolupar com consumir API REST són tasques difícils i costoses per als proveïdors i clients de l'API. De fet, els proveïdors d'API haurien de tenir una estratègia API sòlida que afavoreixi la facilitat d'ús, tot respectant les restriccions REST. D'altra banda, escriure aplicacions per consumir aquestes API normalment requereix enviar peticions HTTP i utilitzar JSON o XML per tal de representar les dades. Aquesta tasca és totalment manual i els desenvolupadors tenen dificultats per integrar API web a les seves aplicacions a causa de la manca de definicions que poden ser llegides per les màquines. De fet, les API web van adoptar un enfocament basat en descripcions textuals informals en llenguatge natural i orientades a ser llegides per persones. Recentment, i amb l'objectiu d'estandarditzar la manera de descriure les API REST, un consorci d'actors importants en el mercat de

l'API ha llançat la *Iniciativa OpenAPI* (*OpenAPI Initiative*, OAI en anglés). Aquesta iniciativa té com a objectiu crear una especificació neutral, portable i oberta per descriure les API REST. L'OAI ha aconseguit atraure empreses importants, i l'especificació OpenAPI s'ha convertit en l'elecció de referència per descriure les API REST. D'altra banda, l'*Open Data Protocol* (OData) és una especificació emergent per a les API web que és especialment útil per exposar i consultar fonts de dades emprant API REST. La versió actual de OData (versió 4.0) ha estat aprovada com un estàndar OASIS.

L'objectiu d'aquesta tesi és facilitar el disseny, la implementació, la composició i el consum de les API REST, especialment aquelles que segueixen l'especificació OpenAPI i elprotocol OData, basant-se en tècniques d'*Enginyeria Dirigida per Models* (*Model-driven Engineering*, MDE en anglés). MDE és una metodologia que promou l'ús de models i transformacions per pujar el nivell d'abstracció i automatització en el desenvolupament de programari. Aquesta tesi proposa les següents contribucions: (i) EMF-REST, un enfocament per generar API REST a partir de models, promovent així la gestió de models en entorns distribuïts; (ii) APIDISCOVERER, una proposta per inferir automàticament les especificacions OpenAPI per a les API REST a partir d'exemples, ajudant així als desenvolupadors a augmentar l'exposició de les seves API sense escriure completament les especificacions de les API; (iii) APITESTER, una proposta per generar casos de prova per a les API REST basant-se en les seves especificacions OpenAPI, i que serveix per a avaluar que el comportament d'una API s'ajusta a la seva especificació; (iv) APIGENERATOR, una proposta basada en tècniques de MDE per a automatitzar la generació d'API REST per a OData directamente desplegables a partir de models conceptuals; i (v) APICOMPOSER, una solució lleugera basada en models per permeteix compondre automàticament les API REST a partir d'els seus models de dades. Aquestes contribucions constitueixen un ecosistema que avança l'estat de l'art al camp de l'enginyeria de programari automàtica per al desenvolupament i el consum de les API REST. Creiem que aquestes contribucions tenen un gran valor per als desenvolupadors de les API web que volen agilitzar les seves tasques de desenvolupament.

Table of Contents

List of Tables

List of Figures

Listings

List of Algorithms

CHAPTER

1 Introduction

The Web was born at CERN in 1989[1]. It started as a software project called "WordWideWeb" led by Tim Berners-Lee with the goal to facilitate knowledge sharing. In the first ever Web page created in 1991[2], Berners-Lee wrote:

> "The WorldWideWeb (W3) is a wide-area hypermedia information retrieval initiative aiming to give universal access to a large universe of documents."

What started as a non-profit project to facilitate knowledge sharing reached an unexpected success, resulting in millions of Web users within only few years. However, the Web core protocols were not regulated nor uniformly implemented and lacked support of caching mechanism [Mas11]. Also, the Web had grown substantially faster than the Internet in terms of number of hosts [Gra96] and Web traffic was exceeding the capacity of the Internet infrastructure [Mas11].

In 1993, Roy Fielding, co-founder of the Apache HTTP Server Project[3], became concerned about the scalability issue of the Web. He identified a set of constraints that should be uniformly satisfied to allow the Web to expand. These constraints collectively define what was referred to as the Web's architectural style.

In order to enforce the Web's architecture, Fielding, Berbers-Lee and others wrote a new version of the Hypertext Transfer Protocol (HTTP) (i.e.,

[1] `https://home.cern/topics/birth-web`
[2] `http://info.cern.ch/hypertext/WWW/TheProject.html`
[3] `http://httpd.apache.org`

HTTP/1.1) [Fie+99] and formalized the syntax of the Uniform Resource Identifier (URI) in RFC 3986 [BFM05]. These new specifications got quickly adopted and paved the way for the Web to grow. In 2000, and after fixing the Web scalability crisis, Roy Fielding described the Web's architectural style in his Ph.D dissertation [Fie00]. "Representational State Transfer (REST)" was the name he gave to this description.

Due to its lightweight nature, adaptability to the Web, and scaling capability, REST has become the preferred style for building Web APIs. With the emergence of REST, Web APIs have become the backbone of Web, Cloud, mobile applications and even many Open Data initiatives aiming at facilitating the access to information using Web APIs rather that the Resource Description Framework (RDF). In fact, Web APIs are now mission critical for any number of businesses, and their implementation is increasing [BW14].

The growing importance of REST APIs has been supported by the proposal of several specification languages aimed at formally describing REST APIs and therefore facilitating their discovery and integration (e.g., SWAGGER[4], API BLUEPRINT[5], RAML[6]). Aiming at standardizing the way to describe REST APIs, a consortium of major actors in the API market has launched the OpenAPI Initiative (OAI)[7]. This initiative has the objective of creating a vendor neutral, portable, and open specification for describing REST APIs. OAI has succeeded in attracting major companies and the OpenAPI specification has become the choice of reference to describe REST APIs. Open Data Protocol (OData), on the other hand, is an emerging specification for Web APIs which is specially useful to expose data sources as REST APIs. OData allows creating resources which are defined according to a data model and can be queried by Web clients using a URL-based query language in an SQL-like style. The current version of OData has been approved as an OASIS standard [PHZ14c].

Model-Driven Engineering (MDE) is a methodology that promotes the use of models to raise the level of abstraction and automation in software development [Sel03]. MDE relies on models and model transformations for the specification and generation of software artifacts, thus hiding the complexity of the target technology. In the last decade, MDE approaches have been successfully applied in several industries including the automotive industry, aerospace, and information systems [HRW11; Hut+11].

[4] `http://swagger.io`
[5] `https://apiblueprint.org/`
[6] `https://raml.org/`
[7] `https://www.openapis.org/`

This thesis proposes a model-driven approach to facilitate the design, implementation, consumption, and composition of REST APIs, targeting the OpenAPI specification and OData protocol.

1.1 Problem Statement

Web APIs are key assets for a number of businesses, thus their implementation and integration are increasing [BW14]. Such importance is reflected by the growing number of public Web APIs, listed in catalogs such as API HARMONY[8] (over 1 000 APIs), MASHAPE[9] & RAPIDAPI[10] marketplace (over 8 000 APIs), APIS.GURU[11] (over 800 APIs), or PROGRAMMABLEWEB (over 19 000 APIs). In practice, most of these Web APIs are "REST-like", meaning that they adhere to REST constraints to some extent [Rod+16]. Writing applications to consume these APIs typically requires sending HTTP requests and using JavaScript Object Notation (JSON) or Extensible Markup Language (XML) to represent data. This is in theory an easy task but in practice developers are having a hard time integrating Web APIs to their applications [EZG15] due to the lack of machine-readable descriptions and incomplete documentations, among others. On the other hand, developing REST APIs is a challenging and time-consuming task for API providers who should have a sound API strategy fostering ease of use while respecting REST constraints.

In the following we describe the challenges addressed by this thesis, namely: (i) *APIfication*, (ii) modeling and discovering REST APIs, (iii) testing REST APIs, (iv) generating REST APIs, and finally (v) composing REST APIs.

1.1.1 *APIfication*

Web APIs are the new communication bus of modern organizations, allowing previously locked systems to integrate with each other without any human interaction. Thus, approaches that enable API-based access to these systems are needed. This is what we call *APIfication*. In the context of this thesis we target the *APIfication* of models. In fact, current modeling

[8] `https://apiharmony-open.mybluemix.net/public`
[9] `https://market.mashape.com/`
[10] `https://rapidapi.com/`
[11] `https://apis.guru/openapi-directory/`

environments (e.g., XTEXT[12], EPSILON[13] or EMFTEXT[14]) and frameworks (e.g., the plethora of modeling facilities in Eclipse such as Eclipse Modeling Framework (EMF) or Graphical Modeling Framework (GMF)) have successfully contributed to the broad use of MDE techniques. However, these frameworks are constrained to be used in current heavyweight desktop environments (e.g., ECLIPSE IDE). Web APIs offer a suitable solution to enable the portability of modeling environments to the Web, thus facilitating the collaborative development of model-based applications.

1.1.2 *Modeling and discovering REST APIs*

Despite their popularity, REST APIs do not typically come with any specification of the functionality they offer. Instead, REST "specifications" are typically informal textual descriptions [PZL08] (i.e., documentation pages), which hampers their integration. Indeed, developers need to read documentation pages, write code to assemble the resource URIs and encode/decode the exchanged resource representations. This manual process is time-consuming and error-prone and affects not only the adoption of APIs but also their discovery. This situation triggered the creation of many specification languages to describe REST APIs (e.g., SWAGGER[15], API BLUEPRINT[16], RAML[17]), which makes choosing a format or another subjective to API providers. To face this situation, the OAI launched a vendor neutral, portable, and open specification for describing REST APIs. OAI has succeeded in attracting major companies in the API ecosystem including its competitors (e.g., MULESOFT[18], the creator of RAML; APIARY[19], creator of API BLUEPRINT) and the OpenAPI specification (formally SWAGGER specification) won the API specification battle. In fact, OpenAPI benefits from the surrounding ecosystem of tools to help automatize many API development tasks such as generating Software Development Kits (SDKs) for different frameworks (e.g., using APIMATIC[20], SWAGGER-CODEGEN[21]) and

[12] https://www.eclipse.org/Xtext/
[13] http://www.eclipse.org/epsilon/
[14] https://marketplace.eclipse.org/content/emftext
[15] http://swagger.io
[16] https://apiblueprint.org/
[17] https://raml.org/
[18] https://swagger.io/blog/news/mulesoft-joins-the-openapi-initiative/
[19] https://blog.apiary.io/We-ve-got-Swagger
[20] https://apimatic.io/
[21] https://github.com/swagger-api/swagger-codegen

generating documentation (e.g., using SWAGGER UI[22], REDOC[23]). There is, therefore, a need of approaches to automatically discover OpenAPI specifications for existing Web APIs. The importance of such approaches is supported by the emergence of new initiatives to infer OpenAPI specifications from, for instance, other specification formats (e.g., API TRANSFORMER[24]) or documentation pages (e.g., [Yan+18], [CFB17]).

1.1.3 *Testing REST APIs*

Testing Web APIs, and specially REST ones, is a complex task due to the lack of machine-readable descriptions [BHH13]. Many approaches have proposed test case generation but mostly for Simple Object Access Protocol (SOAP)-based Web APIs by relying on their Web Services Description Language (WSDL) documents (e.g., [Bai+05; Bar+09; HM08; OX04]). However, works targeting test cases generation for REST APIs are rather limited (e.g., [Arc17]). In the last years we have witnessed an increasing number of open source and commercial tools offering automated API testing for different specification formats such as OpenAPI (e.g., READY API![25], DREDD[26], RUNSCOPE[27]). Nevertheless, these tools only cover nominal test cases (i.e., using correct data) and neglect the fault-based ones (i.e., using incorrect data) which are important when selecting reliable Web APIs for composition [BHH13]. Furthermore, they require extra configuration and the provision of input data.

1.1.4 *Generating REST APIs*

While REST provides a lightweight solution to create Web APIs, designing *true* REST APIs is not a trivial task. A REST API should adhere to REST constraints and not any Web API built on HTTP should be called REST, which Roy Fielding weighed in[28]. In fact, recent reports showed that most APIs do not fully conform to Fielding's definition of REST [Rod+16]. OData[29] is a protocol that defines a set of best practices for building and consuming *RESTful* APIs. In the last years, OData has evolved to become

[22] https://swagger.io/tools/swagger-ui/

[23] https://github.com/Rebilly/ReDoc

[24] https://apimatic.io/transformer

[25] https://smartbear.com/product/ready-api/overview/

[26] http://dredd.readthedocs.io/

[27] https://www.runscope.com/

[28] http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven

[29] http://www.odata.org/

the natural choice for creating data-centric REST APIs (i.e., REST APIs providing access to data sources), specially for Open Data initiatives aiming at facilitating the access to information using Web services. As a result, many service providers have integrated OData in their solutions (e.g., SAP, IBM WebSphere or JBoss Data Virtualization). There are some SDKs for developing OData applications (e.g., RESTIER[30], APACHE OLINGO[31], SDL ODATA FRAMEWORKS[32]) and commercial tools for exposing OData services from already existing data sources (e.g., CLOUD DRIVERS[33], ODATA SERVER[34], SKYVIA CONNECT[35]), but they still require advanced knowledge about OData to implement the business logic of the service, and provide limited support for the OData specification, respectively. Other tools such as SIMPLE-ODATA-SERVER[36] and JAYDATA[37] allow generating a basic OData server from both an entity model expressed in OData format and the corresponding database, but they only cover a subset of the OData protocol.

1.1.5 *Composing REST APIs*

By enabling programmatic access to data sources, Web APIs promote the creation of specialized data-driven applications that combine data from different sources to offer user-oriented value-added APIs. Creating such applications requires API discovery/understanding/composition and coding. Such tasks are not trivial since developers should [EZG14; Aué+18]: (i) know the operations and data models of the API to compose; (ii) define the composition strategy; and (iii) implement an application (usually another Web API) realizing such strategy. While automatic Web API composition has been heavily studied for the classical WSDL/SOAP style [She+14], REST API composition is of broad and current interest specially after the emergence of new REST API specifications such as the OpenAPI specification and OData.

1.2 APPROACH

We propose an MDE approach to address the challenges described in the previous section, namely: *APIfication* of models, discovery of REST API spec-

[30] https://github.com/OData/RESTier
[31] https://olingo.apache.org/
[32] https://github.com/sdl/odata
[33] http://www.cdata.com/odata/
[34] https://rwad-tech.com/
[35] https://skyvia.com/connect/
[36] https://github.com/pofider/node-simple-odata-server
[37] https://github.com/jaystack/jaydata

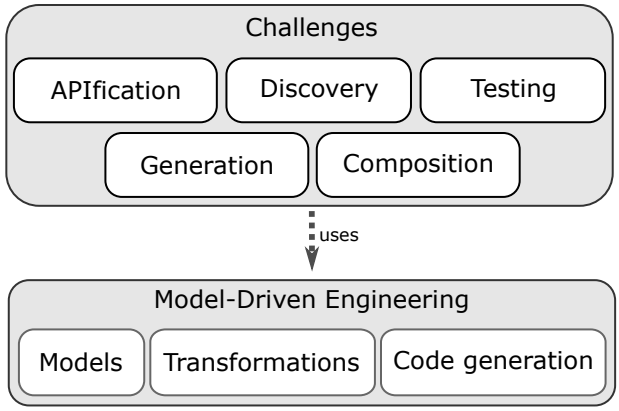


Figure 1.1: The challenges addressed in this thesis and the main Model-Driven Engineering axes.

ifications, API testing, REST APIs generation, and REST APIs composition, as depicted in Figure 1.1.

We created a set of intermediate model-based representations to define the components of our approach such as: (i) the OpenAPI specification, (ii) the OData protocol, and (iii) the test cases. These model-based representations are used throughout this thesis to create our model-driven approach.

1.3 CONTRIBUTIONS

Our model-based representations become the foundations on which we built the main contributions of this thesis which are depicted in Figure 1.2. The figure shows how these contributions are linked together to constitute an ecosystem of tools addressing the challenges described in Section 1.1. In the following we give a brief description of each contribution. Note that these contributions are presented in the same order as the challenges addressed by this thesis.

EMF-REST. One of the objectives of this thesis is to unlock model management from the silo of desktop modeling environments. We propose an approach that generates REST APIs from models, thus promoting model management in distributed environments. As can be seen in Figure 1.2 (see block a), EMF-REST takes as input an EMF model and generates a REST API manipulating such model. The

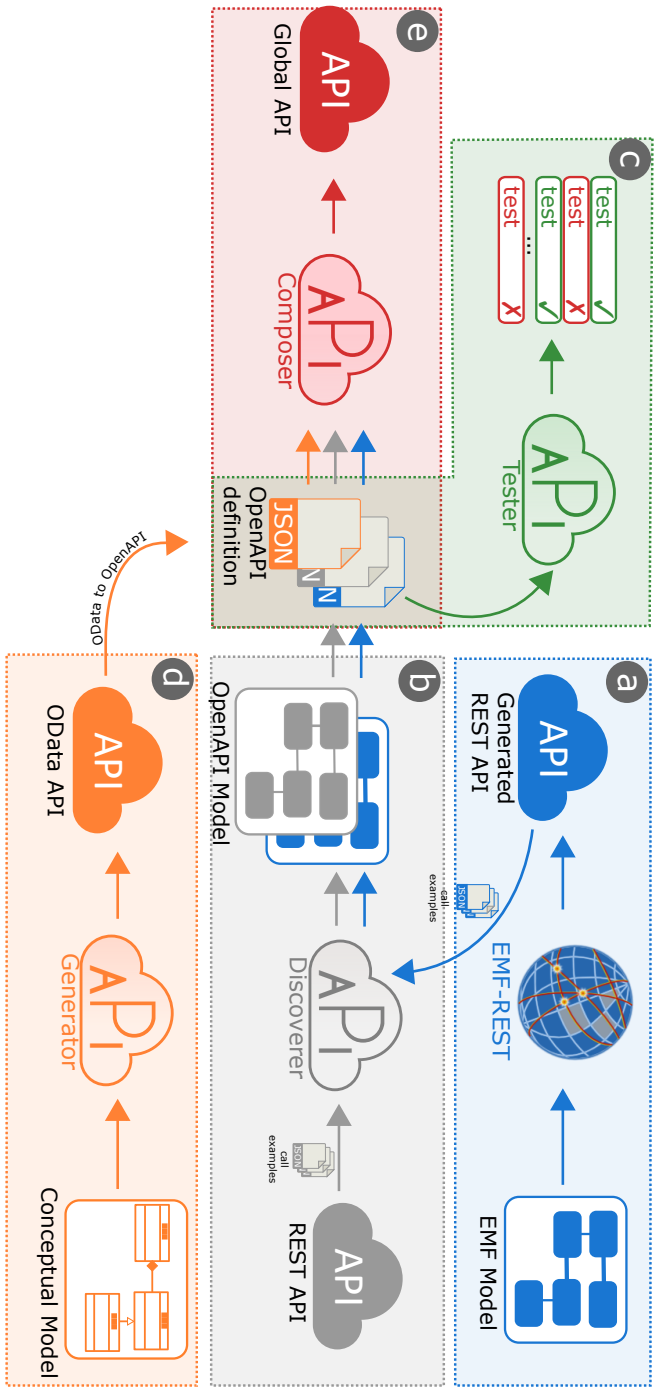


Figure 1.2: Main contributions of this thesis and the link between them.

generated REST API relies on well-known libraries with the aim of facilitating its understanding and maintainability. Adopting a Web-based approach to manipulate models would promote the collaboration between modelers, thus facilitating the collaborative development of new software models. Our solution advances towards the portability of modeling tools to the Web.

APIDISCOVERER. We propose an example-based approach to allow developers to automatically infer OpenAPI specifications for REST APIs, and optionally, store them in a community-oriented directory. APIDISCOVERER generates OpenAPI specifications for REST APIs from call examples of such APIs. As can be seen in Figure 1.2 (see bloc b), APIDISCOVERER takes as input a set of call examples of an existing or generated REST API, then discovers the OpenAPI model corresponding to the API. The later is finally transformed to an OpenAPI definition in JSON format. Our approach is an example-driven approach, meaning that the OpenAPI specification is derived from a set of examples showing its usage. The use of examples is a well-known technique in several areas such as Software Engineering [Lóp+15; Nie+07] and Automatic Programming [Fra+16]. From the user's point of view, our approach facilitates the discovery and integration of existing APIs, favouring software reuse. From the API builder's point of view, our approach helps increase the exposure of the API without the need to learn and fully write the API specifications or alter the API code, thus allowing fast-prototyping of API specifications and leveraging on several existing tool sets featuring API documentation generation (e.g., using SWAGGER UI[38]) or API monitoring and testing (e.g., using RUNSCOPE[39]).

APITESTER. We propose an approach to generate test cases for REST APIs relying on their specifications, in particular the OpenAPI one, with the goal to ensure a high coverage level for both nominal (i.e., correct input data) and fault-based (i.e., incorrect input data) test cases. APITESTER takes as input an OpenAPI specification, then generates a set of test cases for the API under scrutiny, as depicted in Figure 1.2 (see block c). We define a set of parameter inference rules to generate the input data required by the test cases. We follow a model-driven

[38] http://swagger.io/swagger-ui/
[39] https://www.runscope.com/

approach, thus favoring reuse and automation of test case generation. Therefore, we define a TestSuite metamodel to represent test case definitions for REST APIs. Models conforming to this metamodel are created from REST API definitions, and later used to generate the executable code to test the API.

APIGENERATOR. We propose a model-driven approach to automate the generation of ready-to-deploy REST APIs. As can be seen in Figure 1.2 (see block d), APIGENERATOR takes as input a Unified Modeling Language (UML) class diagram (or Entity Relationship (ER) model), then generates an OData API reflecting the data model. Our approach derives all the artifacts required to have the OData API up and running on top of a relational database conforming to the model definition, including the transformation of OData requests to Structured Query Language (SQL) queries and complying with OData protocol. We rely on the OData metamodel, which is used to represent and generate the OData API; thus allowing us to leverage on the plethora of existing modeling tools and therefore enabling our approach to use other input models or target technologies.

APICOMPOSER. We propose a lightweight model-based approach to automatically compose data-oriented REST APIs. As can be seen in Figure 1.2 (see block e), APICOMPOSER takes as input a set of OpenAPI definitions, then generates a global API composing the input APIs. The input OpenAPI definitions can be: (i) provided by the API provider, (ii) generated by tools such as APIDISCOVERER, or (iii) derived from other formats (e.g., from OData[40]). In our approach, the data schema behind the global API is generated during the composition process based on the discovery of matches between the individual data schema of each single API. All these schemas are represented as models and their manipulations (e.g., concept matching or composition) are implemented as model transformations. As a result of the composition, we obtain a global API that hides the complexity of the composition process to the user. Indeed, a user queries the global API and, in a transparent way, the global query triggers a fully automatic process that accesses the individual APIs and combines their data to generate a single response. To facilitate the consumption of the global API, we expose it as an OData service, thus allowing the use of OData query language to interact with the APIs.

[40] https://oasis-tcs.github.io/odata-openapi/

As depicted in Figure 1.2 and explained above, the contributions of this thesis can be used together or separately to automate different tasks related to REST APIs development, consumption, and composition. We believe our contributions bring more agility to REST APIs development tasks and advance the state of the art of automatic software engineering for REST APIs.

1.4 RESULTS

This section presents what has been produced in the context of this thesis including: (i) the scientific publications, and (ii) the tools implementing the contributions.

1.4.1 *Scientific Production*

The contributions of this thesis have been published in the following papers:

- International conferences (full research papers):
  - **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2018, October). Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach. In Proceedings of the IEEE International Enterprise Distributed Object Computing Conference (EDOC), (pp. 181-190).
  - **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2018, May). Model-Driven Development of OData Services: An Application to Relational Databases. In Proceedings of the 12th International Conference on Research Challenges in Information Science (RCIS), (pp. 1-12).
  - **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2017, July). Example-Driven Web API Specification Discovery. In Proceedings of the 13th European Conference on Modelling Foundations and Applications (ECMFA), (pp. 267-284).
  - **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Gómez, A., Tisi, M., Cabot., J. (2016, April). EMF-REST: Generation of RESTful APIs from Models. In Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC), (pp. 1446-1453).
- International conferences (short and demo papers):

- **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2018, September). APIComposer: Data-Driven Composition of REST APIs. In Proceedings of the European Conference on Service-Oriented and Cloud Computing (ESOCC), (pp. 161-169).
- **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2018, June). OpenAPItoUML: A Tool to Generate UML Models from OpenAPI Definitions. In Proceedings of the International Conference on Web Engineering (ICWE), (pp. 487-491).
- **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2017, June). A UML Profile for OData Web APIs. In Proceedings of the International Conference on Web Engineering (ICWE), (pp. 420-428).

◇ National Conferences:

- **Ed-Douibi, H.**, Cánovas Izquierdo, J. L., Cabot., J. (2018, September). Una Propuesta para Componer APIs Orientadas a Datos. Las Jornadas de Ingeniería del Software y Bases de Datos (JISBD).

1.4.2 *Tools Developed*

All the tools developed in the context of this thesis are listed below.

OPENAPIMM. `https://github.com/SOM-Research/openapi-metamodel`
An Eclipse plugin which includes the OpenAPI metamodel and a de/serializer to generate OpenAPI models from JSON files and *vice versa*.

OPENAPIPROFILE. `https://github.com/SOM-Research/openapi-profile`
An Eclipse plugin which includes a UML profile for OpenAPI and an editor based on PAPYRUS[41] which enables the annotation of UML models with OpenAPI stereotypes.

ODATAPROFILE. `https://github.com/SOM-Research/odata`
An Eclipse plugin which includes a UML profile for OData protocol and an editor based on PAPYRUS enabling the annotation of UML models with OData stereotypes.

[41] `https://www.eclipse.org/papyrus/`

OPENAPITOUML. https://github.com/SOM-Research/openapi-to-uml
An Eclipse plugin which allow the generation of UML models from OpenAPI definitions.

EMF-REST. http://emf-rest.com
An Eclipse plugin which allows the generation of REST APIs to manage EMF models.

APIDISCOVERER. https://github.com/SOM-Research/APIDiscoverer
A Web application which allows the generation of OpenAPI definitions from API call examples.

APITESTER. https://github.com/SOM-Research/test-generator
An Eclipse plugin which allows the generation of test cases from OpenAPI definitions.

APIGENERATOR. https://github.com/SOM-Research/odata-generator
An Eclipse plugin which allows the generation of REST APIs following OData protocol from UML models. The plugin also includes the OData metamodel.

APICOMPOSER. https://github.com/SOM-Research/api-composer
A Web application which allows the composition of REST APIs by relying on their OpenAPI definitions.

1.5 THESIS OUTLINE

The remainder of this thesis is structured as follows:

CHAPTER 2 describes the basic concepts that are used as the foundations of this thesis. We start by introducing REST architectural style and REST APIs. Later, we introduce the OpenAPI specification and OData protocol. Finally we present the MDE paradigm and its main standards.

CHAPTER 3 presents a set of model representations to manipulate OpenAPI definitions and OData entity models. For each specification we provide a metamodel and a UML profile as well as a set of

tools to manipulate them. These model representations are used in the following chapters to achieve the intended model-based solutions.

CHAPTER 4 details EMF-REST, our approach to *APIfy* models by deriving REST APIs from EMF models.

CHAPTER 5 presents APIDISCOVERER, our approach to generate OpenAPI definitions from API call examples.

CHAPTER 6 presents APITESTER, our approach to generate test cases from OpenAPI definitions in order to assess that such definitions conform to their back-end implementations

CHAPTER 7 presents APIGENERATOR which provides an approach to generate REST APIs from conceptual models.

CHAPTER 8 presents APICOMPOSER, our approach to compose data-centric REST APIs by relying on their OpenAPI definitions.

CHAPTER 9 concludes this thesis by summarizing its main contributions and introducing some ideas for future work.

CHAPTER

2 Background

This chapter presents some of the concepts, technologies, and tools used as foundations for this thesis. We first present REST APIs which covers the REST architecture style and the REST APIs definition initiatives targeted by this thesis, namely: the OpenAPI specification and OData protocol. Next, we introduce MDE and its main components which are used in the contributions of this thesis.

2.1 REST APIs

A REST or *RESTful* API is a Web Application Programming Interface (API) which follows the Representational State Transfer architectural style. REST is a technical description of how the World Wide Web works. It has been described by Roy Fielding in 2000 [Fie00] and has become a popular approach to design Web APIs. REST outlines the architectural principles, properties, and constraints to build Internet-scale distributed hypermedia systems. This section presents a brief introduction of REST APIs.

2.1.1 *REST Architectural Style*

In the year 2000, Roy Fielding described the Web's architectural style in his Ph.D dissertation [Fie00]. "Representational State Transfer" was the name he gave to this description which is now referred to as REST. REST became the prominent architectural style to design Web APIs mainly due to

its adaptability to Web as it allows creating Web APIs by relying only on URIs and HTTP messages.

The REST architectural style describes six constraints, namely: *Client-Server*, *Stateless*, *Cache*, *Uniform Interface*, *Layered System*, and *Code on Demand*. In general, Web APIs cover the constraints: *Client-Server*, *Stateless*, *Cache*, and *Uniform Interface*. In the following we give a brief description of each of the above constraints.

CLIENT-SERVER. This constraint states that a client should send a request to the server, then the server either rejects or performs the request, and then sends a response back to the client.

STATELESS. This constraint sates that the server cannot hold the state of the client beyond the current request. A client only exists when he/she is making a request to the server. Therefore, the client must include all the required contextual information in each request to the Web server.

CACHE. The *cache* constraints instructs a server to declare the *cacheability* of each response data (i.e., whether a client can reuse previous response data). A client can locally match its requests to the previously received responses, thus saving a round trip communication with the server.

UNIFORM INTERFACE. This is an umbrella term for four interface constraints defined by Fielding, namely: *identification of resources*, *manipulation of resources through representations*; *self-descriptive messages*; and, *hypermedia as the engine of application state*. Together, these constraints define a "uniform interface" which allow Web components to interoperate consistently. These four interface constraints are briefly explained below.

IDENTIFICATION OF RESOURCES. A URI identifies a resource. Thus, a resource is anything that can be identified by a URI. Also, a URI identifies a resource and not its state. While the state of a resource changes, its URI stays the same.

MANIPULATION OF RESOURCES THROUGH REPRESENTATIONS. Representations are transferred between REST components to manipulate resources. A representation is a "sequence of bytes"

which captures the current or intended state of a resource. It consists of an encapsulation of the information (data or metadata) of the resource, encoded using a format (e.g., XML, JSON).

SELF-DESCRIPTIVE MESSAGES. REST enables intermediate processing by constraining messages to be self-descriptive. A message must contain all the necessary information to understand the representation of the resource it describes.

HYPERMEDIA AS THE ENGINE OF APPLICATION STATE. Resources should link to each other in their representations using hypermedia links and forms.

Section 2.1.2 will describe the uniform interface of HTTP which is the mostly used in the design of REST APIs.

LAYERED SYSTEM. This constraint allows adding proxy and gateway components to the client-server architecture. A proxy receives requests from components (i.e., clients, proxies, or gateways) and it does not handle the request. Instead, it transfers the request to another component (i.e., a server, a proxy, or a gateway) and waits for a response. When a response is received, the proxy sends it back to the component that sent the request. *Layered System* is more important to the deployment of Web APIs than to their design [RAR13].

CODE ON DEMAND. This constraint indicates that a server can extend the functionality of a client on runtime by sending executable code (e.g., Java Applets or JavaScript). The *Code on Demand* constraint is optional and not generally used in the context of Web APIs [RAR13].

2.1.2 *HTTP's Uniform Interface*

HTTP is an application-level protocol that defines operations for transferring representations between clients and servers. Although REST is protocol-agnostic, in practice most REST APIs use HTTP as a transport layer. Therefore, HTTP's uniform interface is generally applied to design REST APIs. In fact, the list of methods and their semantics is fixed in HTTP and these semantics are independent of the resources. Ideally, REST APIs map HTTP methods with CRUD operations (i.e., Create, Read, Update, and Delete) as follows:

GET is used to retrieve a representation of a resource. It is a read-only, idempotent and safe operation.

PUT is used to update a resource on the server. It is idempotent as well.

PATCH is similar to PUT but instead of updating the entire resource's representation, this method allows the client to send only the change he/she intends to make. `PATCH` method was added to HTTP in 2010 and defined in RFC 5789[1].

POST is used to create a resource on the server based on the data included in the body request. It is nonidempotent and unsafe.

DELETE is used to remove a resource on the server. It is idempotent as well.

The following HTTP methods are also used in REST APIs:

HEAD is similar to GET but returns only a response code and the header associated with the request.

OPTIONS is used to request information about the communication options of the addressed resource (e.g., security capabilities such as Cross-Origin Resource Sharing (CORS) [W3C14a]).

2.2 REST APIs Specifications and Protocols

REST is an architectural style but does not provide any specification or standard to describe REST APIs. The Web Application Description Language (WADL) [Had06], a specification language for REST APIs, was first proposed to describe REST APIs but failed to get adoption due to its complexity and limitations to fully describe REST APIs. Thus, at the beginning, most REST APIs adopted a human-oriented approach based on informal textual descriptions explaining what they propose (i.e., documentation pages) [PZL08]. Hence, to consume Web APIs, developers needed to read the different documentation pages, manually write the code to assemble the resource URIs and encode/decode the exchanged resource representations. This manual process is time-consuming and error-prone which hinders the automatic discovery and integration of REST APIs. This situation has triggered the

[1] `https://tools.ietf.org/html/rfc5789`

creation of languages and protocols to describe REST APIs in a way that both humans and machine can understand, and to define methods for building and consuming REST APIs, respectively. In the following we introduce the specifications targeted by this dissertation, namely: the OpenAPI specification and OData protocol.

2.2.1 *The OpenAPI Specification*

The OpenAPI specification[2] is "a standard, programming language-agnostic interface description for REST APIs". It has been created and is maintained by the OpenAPI Initiative which is a consortium of API contributers (members of 3SCALE, APIGEE, CAPITAL ONE, GOOGLE, IBM, INTUIT, MICROSOFT, PAYPAL, RESTLET and SMARTBEAR) which aims at standardizing the way to describe REST APIs. The purpose of OpenAPI is to allow both humans and machines to discover and understand the capabilities of the API without the need to check the source code, read the documentation, or inspect the network traffic.

The first version of the OpenAPI specification (version 2) is identical to SWAGGER which was donated by SMARTBEAR[3] to OAI. This version is still widely used even after the release of the third version of the OpenAPI specification. For instance, APIS.GURU[4], a public repository of OpenAPI definitions, lists more than 800 OpenAPI 2 definitions. In this thesis we rely on the OpenAPI 2.

The OpenAPI specification allows the description of an API in terms of:

- ◇ Available endpoints,
- ◇ operations on each endpoint organized using HTTP methods,
- ◇ parameters and responses of each operation,
- ◇ data structures manipulated by the API,
- ◇ authentication methods,
- ◇ metadata information: contact, license, terms of use and other information.

The files describing an API conforming to the OpenAPI specification can be written either in JSON or YAML Ain't Markup Language (YAML).

[2] https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md/

[3] https://smartbear.com/

[4] https://apis.guru/openapi-directory/

As a reference example for the specification, the OpenAPI community has released the Petstore API[5] which provides a description of a pet store REST service. The Petstore API allows users to manage pets (e.g., add a pet to the store, find pets, delete a pet), orders (e.g., place an order for a pet, delete an order), and users (e.g., create a user, delete a user).

Listing 2.1 shows a partial representation of the OpenAPI definition of the Petstore API in JSON format. As can be seen, the root object of the definition includes: (i) the version of the OpenAPI specification (i.e., `swagger` field[6]), (ii) the metadata description of the API (i.e., `info` field), (iii) the host serving the API (i.e., `host` field), (iv) the base path which is relative to the host and serving the API (i.e., `basePath` field), (v) the transfer protocol of the API (i.e., `schemes` field, e.g., `http`), (vi) the list of MIME types the API can consume and produce (i.e., `consumes` and `produces` fields, respectively. e.g., `application/json`), (vii) the available paths and operations of the API (i.e., `paths` field), (viii) the data types produced and consumed by the operations of the API (i.e., `definitions` field), and (ix) the security scheme definitions that can be used by the operations of the API (i.e., `securityDefinitions` field).

The `Info` object provides metadata about the API. It includes fields to describe the title of the API (i.e., `title` field), a description (i.e., `description` field), contact information (i.e., `contact` field), license (i.e., `license` field), and version of the API (i.e., `version` filed).

The `Paths` object is a container which holds the relative paths to individual endpoints (e.g., `/pet/findByStatus`, `/pet/findByTags`). Each relative path is appended to the base path in order to construct a full URL. A `Path` object includes fields to describe the available operations on a single path, such as a field name describes an HTTP method (e.g., `get` for a `GET` operation), while its value defines an operation on this path for the corresponding HTTP method.

An `Operation` object describes a single API operation on a path. The operation `getPetsByStatus`, for instance, includes an operation ID (i.e., `operationId` field, e.g., `findPetsByStatus`), the MIME type it produces (i.e., `produces` field), its parameters (i.e., `parameters` field), and the list of possible responses which can be returned after executing this operation, (i.e., `responses` field).

A `Parameter` object describes a single operation parameter. The parameter `status` of the operation `findPetsByStatus` include its name, its location

[5] `http://petstore.swagger.io/`

[6] It must be `2.0` in OpenAPI 2

Listing 2.1: A snippet of the OpenAPI definition of the Petstore API in JSON format

```
{"swagger":"2.0",
  "info":{
    "title": "Swagger Petstore",
    "description": "This is a sample server Petstore server...",
    "contact": {...}, "license": {...}, "version": "1.0.0"
  },
  "host":"petstore.swagger.io",
  "basePath":"/v2","schemes":["http"],
  "consumes": ["application/xml", "application/json"],
  "produces": ["application/xml", "application/json"],...
  "paths":{
    "/pet/findByStatus":{
    "get":{
     "operationId":"findPetsByStatus",
     "produces":["application/xml","application/json"],
     "parameters":[{
       "name":"status",
       "in":"query",
       "description":"Status values that need to be considered for
           filter",
       "required":true,
       "type":"array",
       "items":{
          "type":"string",
          "enum":["available","pending","sold"],
          "default":"available"},
       "collectionFormat":"multi"}],
     "responses":{
        "200":{
         "description":"successful operation",
         "schema":{type":"array",
            "items":{"$ref":"#/definitions/Pet"}}}}},
    "/pet/findByTags":{  },
    "/pet/{petId}":{  },
    ...
    },
  "definitions":{
    "Pet":{
      "type":"object", "required":["name","photoUrls"],
      "properties":{
        "id":{
          "type":"integer",
          "format":"int64"
          },
        "category":{...},
        "name":{...},
        "photoUrls":{...},
        "tags":{..},
        "status":{..}}
      },
    "Category":{...}, "Tag":{...},
    ...},
  "securityDefinitions":{
    "petstore_auth": { "type": "oauth2",..},
    "api_key": { "type": "apiKey",..}}}
```

(i.e., `in` field), a description, a flag indicating whether such parameter is required (i.e., `required` field) and a type (i.e., `type` field). Since the *status* parameter is of type array, the specification includes the definition of its items (i.e., `items` field).

OpenAPI defines five types of parameters, namely:

- ◇ `path`: the parameter value is part of the path. For example in `/pet/{petId}`, the path parameter is `petId`.
- ◇ `query`: the parameter value is appended to the URL. For example in `/pet/findByStatus?status=###`, the query parameter is `status`.
- ◇ `header`: the parameter value is defined as custom headers that are expected as part of the request.
- ◇ `body`: the parameter value is the payload of the HTTP request.
- ◇ `form`: the parameter value is the payload when `application/x-www-form-urlencoded`, `multipart/form-data` are used as the content type of the request.

The `Responses` object is a container which lists the possible responses expected of an operation. The container maps an HTTP response code to the expected response. A `Response` object describes a single response of an API operation. The `Response` object associated with the status code `200` of the operation `findPetsByStatus` includes a description and a definition of the response data structure (i.e., `schema` field) which indicates that the operation returns an array of *pets*.

The `Definitions` object holds the data types produced and consumed by the operations of the API. The `Schema` object allows the definition of a data type based on a subset of the *JSON Schema Specification Draft 4*[7]. JSON Schema describes the structure of a JSON document (e.g., list of properties, constraints). However, OpenAPI does not support all JSON Schema features (e.g., `OneOf` is not supported) and adapts other constraints to the specification (e.g., `items` and `allOf`). Apart from the JSON Schema subset fields, the OpenAPI specification defines other fields for further schema documentation (e.g., `discriminator` to add support for polymorphism, `readOnly` to declare a property as 'read only'). `Schema` definitions can be used to validate instances (i.e., check that constraints are met) or collect instances such that the constraints are satisfied. As can be seen, the `Definitions` object of the Petstore API includes the `Schema` definitions `Pet`, `Category` and `Tag`. The `Pet` definition indicates that a valid instance of *pet*

[7] `http://json-schema.org/`

Listing 2.2: An example of a request to the Petstore API

```
curl -H Accept:application/json -i http://petstore.swagger.io/v2/
    pet/findByStatus?status=available
HTTP/1.1 200 OK
...
[
    {
        "id": 545646663,
        "category": {
            "id": 0,
            "name": "string"
        },
        "name": "doggie",
        "photoUrls": [
            "string"
        ],
        "tags": [
            {
                "id": 0,
                "name": "string"
            }
        ],
        "status": "available"
    },
    ...
]
```

should be of type object and include the properties `id`, `category`, `name`, `name`, `photoUrls`, `tags` and `status`, such as `name` and `photoUrls` are mandatory. The `findPetsByStatus` operation, for instance, returns an array of *pets* instances respecting this definition.

The `SecurityDefinitions` object includes the security schemes available to be used by the specification. The OpenAPI specification supports three security schemes, namely: Basic, API Key, and OAuth 2. The OpenAPI definition of the Petstore API includes the security schemes `petsore_auth` and `api_key` for OAuth 2 and API Key authorizations, respectively. The complete Petstore API definition can be found in OpenAPI website[8].

Listing 2.2 shows an example of a request targeting the `findPetsByStatus` operation using cURL tool[9]. As can seen, the URL of the request is constructed from the protocol scheme (i.e., `http`), the host (i.e., `petstore.swagger.io`), the base path (i.e., v2) and the relative path associated with the `findPetsByStatus` operation (i.e., `/pet/findByStatus`). The URL includes also the query parameter `status` with the value `available`. The request includes the header

[8] `http://petstore.swagger.io/`

[9] `https://curl.haxx.se/`

`Accept:application/json` to request the response in JSON format. The API returns a response with the status code `200` including an array of *pets* having the status `available`.

2.2.2 *OData Protocol*

The Open Data Protocol (OData) is a protocol for creating data-oriented REST APIs with query and update capabilities. It is specially adapted to expose and access information from a variety of data sources such as relational databases, file systems and content management systems. In the last years, OData has evolved to become the natural choice for creating data-centric Web services, specially for Open Data initiatives aiming at facilitating the access to information using Web services rather than RDF [W3C14b] or text based formats. As a result, many service providers have integrated OData in their solutions (e.g., SAP[10], IBM WEBSPHERE[11] or JBOSS DATA VIRTUALIZATION[12]). The current version of OData (version 4.0) has been approved as an OASIS standard [PHZ14c].

Some of the specifications defined by the OData protocol are the following:

COMMON SCHEMA DEFINITION LANGUAGE XML REPRESENTATION. It defines an XML representation of the data model exposed by an OData service [PHZ14b].

COMMON SCHEMA DEFINITION LANGUAGE JSON REPRESENTATION. It defines a JSON representation of the data model exposed by an OData service [PHZ14a].

URL CONVENTIONS It defines a set of rules for constructing URLs to queries the data [PHZ14d].

ODATA JSON FORMAT. It defines the JSON format of the resource representations that are exchanged using OData [HPB14].

ODATA ATOM FORMAT. It defines the Atom format of the resource representations that are exchanged using OData [ZPH14].

[10] `https://www.sap.com`
[11] `https://www.ibm.com/cloud/websphere-application-platform`
[12] `https://developers.redhat.com/products/datavirt/overview/`

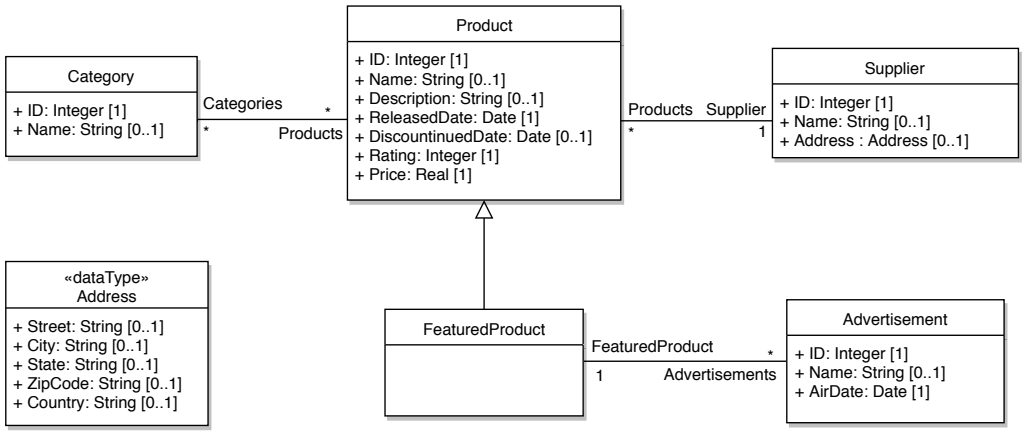


Figure 2.1: Class diagram of an online store.

To illustrate the OData protocol, we use as running example the UML class diagram shown in Figure 2.1 representing a data model to manage an online store. This example is inspired by the official reference example released by the OData community to illustrate the OData protocol. The model includes the classes: `Product` to represent products; `Supplier` to represent the supplier of a product; `Category` to classify products; `FeaturedProduct` for premium products to be featured in commercials; and `Advertisement` which records the data about these commercials. The address of a supplier is defined using the data type `Address`.

OData protocol supports the description of data models and the editing and querying of data according to these models. Data models are described using an Entity Data Model (EDM) [PHZ14c] which borrows some concepts from the ER model and defines an abstract conceptual model of the data exposed by an OData service. An *OData Metadata Document* is a representation of this data model exposed for client consumption. This document can be retrieved by appending `$metadata` to the root URL of the host of the service.

Listing 2.3 shows an except of the metadata document of the running example in Common Schema Definition Language (CSDL) XML format [PHZ14b]. The `edmx:Edmx` element is the root element of the document and includes the version of the OData protocol (i.e., `version` attribute) and an `edmx:DataServices` element. The `edmx:DataServices` element contains a schema (i.e., `Schema` element) which defines the data model exposed by the service.

The schema includes attributes to define a namespace (e.g., `com.example.ODataDemo`) and an alias (e.g., `ODataDemo`), and contains the

Listing 2.3: A simple OData Metadata Document for the products service

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx"
    Version="4.0">
  <edmx:DataServices>
    <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm"
        Namespace="com.example.ODataDemo" Alias="ODataDemo">
      <EntityType Name="Product">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Description" Type="Edm.String"/>
        <Property Name="ReleasedDate" Type="Edm.DateTimeOffset"
            Nullable="false"/>
        <Property Name="DiscontinuedDate" Type="Edm.DateTimeOffset"
            />
        <Property Name="Rating" Type="Edm.Int16" Nullable="false"/>
        <Property Name="Price" Type="Edm.Double" Nullable="false"/>
        <NavigationProperty Name="Supplier" Type="ODataDemo.
            Supplier" Partner="Products"/>
        <NavigationProperty Name="Categories" Type="Collection(
            ODataDemo.Cotegory)" Partner="Products"/>
      </EntityType>
      <EntityType Name="Category">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <NavigationProperty Name="Products" Type="Collection(
            ODataDemo.Product)" Partner="Categories"/>
      </EntityType>
      <EntityType Name="FeaturedProduct" baseType="Product">...</
          EntityType>
      <EntityType Name="Advertisement">...</EntityType>
      <EntityType Name="Supplier">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Address" Type="ODataDemo.Address"/>
        <NavigationProperty Name="Products" Type="Collection(
            ODataDemo.Product)" Partner="Supplier" />
      </EntityType>
      <ComplexType Name="Address">
      <Property Name="Street" Type="Edm.String"/>...
      </ComplexType>
      <EntityContainer Name="ODataDemoService">
        <EntitySet Name="Products" EntityType="ODataDemo.Product">
          <NavigationPropertyBinding Path="Categories" Target="
              Categories"/>
          <NavigationPropertyBinding Path="Supplier" Target="
              Suppliers"/>
        </EntitySet>
        ...
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

data model elements defined by the service. These data model elements are organized into entity types (i.e., `EntityType` elements) and complex types (i.e., `ComplexType`). On the one hand, entity types are named structured types with a key. They define the properties and relationships of an entity and may derive by inheritance from other entity types. On the other hand, complex types are keyless named structured types consisting of a set of properties. The schema in the example includes the entity types `Product`, `Category`, `FeaturedProduct`, and `Supplier`; and the complex type `Address`. Each entity type includes a name (i.e., `name` attribute) and may specify a base type (i.e., `BaseType` attribute) if such entity type inherits from another entity type (e.g., the entity type `FeaturedProduct` inherits from the entity type `Product`). It also contains an identifier defined using the `Key` element, a set of properties (i.e., `Property` elements), and one or more relationships (if any) (i.e.; `NavigationProperty` element). The `Product` entity type, for instance, includes the properties `ID`, `Name`, `Description`, `ReleasedData`, `DiscountinuedDate`, `Rating`, and `Price`; and the navigation properties `Supplier` and `Categories`.

A property includes a name (i.e., `Name` attribute) and a type (i.e., `Type` attribute), and may contain facet attributes (e.g., `Nullable` attribute). The type of a property is either primitive[13] (e.g., `Edm.String`), complex (e.g., `ODataDemo.Address`), enumeration (the example does include an enumeration), or a collection of one of these types (e.g., `Collection(Edm.String)`). A property may contain type facets which modify or constrain its valid values. For instance, the `ID` property of the `Product` entity type includes the type facet `Nullable` which specifies that a value is required. OData defines other types facets such as default value, max value, and max length.

A navigation property allows the navigation to related entities. It contains a name (i.e., `Name` attribute), a type (i.e., `Type` attribute), and optionally includes extra attributes (e.g., `Partner`). The type of a navigation property is an entity type within the schema (e.g., `ODataDemo.Supplier`) or a collection of entity types (e.g., `Collection(ODataDemo.Categories)`). It may also contain other optional attributes, namely: `Nullable` to specify that the declaring entity type may have no related entity, `Partner` to define the opposite navigation property for bidirectional relationships (e.g., the navigation property `Supplier` of the entity type `Product` defines the `Products` as the opposite navigation property), and `containsTargets` to declare the

[13]The complete list of the supported primitive types by OData is available at `http://docs.oasis-open.org/odata/odata-csdl-xml/v4.01/cs01/odata-csdl-xml-v4.01-cs01.html#sec_PrimitiveTypes`

relationship as containment.

The schema also includes an entity container (i.e., `EntityContainer` element) which defines the entity sets (i.e., `EntitySet` element), singletons (i.e., `Singleton` element), function and action imports (i.e., `FunctionImport` and `ActionImport`, respectively) exposed by the service. An entity set gives access to a collection of entity type instances (e.g., the entity set `Products` allows access to `Product` entity type instances). If an entity type declares navigation properties, the corresponding entity set includes navigation property bindings (i.e., `NavigationPropertyBinding` element) describing which entity set will contain the related entity (i.e., `Target` attribute). A singleton, on the other hand, allows addressing a single entity, while a function import and an action import is used to expose a function and an action, respectively.

The OData specification defines standard rules to query data via HTTP `GET` requests and perform data modification actions via HTTP `POST`, `PUT`, `PATCH`, and `DELETE` requests. OData defines a set of recommended rules for constructing URLs to identify the data and metadata exposed by an OData service, and a set of URL query string operators. A URL of an OData request has three parts [PHZ14d]: (1) the service root URL, which identifies the root of an OData service; (2) a target resource path, which identifies a resource to query or update (e.g., products, a single product, supplier of a product); and (3) a set of query options.

$$\underbrace{\texttt{GET}}_{\text{HTTP method}} \quad \underbrace{\texttt{http://example.com/service}}_{\text{service root URL}} \texttt{/} \underbrace{\texttt{Products(1)/Categories}}_{\text{target resource path}} \texttt{?} \underbrace{\texttt{\$top=2\&orderby=Name}}_{\text{query options}} \tag{2.1}$$

The Expression 2.1 shows an example of an OData request to retrieve the first two records of the categories of the product having the `ID` 1, ordered by name. This request employs the query operators `$top` and `$orderby`. OData defines other query options, for instance, to filter data. Table 2.1 shows some examples of OData queries using the query options defined by OData.

OData defines representations for the OData requests and response using JSON format and XML atom format. Listing 2.4 shows the request 2.1 and its response in JSON format using `CURL`. As can be seen, the `value` array includes the first two categories of the product having the `ID` 1, ordered by name.

Table 2.1: OData query options examples.

OPTION	DEFINITION	QUERY EXAMPLE	DESCRIPTION
`$filter`	Filter a collection of resources that are addressable by a request URL	`http://host/service/Products?$filter=Name eq 'Milk'`	The products with name equals to *Milk*
`$expand`	Include relative resource in line with retrieved resources	`http://host/service/Suppliers(1)?$expand=Products`	Supplier with ID 1 and his products
`$select`	Request a specific set of properties	`http://host/service/Products?$select=Name,Price`	The name and price of the products in the collection
`$count`	Request a count of the resources	`http://host/service/Product?$count`	The number of products in the collection
`$search`	Request entities matching a free text search	`http://host/service/Product?$search='Milk'`	The products with a text field equals to *Milk*

Listing 2.4: An example of a request to the online store

```
curl -H Accept:application/json -i https://services.odata.org/V4/
    OData/OData.svc/Products(1)/Categories?$top=2&$orderby=Name
HTTP/1.1 200 OK
...
{
    "@odata.context": "https://services.odata.org/V4/OData/OData.
        svc/$metadata#Categories",
    "value": [
        {
            "ID": 1,
            "Name": "Beverages"
        },
        {
            "ID": 0,
            "Name": "Food"
        }
    ]
}
```

2.3 MODEL-DRIVEN ENGINEERING

The Model-Driven Engineering paradigm emphasizes the use of models to raise the level of abstraction and to automate the development of software. Abstraction is a primary technique to cope with complexity, whereas automation is the most effective method for boosting productivity and qual-

ity [Sel03]. We apply MDE techniques in this thesis to the context of REST APIs. This section gives a brief introduction of MDE and its technologies.

2.3.1 *History*

Since the early ages of computing, software developers have been using abstractions to help them program in their design intent instead of the underlying computing environments, thus hiding the complexity of such environments. These abstractions included both language and platform technologies. For instance, early programming languages, such as assembly languages and FORTRAN, allowed developers to hide the complexity of programming directly with machine code. Similarly, early operating systems, such as OS/360 and Unix, allowed developers to hide the complexity of programming directly with hardware devices [Sch06].

The emergence of Computer Aided Software Engineering (CASE) technology in the 80s was considered to be the first move toward the support of MDE. These tools allowed developers to express their design intents in terms of graphical notations such as structural diagrams, whilst also generating software artifacts. However, CASE technology did not gain a wide adoption due to the lack of standarizations and its limited capability to express system designs using graphical representations as well as the transformation of such representations to software artifacts.

Languages and platforms have evolved during the last years to provide high abstraction levels for software development, thus making the development of increasingly complex systems possible. For instance, nowadays developers typically rely on high level object-oriented languages, such as C++, Java or C# instead of low level programing languages such as FORTRAN or C. However, the evolution and maintenance of software systems has become a hard and time consuming task in such evolving field.

MDE is a methodology that applied the lessons learned from early attempts to build CASE tools. MDE aims at organizing software and system artifacts in different abstraction levels, advocating the use of models as the first class artifacts throughout the entire system development cycle. A model consists of a set of elements that provide an abstract description of a system from a particular perspective. In an MDE approach, a system is initially described using one or more models which capture its requirements regardless of the target platform or implementation technology. A series of refinements and transformations of models are then performed to decrease the abstraction level in each step. The final step is a transformation which

generates code from the most refined model.

Next section describes Model-Driven Architecture (MDA), a methodology that supports MDE and provides a set of standards to describe models and transformations.

2.3.2 *Model-Driven Architecture*

The Object Management Group (OMG)[14] launched the MDA initiative [OMG14a] as a guideline for structuring software specifications that are expressed as models. MDA places models at the heart of the software development process and separates business and application logic from the underlying platform technology. In such approach, Platform-Independent Models (PIMs) which represent a system from the platform independent viewpoint are expressed using an associated OMG modeling standard such as UML [OMG17]. These PIMs are later transformed to Platform-Specific Models (PSMs) which are adapted to a target platform of language. Finally, these PSMs need to be eventually transformed into software artifacts such as code, deployment specifications, reports, documents, etc. The MDA approach proposes also to automate model transformation, thus resulting in software development tasks to be focused on modeling instead of coding.

MDA relies on several OMG standards to achieve its goal. Some of these standards are the following:

META-OBJECT FACILITY (MOF) is the meta-metamodel used to describe metamodels in the MDA approach, and therefore new vocabularies [OMG16b]. It is used for instance to create the UML metamodel.

UML is a family of modeling notations unified by a common metamodel covering multiple aspects of business and systems modeling [OMG17]. UML will be introduced in Section 2.3.3.

UML PROFILES provide an extension mechanism of UML language to specific requirements in a unified tooling environment [OMG17]. UML profiles will be introduced in Section 2.3.4.

OCL is a declarative language which expresses queries and describes constraints over UML and other MOF-based metamodels and models [OMG14b]. It will be introduced in Section 2.3.5.

[14] `https://www.omg.org/`

QUERY/VIEW/TRANSFORMATION (QVT) is a standard to describe model transformations and equivalence relationships among MOF-based models [OMG16a]. It uses OCL to express queries over the candidate models.

XML METADATA INTERCHANGE (XMI) defines an XML-based interchange format for UML and other MOF-based metamodels and models [OMG15].

2.3.3 *UML*

UML [OMG17] defines a graphical language for visualizing, specifying, constructing, and documenting system artifacts. The UML specification describes a common MOF-based metamodel that specifies the abstract syntax of UML and its semantics. It also defines a specification of the human-readable notation for representing the individual UML modeling concepts as well as rules for combining them into a variety of different diagram types corresponding to different aspects of system modeling.

UML defines 14 different diagrams, namely: *Activity Diagram*, *Class Diagram*, *Communication Diagram*, *Component Diagram*, *Composite Structure Diagram*, *Deployment Diagram*, *Interaction Overview Diagram*, *Object Diagram*, *Package Diagram*, *Profile Diagram*, *State Machine Diagram*, *Sequence Diagram*, *Timing Diagram*, *Use Case Diagram*. In practice, the mostly used diagrams in software development are: *Use Case Diagrams*, *Class diagrams*, and *Sequence diagrams*. In this thesis, we will be using *Class diagrams*, *Object diagrams*, and *Profile diagrams* for the specification and illustration of our contributions. Figure 2.1 showed an example of a Class diagram for managing an online store.

2.3.4 *UML Profiles*

To define domain-specific knowledge, we can either create a new metamodel from scratch or extend an existing modeling language such as UML. UML profiles provide a fast way to describe a new language and allow the reuse of the tooling infrastructure of UML. In fact, UML profiles tailor the UML language to a particular domain (e.g., Service Oriented Architecture) or platform (e.g., JEE, .Net).

A UML Profile is defined using a *Profile Diagram* which includes custom stereotypes, tag definitions, and constraints that are applied to specific UML elements (e.g., `Class`, `Operation`) in order to enrich such elements with custom vocabulary needed in a particular domain.

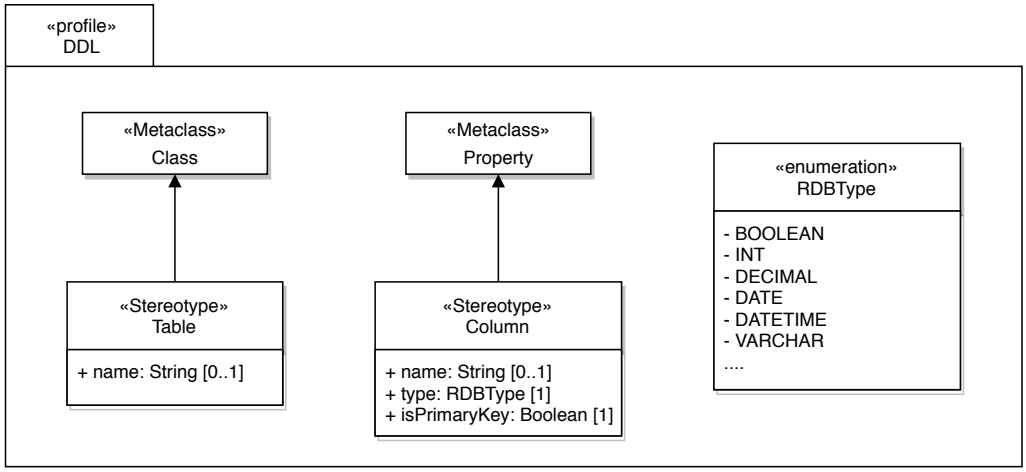


Figure 2.2: A simple UML profile for relational databases.

Figure 2.2 shows a simple UML profile to extend Class diagram vocabulary to support relational databases. As can be seen, this profile defines the stereotypes `Table` and `Column` which extend the metaclasses `Class` and `Property`, respectively. The `Table` stereotype includes the property `name`, which defines the name of the table corresponding to a Class element. The `Column` stereotype includes the properties `name`, `type` and `isPrimaryKey` to define a column name, its type, and whether such column is a primary key, respectively. Figure 2.3 shows an example of this profile application on the Class `Product` of the online store example showed in Figure 2.1. As can be seen, the the Class `Product` has the stereotype `Table` which defines the name of the table associated with this Class (i.e., `product`). Each class attribute has the stereotype `Column` which provides the details of the corresponding column in the table. For instance, the stereotype `Column` of the attribute `ID` defines the column name (i.e., `id`) and its type (i.e., `INT`), and specifies that this column is a primary key (i.e., `isPrimaryKey=true`). This profile could be used for example to generate a Data Definition Language (DDL) script to create the database.

2.3.5 *OCL*

A UML diagram, such as a class diagram, does not typically provide all the relevant aspects of a specification. While UML profiles provide an extension mechanism to tailor UML models to a specific requirement, there is still a need to describe additional constraints about the elements in the model. OCL has been created to fill this gap by offering a standard language

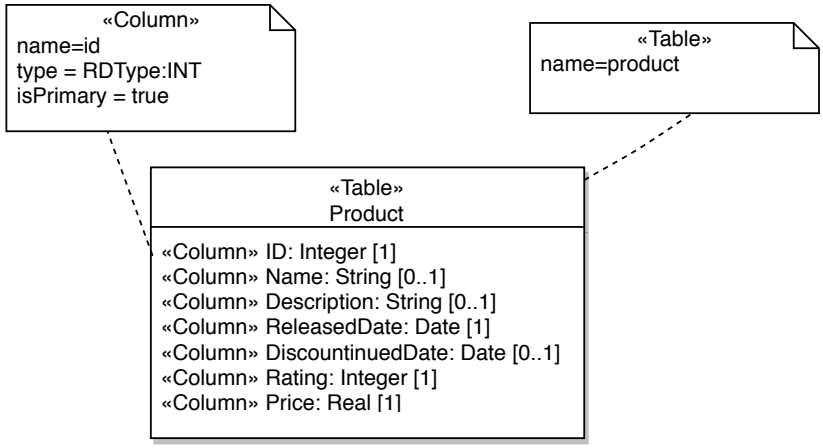


Figure 2.3: A example of a UML Class diagram with a profile.

to define expressions on model elements. OCL is a general-purpose formal language which can be used to define several kinds of expressions complementing UML models or any kind of MOF-based models. The language is typed (i.e., each expression is conform to a type), declarative (i.e., expressions define the desired operation not how to perform them), and side-effect free (i.e., an expression does not change the state of a system).

OCL can be used for different purposes such as specifying invariant conditions that must hold for the modeled system, creating queries over model elements, specifying type invariants for stereotypes, and describing guards for model transformation. Listing 2.5 shows an example of an OCL expression specifying an invariant on the Class `Product` of the online store example showed in Figure 2.1. The expression `validDiscountinuedDate` specifies that the value of the attribute `DiscountinedDate` of the Class `Product` (if exists) should be higher that the value of the attribute `ReleasedDate` of the same Class.

Listing 2.5: Simple OCL example.

```
1 context Class
2         ivr discountinuedConst : self.DiscountinuedDate.
              oclIsUndefined().not() implies
3                         self.DiscountinuedDate > self.ReleasedDate
```

2.3.6 *Supporting Frameworks for MDE*

The Eclipse platform[15] provides a wide ecosystem of projects and tools to support MDE and its associated standards (e.g., MDA, MOF, UML). This section presents the most relevant projects and tools that provide support to the different standards that are used in the implementation of the contributions of this thesis.

Eclipse Modeling Framework. The EMF provides modeling, metamodeling, and code generation capabilities within the Eclipse platform. EMF uses *Ecore* to describe metamodels which can be considered as an implementation of the EMOF specification proposed by OMG[16]. EMF relies on the OMG standard XMI to persist *Ecore* models. It also provides a generative solution which constructs Java APIs out *Ecore* models to facilitate their management, thus promoting the development domain-specific applications.

Model Development Tools. The Model Development Tools (MDT) project is an Eclipse project which aims at providing an EMF-based implementation of industry standards models, most of which are OMG specifications, and tools for developing models based on those metamodels. Some of the metamodels and tools which are used in the implementation of this thesis are:

OCL, which provides an implementation of OCL for EMF-based models and metamodels.

UML2, which implements the UML metamodel using *Ecore* and provides a Java API to manipulate UML models.

PAPYRUS, which provides diagramming tool facilities for many OMG standards such as UML, SysML, and MARTE. In particular, it provides diagram editors for EMF-based modeling languages and offers an advanced support for the definition of UML profiles. In this thesis, we rely on PAPYRUS for the creation of UML diagrams and UML profiles.

Model Transformations. There are different ways to define model transformations within the Eclipse platform for EMF-based models. For instance,

[15] `https://www.eclipse.org/`

[16] OMG has defined two compliance points for MOF: EMOF for Essential MOF and CMOF for Complete MOF.

model transformations could be expressed in Java using the generated Java API from EMF models. They could also be expressed using a dedicated Domain Specific Language (DSL) such the ATL Transformation Language (ATL) language [Jou+08]. ATL is a Model-to-Model (M2M) transformation language and a toolkit created initially by the ATLANMOD team (initially ATLAS GROUP) but is maintained actually by OBEO[17]. An ATL transformation is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. ATL provides a hybrid language (imperative/declarative) to perform *Ecore* model transformations.

Code Generation. As explained before, EMF provides a facility to generate Java APIs from *Ecore* models. The EMF code generation engine uses a language called JET[18], which relies on JSP-like template files, to generate Java code. JET is typically used to customize the generated Java APIs from *Ecore* models, as we will see in Chapter 4. However, JET is not used to define new code generators due to its complexity and lack of tooling supporting it. In this thesis, we will rely on ACCELEO[19] language, which provides an implementation of the OMG Model-to-Text (M2M) standard, to create new code generators for our approaches.

2.4 SUMMARY

This chapter shed some light on the technological background used as the foundations of this thesis. We presented first REST architectural styles and the two specifications we focus on our contributions, namely: the OpenAPI specification and OData protocol. Later, we presented MDE and the technologies used for the implementation of the contributions of this thesis.

[17] https://www.obeo.fr/en/
[18] https://www.eclipse.org/modeling/m2t/?project=jet
[19] https://www.eclipse.org/acceleo/

CHAPTER

3 Modeling REST APIs

This chapter presents the different model-based representations we created for modeling REST APIs. In particular, we created a metamodel and a UML profile for OpenAPI and OData. On one hand, metamodels give more freedom to create a syntax closer to OpenAPI and OData domains, and therefore ease reverse/forward engineering. On the other hand, profiles rely on the well-known UML standard and allow the reuse of its tooling infrastructure to define OpenAPI and OData editors. These artifacts will become the foundations of our approach and help us integrate REST APIs into our model-driven approach as we will present in the next chapters.

This chapter is organized as follows. Section 3.1 is dedicated to the OpenAPI specification and presents (i) the OpenAPI metamodel, (ii) a UML profile for OpenAPI, and (iii) tool support. Similarly, Section 3.2 is dedicated to OData protocol and presents (i) a metamodel to describe OData, (ii) a UML profile we developed for OData, and (iii) tool support.

3.1 Modeling OpenAPI

This section describes the different modeling artifacts we created for the OpenAPI specification. These artifacts will facilitate the integration of OpenAPI into model-driven development techniques such as transformations and code generation.

We will start this section by presenting the OpenAPI metamodel. Later, we will describe the UML profile for OpenAPI. Finally, we will present the

tools we created to support theses representations.

3.1.1 *A Metamodel for OpenAPI*

The OpenAPI metamodel is derived from the concepts and properties described in the OpenAPI specification document[1]. In the following, we will explain its main parts, namely: (i) behavioral elements, (ii) structural elements, (iii) metadata elements, (iv) serialization/deserialization elements, and finally (v) security elements.

Behavioral Elements. Figure 3.1 shows the behavioral elements of the OpenAPI metamodel. An API definition is represented by the `API` element, which is the root element of our metamodel. This element includes attributes to specify the host serving the API, the base path of the API, the transfer protocol/s supported by the API (i.e., `schemes` attribute) and the list of MIME types the operations of the API can consume/produce (i.e., `consumes` and `produces` attributes, respectively). The transfer protocols supported by the OpenAPI specification are `http`, `https`, `ws`, and `wss`. The MIME type definitions should comply with RFC 6838[2] (e.g., `application/json` for JSON content). The API definition also includes references to the available paths, the data types used by the operations (i.e., `definitions` reference) and the possible responses of the API calls.

The `Path` element contains a relative path to an individual endpoint and the operations for the HTTP methods (e.g., `get` and `put` references). The description of an operation (i.e., `Operation` element) includes an identifier `operationId`, the MIME types the operation can consume/produce, and the supported transfer protocols for the operation (i.e., `schemes` attribute). The MIME types and schemes declared at this level will override the ones declared at the API level. An operation includes also the possible responses returned from executing the operation (i.e., `responses` reference).

`API`, `Path` and `Operation` elements inherit from `ParameterContext`, which allow them to define parameters at API level (applicable for all API operations), path level (applicable for all the operations under this path) or operation level (applicable only for this operation).

The `Response` element defines the possible responses of an operation and includes the HTTP response code, a description, the list of headers sent with the response, and optionally an example of the response message.

[1] `https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md`

[2] `http://tools.ietf.org/html/rfc6838`

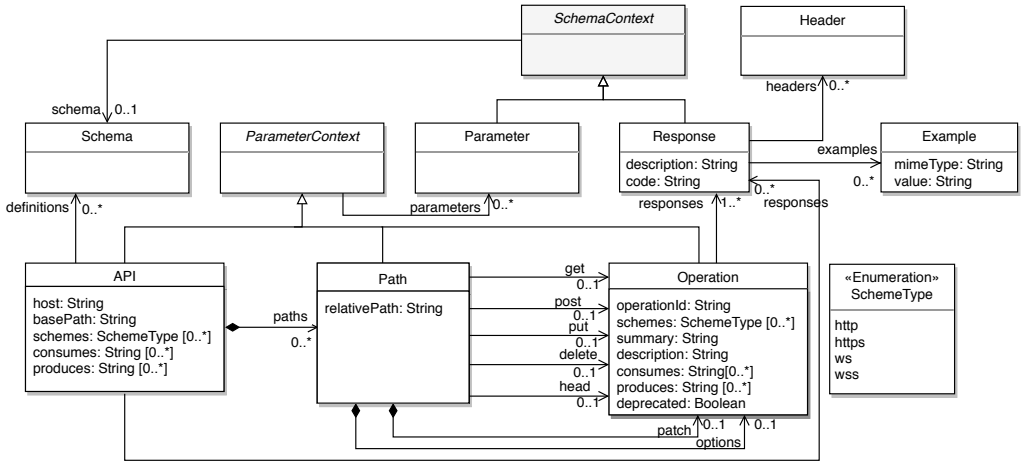


Figure 3.1: The OpenAPI metamodel: behavioral elements

`Response` and `Parameters` elements inherit from `SchemaContext` thus allowing them to add the definition of the response structure and the data type (`schema` reference) used for the parameter, respectively. `Parameter` and `Schema` elements will be explained while presenting the structural elements.

Structural Elements. Figure 3.2 shows the structural elements used in a REST API, namely: the `Schema` element, which describes the data types; the `Parameter` element, which defines the parameter of an operation; the `ItemsDefinition` element, which describes the type of items in an array; and the `Header` element, which describes a header sent as part of a response. These elements use an adapted subset of the JSON Schema Specification defined in the super class `JSONSchemaSubset`[3].

A parameter definition includes a name and two boolean properties to specify whether the parameter is required and if empty values are accepted[4].

The location of the parameter is defined by the `location` attribute. The possible locations are:

- `path`, when it is part of the URL (e.g., `petId` in `/pet/{petId}`),
- `query`, when it is appended to the URL (e.g., `status` in `/pet/findByStatus?status="sold"`),

[3]More information about the schema information can be found at `http://json-schema.org/latest/json-schema-validation.html`

[4]Required parameters are parameters which should be included in the request, but they may only have a name and no value. When setting `allowEmptyValues` to `false`, required parameters should include a value as well.

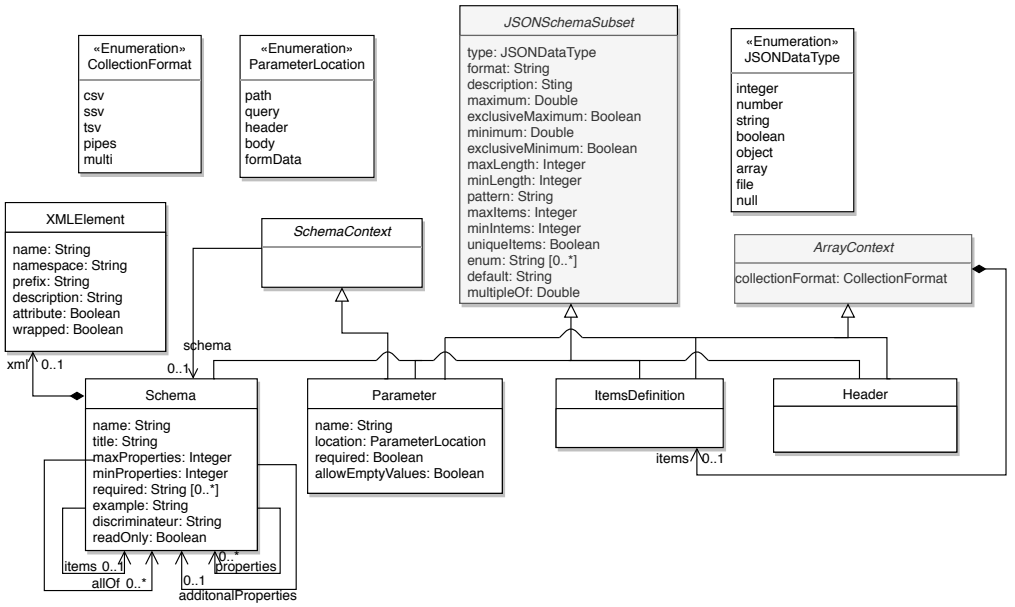


Figure 3.2: The OpenAPI metamodel: structural elements

- ◇ `header`, for custom headers,
- ◇ `body`, when it is in the request payload,
- ◇ `formData`, for specific payloads[5].

`Parameter` and `Header` elements inherit from `ArrayContext` to allow them to specify the collection format and the items definition for attributes of type `array`. Additionally, the `Parameter` element inherits from `SchemaContext` to define the data structure when the attribute `location` is of type `body` (i.e., `Schema` reference).

The `Schema` element defines the data types that can be consumed and produced by operations. It includes a name, a title, and an example. Inheritance and polymorphism is specified by using the `allOf` reference and the `discriminator` attribute, respectively. Furthermore, when the schema is of type `array`, the `items` reference makes possible to specify the values of the array.

Metadata Elements. Figure 3.3 describes the metadata and documentation elements of the OpenAPI metamodel. As can be seen, an API definition includes the description of the global information about the API (i.e., `info`

[5] `application/x-www-form-urlencoded` or `multipart/form-data`

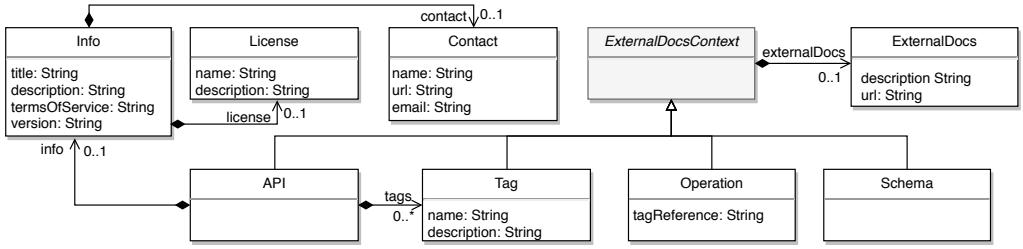


Figure 3.3: The OpenAPI metamodel: metadata and documentation elements.

reference). The description (i.e., `Info` element) includes: the title of the API, a short description, the terms of service to use the API, the API version, and a list of tags (`tags` reference). It also includes references to represent contact and license information (`contact` and `license` references, respectively). On the one hand, a contact description includes a name, a URL, and an email address. On the other hand, a license description includes the name of the license and is URL.

Additionally, `API`, `Tag`, `Operation`, and `Schema` elements inherit from the `ExternalDocsContext` which allows the definition of external documentation (`externalDocs` reference).

Security Support. Figure 3.4 shows the security elements of the OpenAPI metamodel. As can be seen, an API definition may include a set of security definitions (i.e., `securityDefinitions` reference). The `SecurityScheme` element allows the definition of a security scheme that can be used and includes a name to identify the scheme; a description; a type, which can be `basic` for basic authentication, `apiKey`, or `oauth2` for OAuth2 common flows; and a location, which can be either `query` (i.e., as a query parameter in the URL of the request) or `header` (i.e., as part of the header of the request). For schemes of type `oauth2`, the definition also includes the flow type, the authorization and the token URLs to be used for the flow. Additionally, OAuth2 security schemes include the available scopes (`scopes` reference). The `SecurityScope` element defines a scope of an OAuth2 security scheme and includes a name and a short description.

The `API` and `Operation` elements inherit from the `SecurityContext` element, thus allowing the specification of security schemes that can be used to authenticate the requests (i.e., `securityRequirements` reference). While the security schemes declared in the API level are applied for the API as a whole, the ones declared in an operation level are only applicable

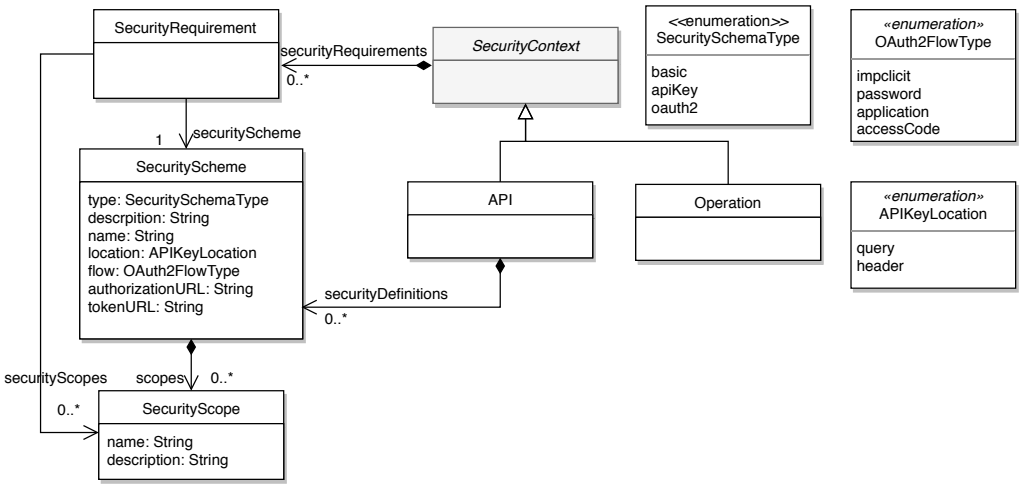


Figure 3.4: The OpenAPI metamodel: security elements.

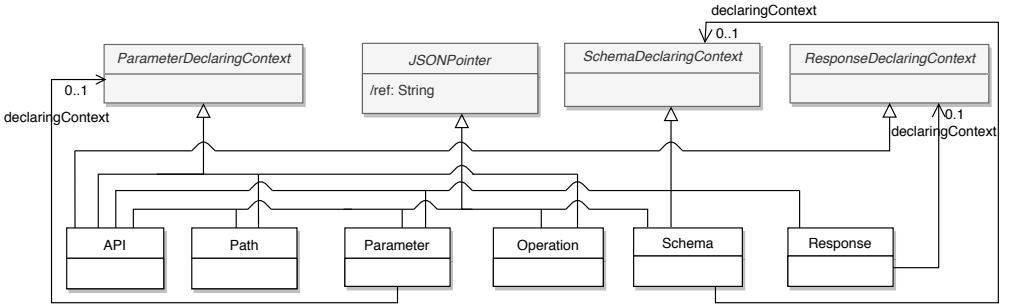


Figure 3.5: The OpenAPI metamodel: serialization/deserialization elements.

for such operation. The `SecurityRequirement` element allows the declaration of a security requirement and includes the targeted security scheme (i.e., `securityScheme` reference) and, if applicable, the required scope (i.e., `securityScopes` reference).

Serialization/Deserialization Support. Figure 3.5 shows the elements of the metamodel to support serialization and deserialization of OpenAPI models in JSON (or YAML) format. As explained before, a parameter can be defined at the API level, path level, or operation level. To specify this, `API`, `Path`, and `Operation` elements inherit from the `ParameterDeclaringContext` element which is referenced in each parameter (i.e., `declaringContext` reference). A similar strategy is followed by the `Schema` element (the schema can be declared in the API level, parameter level, response level, or inside a schema) and the `Response` element (a

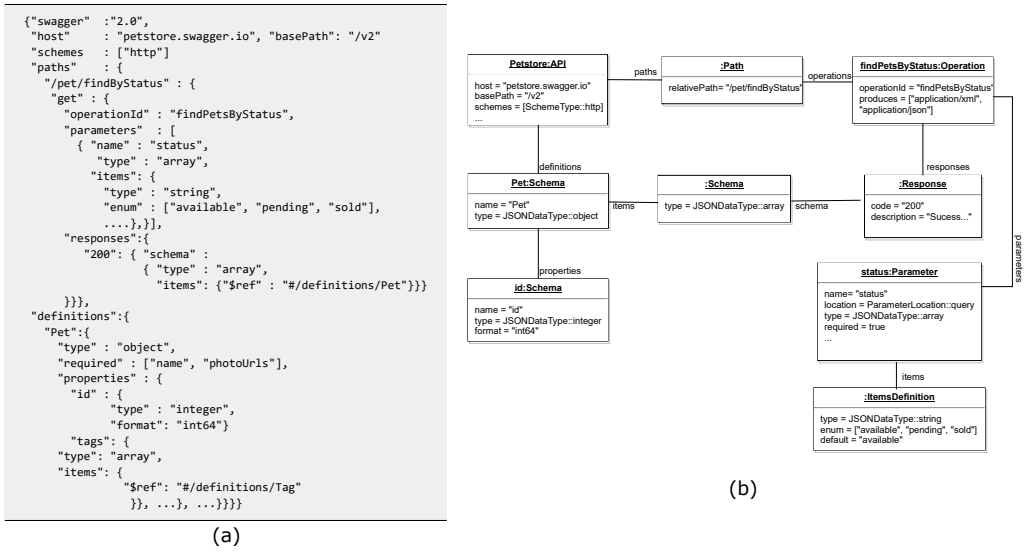


Figure 3.6: OpenAPI model example: (a) an excerpt of the Petstore OpenAPI definition and (b) an except of the corresponding OpenAPI model.

response can declared at the API level or operation level).

All behavioral elements inherit from the `JSONPointer` element which defines a JSON reference for each element. This element includes a derived attribute called `ref` which is dynamically calculated depending on its declaring context. This attribute specifies the path of the element within a JSON document following the RCF 6901[6], which can be used to reference a JSON object within the JSON document.

Figure 3.6 shows an excerpt of the OpenAPI definition the Petstore API presented in Chapter 2 (see Section 2.2) and the corresponding OpenAPI model.

3.1.2 *A UML Profile for OpenAPI*

This section presents the UML profile we created to extend UML class diagrams in order to support OpenAPI definitions. We will explain first how we map OpenAPI elements and UML concepts. Later, we will present the stereotypes we created to manage OpenAPI definitions.

3.1.2.1 Mapping UML and OpenAPI

We opted to use UML class diagrams as the basis of our OpenAPI profile

[6] `https://tools.ietf.org/html/rfc6901`

Table 3.1: Mapping OpenAPI and UML elements.

OpenAPI element	Condition	UML element	Details
Schema	A Schema definition of object	Class	
	A Schema property of type primitive or array of primitives	Property	A class property
	A Schema property of type object or array of objects	Property	Association end
Operation	-	Operation	
Parameter	-	Parameter	direction = in
Response	-	Parameter	direction = return

given the semantic similarities between class diagrams and the OpenAPI specification. Particularly, we rely on UML class diagrams to represent the data model (i.e., schema definitions) and operations of OpenAPI definitions.

Table 3.1 shows how we map OpenAPI elements to UML concepts. Columns one and two show an OpenAPI element and a specific condition to meet (if any), respectively. Columns three and four show the corresponding UML element and the initialization details of such element, respectively. As explained before, `Schema` definitions allow the definition of data types in the OpenAPI specification. They are also used to define the structure (i.e., properties and associations) of other `Schema` definitions of type `Object`. Thus, depending of the role of a `Schema` definition, we associate the adequate UML element (i.e., class, attribute, association end) as shown in the table. Furthermore, the `Operation` concept in OpenAPI is mapped to the UML concept `Operation`. Finally, both `Parameter` and `Response` concepts of an API are mapped to the UML concept `Parameter` as mentioned in the table.

UML is used to provide the details of OpenAPI definitions when the corresponding semantic meaning can be represented in a Class diagram. The rules below are applied to represent the supported OpenAPI properties in UML.

- The `required` property is represented by the lower bound cardinality of the corresponding UML element,
- `array` types are represented by the upper bound cardinality of the corresponding UML element,
- `enum` property is represented by `Enumeration` in the Class diagram,

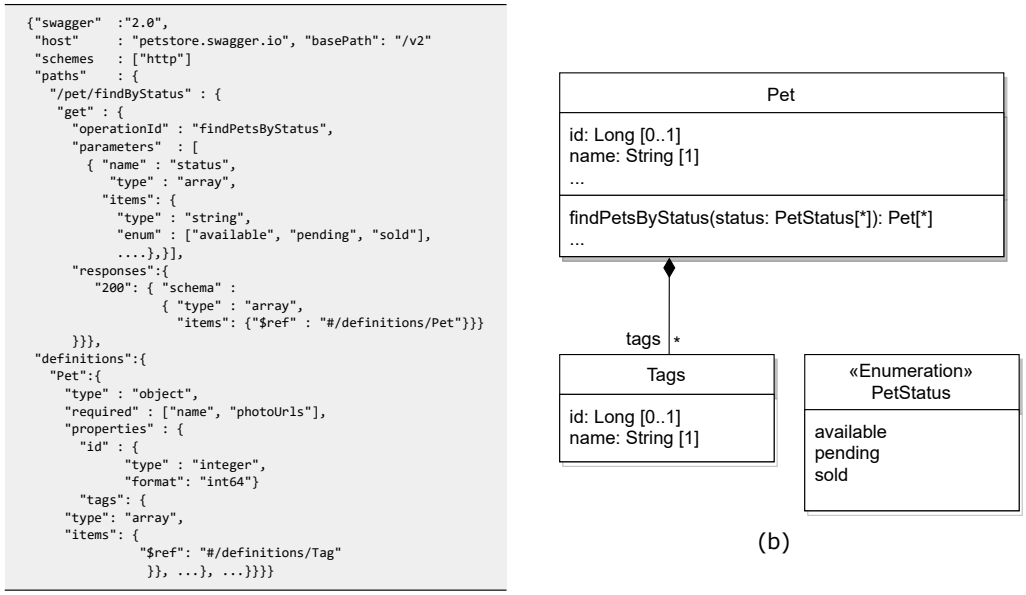


Figure 3.7: OpenAPI model example: (a) an excerpt of the Petstore OpenAPI definition and (b) the corresponding UML model.

- ◇ The `default` property is represented by the `default` value of the corresponding UML element,
- ◇ The `discriminator` and `allOf` properties (used to define hierarchy in OpenAPI) are represented by inheritance in a Class diagram,
- ◇ The `operationId` property is represented by the name of the corresponding UML operation,
- ◇ The `name` property is represented by the name of the corresponding UML element,
- ◇ The `readOnly` property is represented by the `isReadOnly` property of the corresponding UML element.

Figure 3.7.b shows the Class diagram corresponding to the excerpt of the Petstore definition shown in Figure 3.7.a. As can be seen, the schema definition `Pet` and its properties are represented by the Class `Pet` which includes the corresponding attributes following the rules we described before. The API operation `findPetsByStatus` is represented by the corresponding class operation which includes the required parameters. The class diagram also

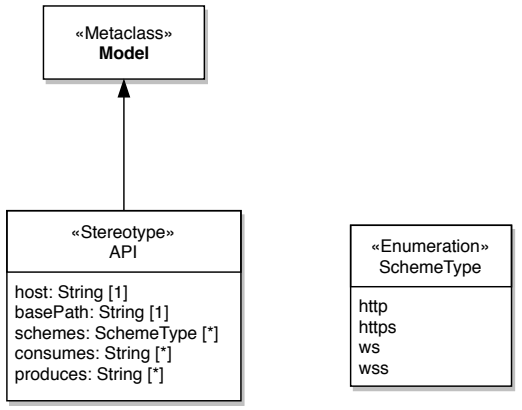


Figure 3.8: OpenAPI profile: the API element.

includes the Class `Tag`, to represent the schema `Tag`; and the enumeration `PetStatus`, to represent the `enum` property of the parameter `status`. The property `tags` of the schema `Pet` is represented by an association between the classes `Pet` and `Tag`. Next we will present the profile we created to extend the definition of class diagrams in order to complete the definition of the OpenAPI specification.

3.1.2.2 The OpenAPI Profile

This section describes the UML profile we defined for OpenAPI. This profile includes a set of stereotypes, properties and data types which enable modeling OpenAPI definitions using UML class diagrams. We present next the main parts of the OpenAPI profile, namely: (i) the API element, (ii) the structural elements, (iii) the behavioral elements, (iv) the metadata elements, and finally (v) the security elements.

The API Element. The root of an OpenAPI definition includes the global details of an API such as host, base type, and the supported transfer protocol of the API. We consider that an API definition is represented by the element `Model` of a UML class diagram. Thus, `API` stereotype extends the metaclass `Model` as shown in Figure 3.8. This stereotype defines similar properties to those included in the OpenAPI metamodel. It includes a host, a base type, the transfer protocols the API supports (i.e., `schemes` property), the MIME types the API consumes and produces (i.e., `consumes` and `produces` properties, respectively). The possible values for the transfer protocol are: `http`, `https`, `ws`, and `wss`. The MIME types consumed and pro-

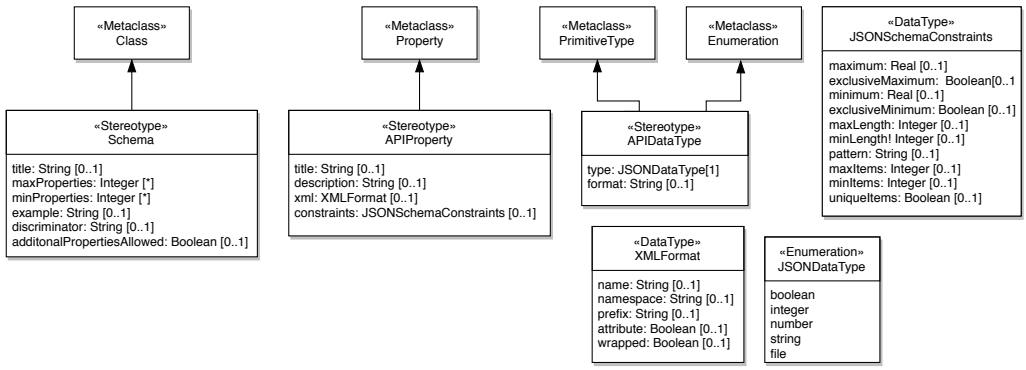


Figure 3.9: OpenAPI profile: structural elements.

duced by the API should comply with RFC 6838[7] (e.g., `application/json`, `application/xml`).

Structural Elements. Figure 3.9 shows the stereotypes related to OpenAPI data types and their mapping with UML concepts. As can be seen, the stereotypes `Schema`, `Property`, and `APIDataType` allow the extension of the UML metamodel to support data types according to the OpenAPI specification.

The stereotype `Schema` extends the metaclass `Class` and allows the definition of schema objects. It includes a title, the maximum/minimum number of properties of the schema (i.e., `maxProperies` and `minProperties` properties, respectively), an example of an instance for this schema, a schema property name to know the concrete type of an instance (i.e., `discriminator` property), and whether additional properties are allowed (i.e., `additionalPropertiesAllowed` property).

The stereotype `APIProperty` extends the metaclass `Property` and allows the definition of a schema property. It includes a title, a description, the metadata to describe the XML representation format of the property (i.e., `xml` property), and a set of constraints related to the property (i.e., `constraints` property). The `XMLFormat` data type allows the definition of the XML representation of the property (e.g., `namespace`, `prefix`), while the `JSONSchemaConstraints` data type describes additional validation constraints for the property (e.g., `maximum`, `minimum`). More information about these constraints can be found in JSON Schema Core[8] and Validation[9].

[7] `https://tools.ietf.org/html/rfc6838`

[8] `https://tools.ietf.org/html/draft-zyp-json-schema-04`

[9] `https://tools.ietf.org/html/draft-fge-json-schema-validation-00`

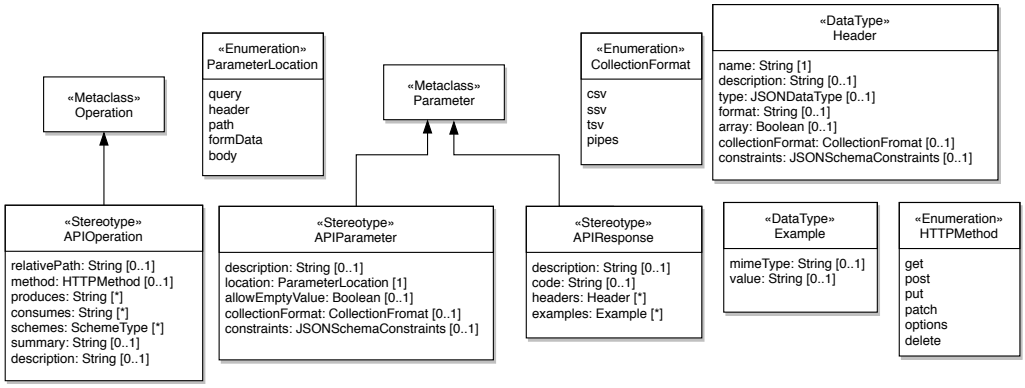


Figure 3.10: OpenAPI profile: behavioral elements.

The stereotype `APIDataType` extends the metaclasses `PrimitiveType` and `Enumeration` to add support for the primitive types defined by the OpenAPI specification. It includes a type and a format such as the possible values for type are `boolean`, `integer`, `number`, `string`, and `file`; while type format is an open string-valued property to define the format of the data type being used (e.g., `email` and `date`). Table 3.2 lists the primitive types defined by the OpenAPI specification. The primitive type `file` is reserved for parameters and responses to set the parameter type or the response as being a file.

Behavioral Elements. Figure 3.10 shows the stereotypes which allow extending the UML class diagram in order to define the behavior of the API (i.e., operations and inputs/outputs). The stereotype `APIOperation` extends the metaclass `Operation` and allows the definition of an API operation. This stereotype includes the relative path of the operation (i.e., `relativePath` property), the HTTP method of the operation, the MIME types the operation

Table 3.2: The primitive data types defined by the OpenAPI specification

COMMON NAME	TYPE	FORMAT
integer	`integer`	`int32`
long	`integer`	`int64`
float	`number`	`float`
double	`number`	`double`
string	`string`	
byte	`string`	`byte`
binary	`string`	`binary`
boolean	`boolean`	
date	`string`	`date`
dateTime	`string`	`date-time`
password	`string`	`password`

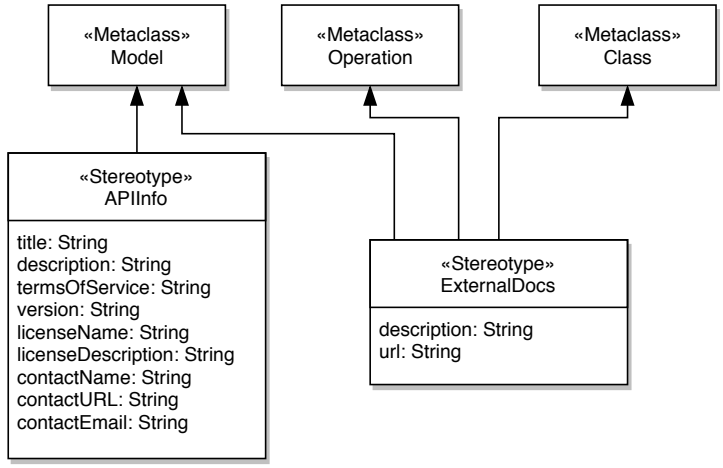


Figure 3.11: OpenAPI profile: metadata Elements.

consumes and produces (i.e., `consumes` and `produces` properties, respectively), the supported transfer protocol schemes (i.e., `schemes` property), a summary, and a description. As can be seen, the definition of the supported MIME type and the transfer protocol can also be declared at the operation level. Similar to the OpenAPI metamodel, this will override their definition at the API level (if any). Note also that the concept *path* is not present in the OpenAPI profile. Thus, the information of the path and the appropriate HTTP method are included in the `APIOperation` stereotype.

The stereotypes `APIParameter` and `APIResponse` extend the the metaclass `Parameter` and allow the definition of an API operation parameter and an operation response, respectively. The stereotype `APIParameter` includes a description, the location of the parameter in the API request (i.e., `location` property), a flag to specify whether empty values are allowed (i.e., `allowEmptyValues` property), the format of the collection if the parameter is multivalued (i.e., `collectionFormat` property), and a set of constraints regarding the values of the parameter (i.e., `constraints` property). The possible values for parameter location are `query`, `header`, `path`, `formData`, and `body`.

The stereotype `APIResponse` defines an API response and includes a description, the HTTP status code related to the response (i.e., `code` property), a list of headers that are sent with the response, and a list of examples of the response message. The datatype `Header` allows the definition of a header sent with the response, while the datatype `Example` allows the definition of an example for a response message. As can be seen, the stereotypes

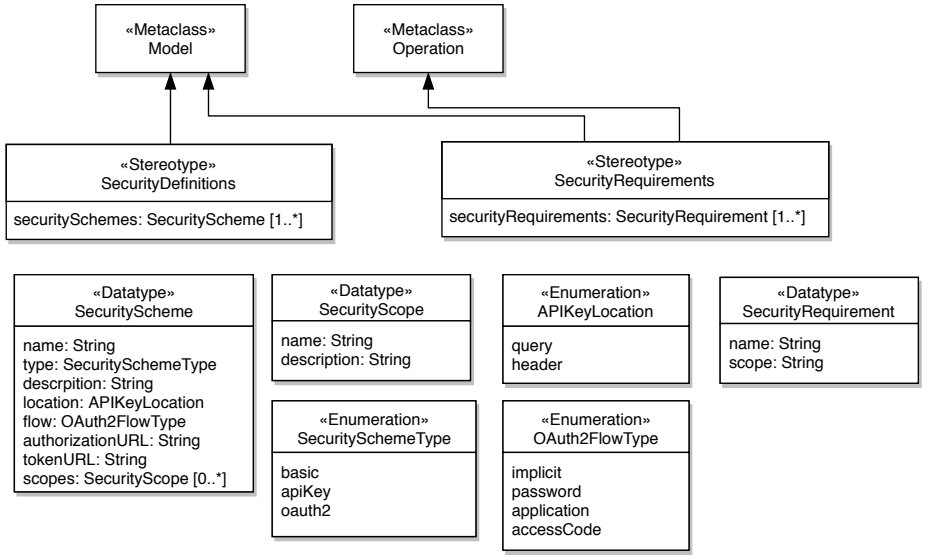


Figure 3.12: OpenAPI profile: security Elements.

`APIParameter` and `APIResponse` do not include information about the data types associated with the parameter or the response. Such information is defined by the type of the UML parameter.

Metadata Elements. Figure 3.11 shows the stereotypes allowing to associate metadata and documentation information to UML model. The stereotype `APIInfo` extends the metaclass `Model` and defines the metadata information of the API. It includes the title of the API, a description, the terms of service, the version of the API, license information (i.e., `licenseName` and `licenseURL`), and contact information (i.e., `contactName`, `contactURL`, and `contactEmail` properties). The stereotype `ExternalDocs` extends the metaclasses `Model`, `Class`, and `Operation` and allows the definition of additional documentation regarding the extended element. It includes a description and the URL of the documentation.

Security Elements. Figure 3.12 shows the stereotypes with allow the definition of the security details of a REST API. The stereotype `SecurityDefinitions` extends the metaclass `Model` and allows the definition of the scheme definitions that can be used across the specification. The datatype `SecurityScheme` allows the definition of a security scheme and includes the type of the security scheme which can be `basic`, `apiKey`, and `oauth2`; and a description. Fore API key authorizations, the security scheme allows the definition of the name of the key and its location. For

OAuth2 authorizations, the security schema allow the definition of the flow used by the scheme, the authorization URL used by this flow, the token URL to be used, and a list of scopes. The definitions of `SecurityScope`, `APIKeyLocation` and `OAuth2FlowType` elements are similar to the ones of the OpenAPI metamodel.

The stereotype `SecurityRequirements` extends the metaclasses `Model` and `Operation` and allows the declaration of the possible authorization methods for such model or operation, respectively. This stereotype includes a list alternative security schemes that can be used for the API as a whole when applied to the model element, or a particular operation when applied to an operation. The data type `SecurityRequirement` allows the declaration of a security requirement and includes a name corresponding to a security scheme which is declared in the security scheme definitions and a list of scope names in case of an OAuth authorization.

Figure 3.13 shows an excerpt of the UML model of the Petstore API including the applied OpenAPI stereotypes. As can be seen, the stereotypes `API`, `APIInfo`, and `SecurityDefinitions` are applied to the model element to include information about the API as a whole (e.g., `host` and `basePath`), metadata (e.g., `version`), and security definitions, respectively. The stereotype `Schema` is applied to the classes *Pet* and *Tag* to mark them as schema definitions. The stereotype `APIProperty` is applied to the properties of the classes *Pet* and *Tag*. The stereotype `APIOperation` is applied to the operation *findPetsByStatus* to specify the details of the operation (e.g., the relative path, the HTTP method). The stereotypes `APIParameter` and `APIResponse` are applied to the parameter *status* and the returned *Pet* list to add information about the operation parameter (e.g., location of the parameter) and the API response (e.g., code of the response), respectively.

3.1.3 *Tool Support*

This section presents the tool support for OpenAPI which consists of a set of Open Source Eclipse plugins, namely: (1) OPENAPIMM, a plugin for the OpenAPI metamodel; OPENAPIPROFILE, a plugin for the UML profile for OpenAPI; and (3) OPENAPITOUML, a plugin to generate UML models from OpenAPI definitions.

3.1.3.1 OpenAPIMM

OPENAPIMM is an Open Source Eclipse plugin we created to provide an implementation of the OpenAPI metamodel and manage OpenAPI models.

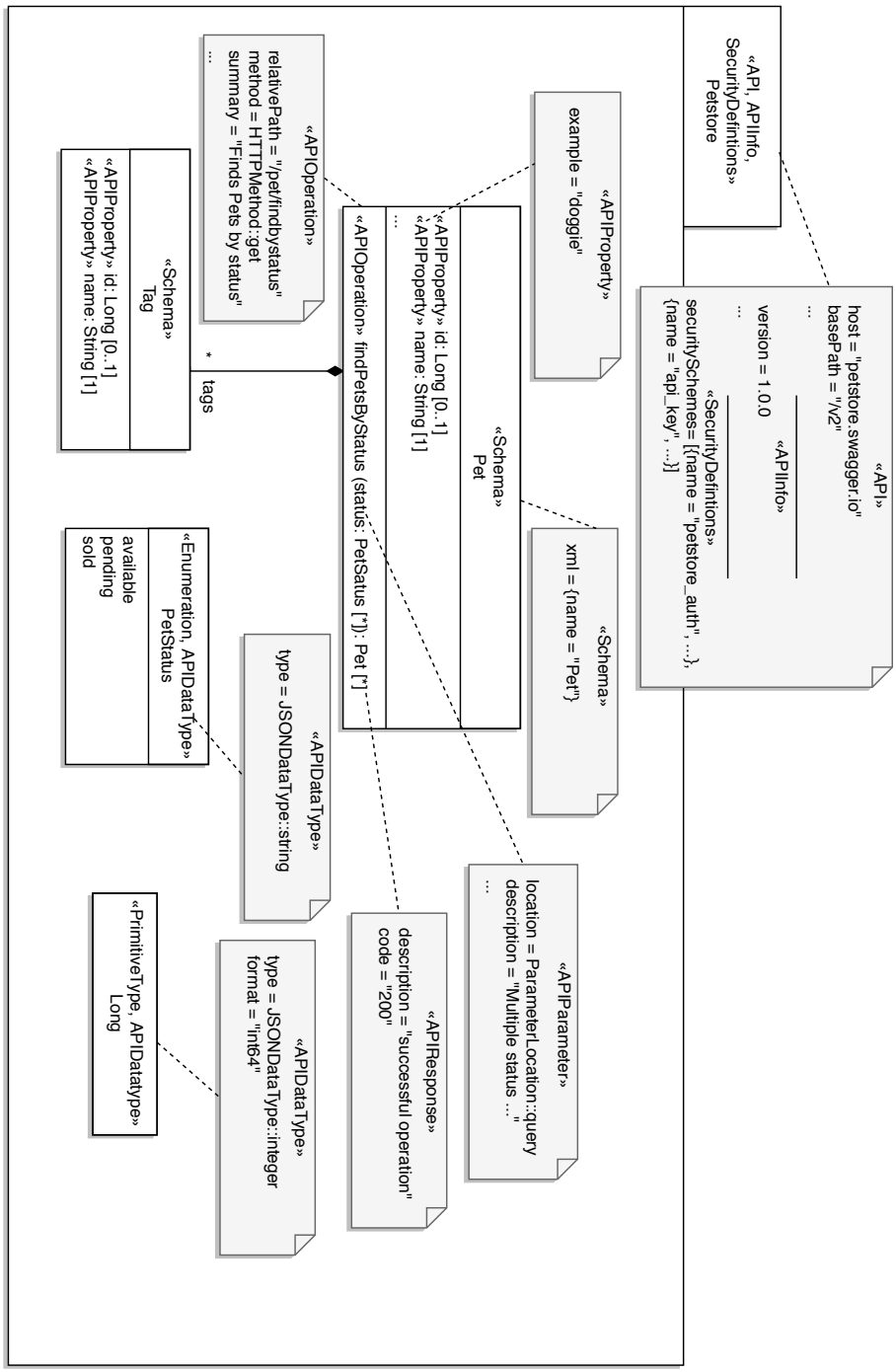


Figure 3.13: OpenAPI profile: Petstore example.

The plugin relies on EMF[10], *de facto* modeling framework in Eclipse. Thus, the OpenAPI metamodel has been implemented as an *Ecore* model which has been used to generate a Java API to manage OpenAPI models. The metamodel implementation has been enriched with a set of OCL constraints to enable the validation of OpenAPI models against the OpenAPI specification. The generated Java API has been extended to add a set of methods to facilitate the traversal of models (e.g., find operations, find definitions) and to validate models by executing the executing the OCL queries.

To facilitate the extraction and serialization of OpenAPI models, the plugin includes also a contextual menu to: (1) generate OpenAPI models from JSON files and (2) generate JSON files from OpenAPI models. The support for JSON relies on the framework `google-gson`[11].

The model extractor generates an OpenAPI model from an OpenAPI-compliant JSON file. Such process is rather straightforward, as our metamodel mirrors the structure of the OpenAPI specification. Only special attention had to be paid to resolve JSON references (e.g., `#/definitions/Pet`). Thus, the root object of the JSON file is transformed to an instance of the `API` model element, then each JSON field is transformed to its corresponding model element. Figure 3.6.b shows an excerpt of the generated OpenAPI model of the Petstore API including the *findPetsByStatus* operation and a partial representation of the `Pet` schema.

The JSON serializer creates a OpenAPI-compliant JSON file from an OpenAPI model by means of a model-to-text transformation. The root object of the JSON file is the `API` model element, then each model element is transformed to a pair of name/value items where the type for the value is (i) a string for primitive attributes, (ii) a JSON array for multivalued element or (iii) a JSON object for references. Serialization/deserialization model elements are used to resolve references. As said in Section 3.1.1, elements such as `Schema`, `Parameter`, and `Response` can be declared in different locations and reused by other elements. While the `declaringContext` reference is used to define where to declare the object, the `ref` attribute (inherited form `JSONPointer` class) is used to reference this object from another element.

OPENAPIMM will be used in Chapters 5, 6, and 8 to discover OpenAPI definitions, generate test cases for REST APIs, and compose REST APIs, respectively. OPENAPIMM is available in our GitHub repository[12].

[10] `https://www.eclipse.org/modeling/emf/`

[11] `https://github.com/google/gson`

[12] `https://github.com/SOM-Research/openapi-metamodel`

3.1.3.2 OpenAPIProfile

`OpenAPIProfile` is an Open Source Eclipse plugin which provides the implementation of the OpenAPI profile and an editor to use the profile with UML Class diagrams. The plugin extends PAPYRUS modeling editor[13], an Open Source UML modeling editor in Eclipse, in order to integrate the OpenAPI profile with Papyrus UI. The instructions on how to install and use the plugin are described in our GitHub repository[14].

3.1.3.3 OpenAPItoUML

This section presents OPENAPITOUML, a tool which generates UML models from OpenAPI definitions. Targeting the well-known UML standard, which has been successfully adopted in the industry[15], allows our tool to be useful in a variety scenarios. For instance, it can be used for documentation purposes by offering a better visualization of the capabilities of REST APIs using the well-known UML class diagram. To the best of our knowledge, current documentation tools for OpenAPI (e.g., ReDoc[16] and Swagger UI[17]) display the operations and data structures of the definitions in HTML pages using only text and code samples, which complicate the understanding and visualization of REST APIs. Only JSONDiscoverer [CC16] allows visualizing the data schema of JSON-based REST APIs but focus on the inputs/outputs of the operations and does not model the operations themselves. OPENAPITOUML can also be used to ease the integration of REST APIs in all kinds of model-based processes.

OPENAPITOUML follows a model-based approach to visualize OpenAPI definitions as UML Class diagrams. Given an input OpenAPI definition, the OPENAPITOUML approach extracts first an OpenAPI model which is then transformed into a UML model (i.e., Class diagram) showing the data structure and operation signatures of the API. While the intermediate OpenAPI model is useful to perform other kinds of advanced analysis on the OpenAPI definition, it is more convenient to generate a UML model for visualization and comprehension purposes. Being a standard UML model, our result can be automatically rendered and modified using any of the plethora of UML modeling tools (e.g., PAPYRUS or UML DESIGNER).

[13] `https://www.eclipse.org/papyrus/`
[14] `https://github.com/SOM-Research/openapi-profile`
[15] `https://www.uml.org/uml_success_stories/index.htm`
[16] `http://rebilly.github.io/ReDoc/`
[17] `https://swagger.io/swagger-ui/`

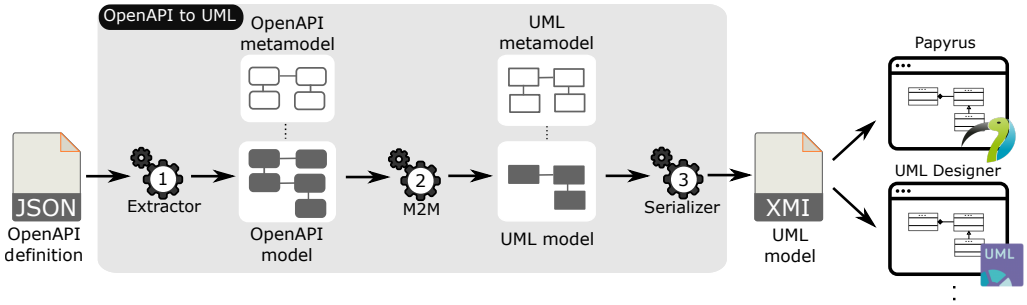


Figure 3.14: The OPENAPITOUML approach.

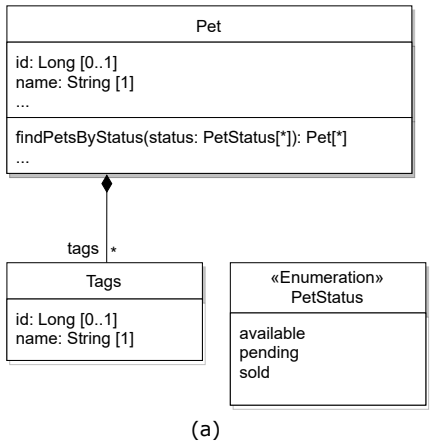
The OPENAPITOUML process is depicted in Figure 3.14. As can be seen, the process takes as input an OpenAPI definition, which can be (i) provided by the API provider; (ii) generated by tools such as APIDISCOVERER, which allows discovering OpenAPI definitions from API call examples as we will present in Chapter 5; or (iii) derived from other API definition formats (e.g., API Blueprint, RAML) using tools such as API TRANSFORMER[18], which allows converting API definitions.

The OPENAPITOUML process generates UML models in three steps (see the steps 1, 2, and 3 in Figure 3.14), which we will illustrate with the Petstore API example. Figure 3.6.a showed an excerpt of the Petstore OpenAPI definition including the operation `findPetsByStatus` and the schema definition `Pet`.

The first step (see step 1 in Figure 3.14) extracts a model conforming to our OpenAPI metamodel from the input OpenAPI definition. This step relies on the OPENAPIMM presented in Section 3.1, which provides the implementation of the OpenAPI metamodel and the support to extract OpenAPI models from OpenAPI definitions. The second step (see step 2 in Figure 3.14) performs a model-to-model transformation to generate a model conforming to the UML metamodel from the previously extracted OpenAPI model. This transformation iterates over the operations and definitions of the OpenAPI model in order to generate classes, properties, operations, data types, enumeration, and parameters, accordingly. This process relies on a set of heuristics to identify the most adequate UML class to attach each OpenAPI operation to. Heuristics are based on the analysis of the `tags`, `parameters` and `responses` of the operation[19]. The full list of heuristics can

[18] `https://apimatic.io/transformer`

[19] When no class is a good fit for the operation, an artificial class is created to host the operation.



(a)

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001" ...>
  <packagedElement xmi:type="uml:Class"
  name="Pet">
    <ownedAttribute xmi:type="uml:Property"
     name="id">
      ....
    </ownedAttribute>
    <ownedOperation xmi:type="uml:Operation"
    name="findPetsByStatus" ...>
      <ownedParameter xmi:type="uml:Parameter"
      name="status"/>
      ...
      </ownedParameter>
    </ownedOperation>
  <packagedElement xmi:type="uml:Class"
  name="Tag">
   ...
  </packagedElement>
  <packagedElement xmi:type="uml:Enumeration"
   name="PetStatus">
   ...
  </packagedElement>
  ...
</uml:Model>
```

(b)

Figure 3.15: The Petstore example: (a) generated UML model, and (b) serialized UML model.

be found in the tool repository[20].

Figure 3.15.a shows an excerpt of the generated UML model for the Petstore API. As can be seen, the OpenAPI schema `Pet` is transformed to the UML class `Pet`, while the OpenAPI operation `findPetsByStatus` is transformed into the UML operation `findPetsByStatus` in the `Pet` class.

The last step of the process (see step 3 in Figure 3.14) serializes the generated UML model as an XMI file (standard XML format for UML tool interoperability). Figure 3.15.b shows an excerpt of the serialized UML model for the Petstore API. Users can rely on tools such as PAPYRUS and UML DESIGNER to open and visualize such file.

OPENAPITOUML has been implemented in Java as a plugin for the Eclipse platform[21]. The plugin extends the platform to provide a contextual menu to obtain a UML model from an OpenAPI definition (using its JSON representation format). Figure 3.16 shows a screenshot of our plugin including the created contextual menu (on the left side) and the generated Class diagram for the Petstore API displayed using PAPYRUS modeling editor. The OpenAPI metamodel has been implemented using EMF as shown before,

[20] `https://github.com/SOM-Research/openapi-to-uml#notes`

[21] `https://github.com/SOM-Research/openapi-to-uml`

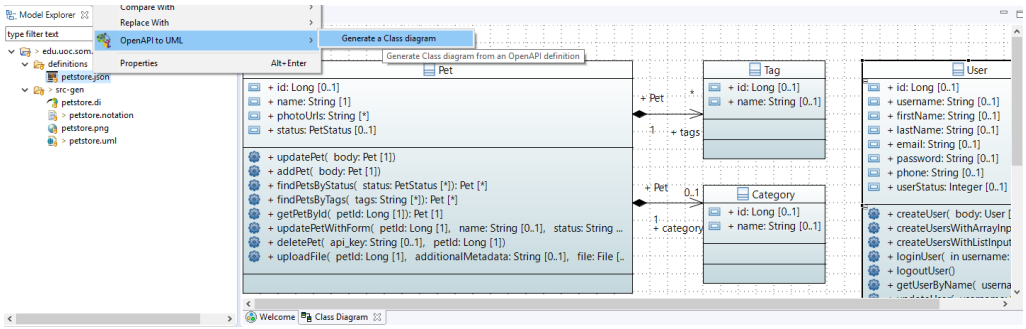


Figure 3.16: A screenshot of the OPENAPITOUML plugin.

while UML models rely on UML2[22], an EMF-based implementation of the UML 2.5 OMG metamodel.

3.2 MODELING ODATA

This section presents the model-based representations we created for OData protocol. In particular, Section 3.2.1 describes the OData metamodel we created to specify OData entity models and Section 3.2.2 presents the UML profile for OData entity models. Finally, Section 3.2.3 presents the tool support for OData.

3.2.1 *The OData Metamodel*

This section presents the OData metamodel, a metamodel we created to specify OData services. The OData metamodel is aligned with the OData CSDL specification [PHZ14e], which defines the main concepts to be exposed by any OData service. Figure 3.17 shows an excerpt of this metamodel.

The top part of the metamodel comprises the `ODService` element, which includes a set of schemas (i.e., `schemas` reference). One or more schemas define the data model of an OData service. A schema is represented by the `ODSchema` element which includes the namespace of the schema (e.g., `com.example.ODataDemo`) and an alias for the schema namespace (e.g., `ODataDemo`). It also includes references to the data structures defined by the schema which comprise enumerations (i.e., `enumTypes` reference), complex types (i.e., `complexTypes` reference) and entity types (i.e., `entityTypes`

[22] `https://wiki.eclipse.org/MDT/UML2`

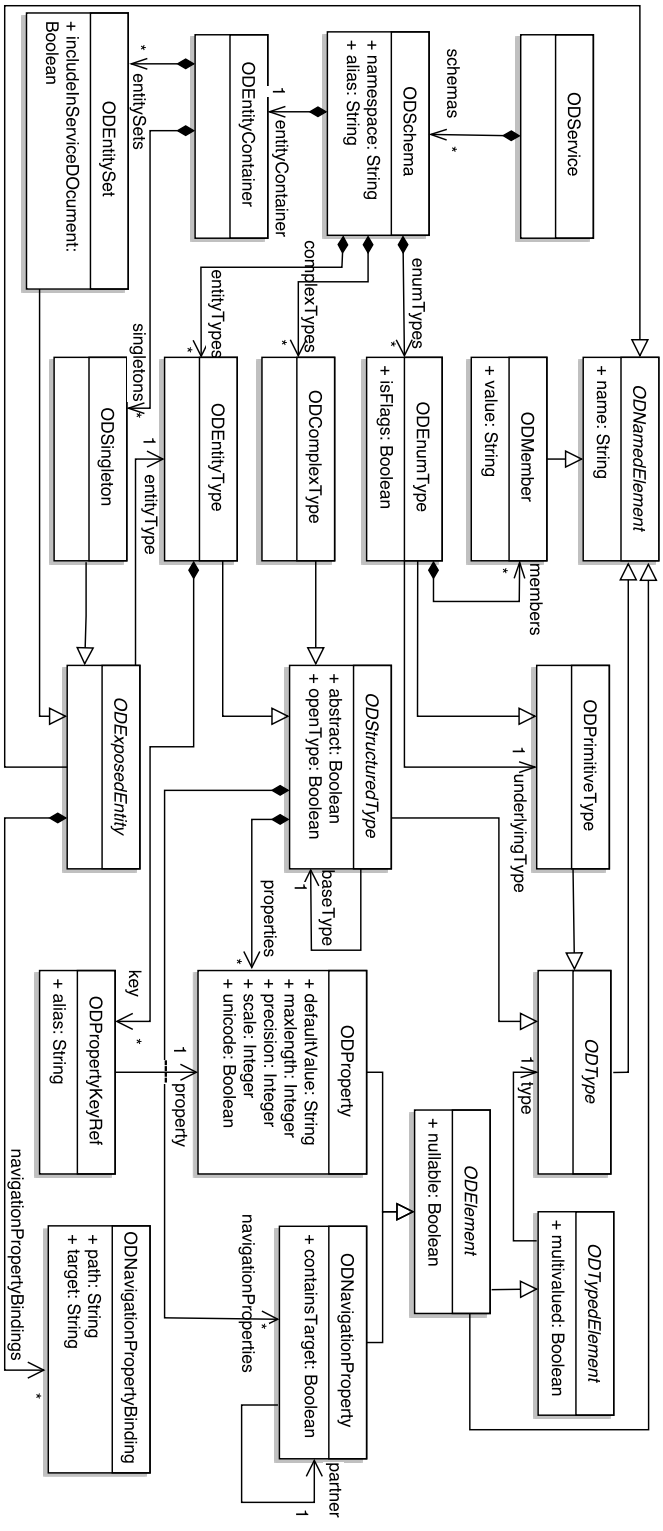


Figure 3.17: An excerpt of OData metamodel.

reference). All data structures in the metamodel are subtypes of the ODType element which describes the structure of an abstract data type.

An enumeration is represented by the ODEnumType element which is a subtype of the ODPrimitiveType element and includes a name (inherited from ODNamedElement), an attribute indicating whether the enumeration allows multi-selection (i.e., isFlags attribute), an underlying primitive type (i.e., underlyingType reference), and a list of members (i.e., members reference). The element ODMember defines the options for the enumeration type and includes a name (inherited from ODNamedElement) and a value.

Entity types are named structured types with a key, while complex types are keyless named structured types. Entity types and complex types are represented by the ODEntityType and ODComplexType elements, respectively. Both elements are subtypes of the ODStructuredType abstract element which represents a structured type. The ODStructuredType element is a subtype of the ODType element and includes a name (inherited from ODNamedElement), an attribute indicating whether the structural type cannot be instantiated (i.e., abstract attribute), and an attribute indicating whether undeclared properties are allowed (i.e., openType property[23]). A structured type is composed of structural properties (i.e., properties reference) and navigation properties (i.e., navigationProperties reference) and may define a base type (i.e., baseType reference). Additionally, the ODEntityType element includes a key (i.e., key reference) which indicates the properties identifying an entity (i.e., property reference of the element ODPropertyKeyRef) and an alias name for each property.

The ODProperty and ODNavigationProperty elements represent a structural property and a navigation property, respectively. While the ODProperty element defines an attribute of a structured type, the element ODNavigationProperty defines an association between two entity types. Both elements are subtypes of the ODElement abstract element which defines the common features of structural properties. This element includes a name (inherited from ODNamedElement), an attribute indicating whether the element can be null (i.e., nullable property), a type (i.e., type reference inherited from ODTypedElement), and a cardinality (i.e., the multivalued property inherited from ODTypedElement). Additionally, the ODProperty element includes several attributes to provide additional constraints about the value of the structural property (e.g., maxLength and precision properties). The ODNavigationProperty element, on the other hand, includes a containment attribute (i.e., containsTarget reference) and an opposite

[23]Open types entities allows clients to persist additional undeclared properties.

navigation property (i.e., `partner` reference).

The `ODSchema` element also includes an entity container (i.e., `entityContainer` reference) defining the entity sets and singletons queryable and updatable by the service. The `ODEntityContainer` element defines an entity container and includes a set of entity sets (i.e., `entitySets` reference) and singletons (i.e., `singleton` reference). An entity set allows addressing a collection of entities, while a singleton allows addressing a single entity directly from the entity container. The two concepts are materialized by the `ODEntitySet` and `ODSingleton` elements, respectively. Both elements are subtypes of the `ODExposedEntity` abstract element which includes a reference to the target entity type (i.e., `entityType` reference), a name (inherited from `ODNamedElement`), and a set of navigation properties bindings (i.e., `navigationPropertyBindings` reference).

OData metamodel also includes elements to define annotations and vocabularies which provide an extension mechanism to add additional characteristics or capabilities of OData elements. The complete metamodel is available in our GitHub repository[24].

3.2.2 *A UML Profile for OData*

This section presents the UML profile for OData Web APIs. This profile extends the UML class diagram which share similar concepts with OData. We organized the OData profile into two parts, namely: (i) the EDM which describes the data exposed by an OData Web service, and (ii) the advanced configuration model, which defines additional characteristics or capabilities of OData Web APIs (i.e., what parts of the EDM can be modified, what permissions are needed,...). Next, we present the OData profile. Later, we will present the rules we defined to apply and initialize this profile.

3.2.2.1 The Entity Data Model

OData Service Wrapper. An OData Web service exposes a single entity model which may be distributed over several schemas, and should include an entity container that defines the resources exposed by the Web service. Figure 3.19a shows the extension of UML to define these elements. We consider that the entity model is defined in one schema, represented by the element `Model` of a UML class diagram. Thus, `ODService` stereotype extends

[24] `https://github.com/SOM-Research/odata-generator/tree/master/metamodel/som.odata.metamodel`

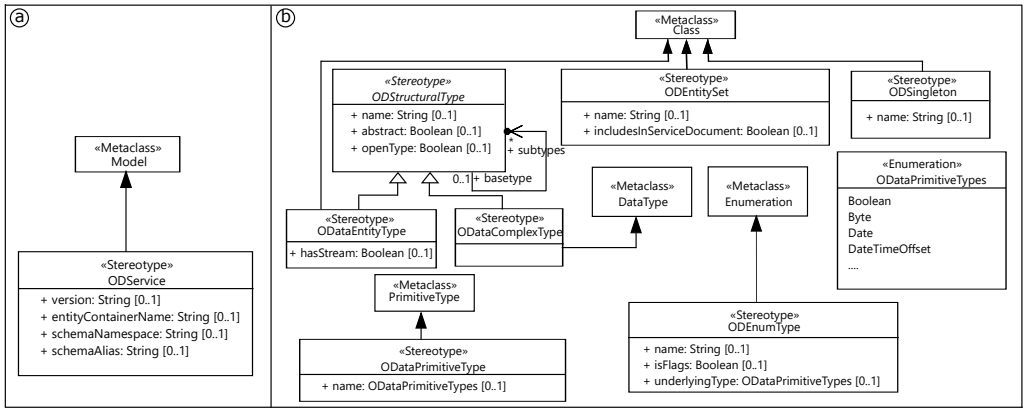


Figure 3.18: OData profile: (a) the service wrapper and (b) data types elements.

the metaclass `Model` and includes: the version the OData specification, the namespace of the schema (e.g., `com.example.ODataDemo`), an alias for the schema namespace (e.g., `ODataDemo`), and the name of the entity container (e.g., `ODataService`).

Data Types. An OData entity model defines data types in terms of structural types, enumerations, and primitive types. There are two kinds of structural types: *entity types* and *complex types*. An *entity type* is a named structured type which defines the properties and relationships of an entity. *Entity types* are mapped to the concept `Class` in a UML model. A *complex type* is a named structural type consisting of a set of properties. *Complex types* are mapped to the concept `DataType` in a UML model.

Figure 3.18b shows the stereotypes related to data types and their mapping with UML concepts. The abstract stereotype `ODStructuralType` defines the common features of all the structural types and includes a name, a property indicating whether the structural type cannot be instantiated (i.e., `abstract` property), and a property indicating whether undeclared properties are allowed (i.e., `openType` property[25]).

`ODStructuralType` supports also the concept of inheritance by allowing the declaration of a base structural type (i.e., `basetype` association). The stereotypes `ODEntityType` and `ODComplexType` inherit from `ODStructureType` and extend the metaclasses `Class` and `DataType`, respectively. Additionally `ODEntityType` includes the `hasStream` property, indicating if the entity is a media entity (e.g., photograph). The stereo-

[25]Open types entities allows clients to persist additional undeclared properties.

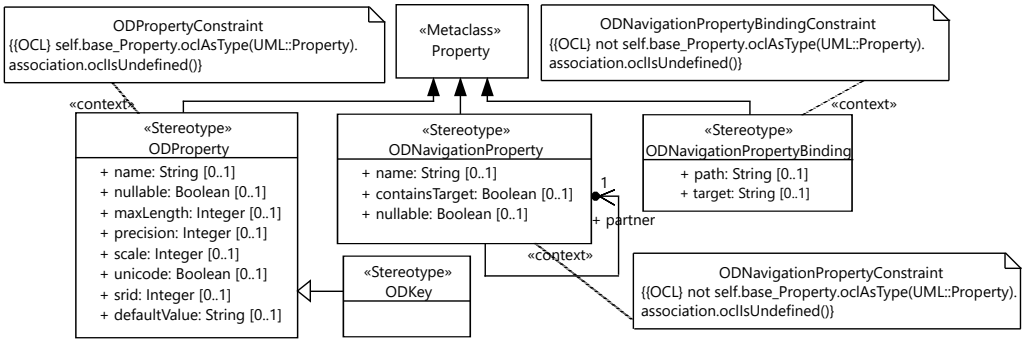


Figure 3.19: OData profile: properties and associations stereotypes.

type `ODPrimitiveType` extends the metaclass `PrimitiveType` and includes a name which corresponds to the associated OData primitive type (e.g., Binary, Boolean, etc.). The stereoType `ODEnumType` extends the metaclass `Enumeration` and includes a name, a boolean property indicating whether more than one member may be selected at a time (i.e., `IsFlags` property), and the corresponding OData type.

The profile also allows modeling the entity sets and singletons exposed by the OData service. While an entity set can expose instances of the specified entity type, a singleton allows addressing a single entity directly from the entity container. These two concepts are materialized with the stereotypes `ODEnititySet` and `ODSingleton` which extend the metacalass `Class`.

Properties and Associations. Properties define the structure and the relationships in OData. Structural properties define the attributes of an entity type or a complex type whereas navigation properties define associations between entity types. In UML, the element `Property` is a `StructuralFeature` which, when related by `ownedAttribute` to a `Classifier` (other than `Association`), represents an attribute, and when related by `memberEnd` of an `Association`, represents an association end. Both structural properties and navigation properties are mapped to the concept `Property` in a UML model.

Figure 3.19 shows the stereotypes defining properties and associations. The stereotypes `ODProperty` and `ODnavigationProperty` represent a structural property and a navigation property, respectively. They both extend the metaclass `Property`. The stereotype `ODProperty` includes a name and several attributes to provide additional constraints about the value of the structural property (e.g., `nullable`, `maxLength` properties). Additionally, the stereotype `ODKey` inherits from `ODProperty` and defines a property as the

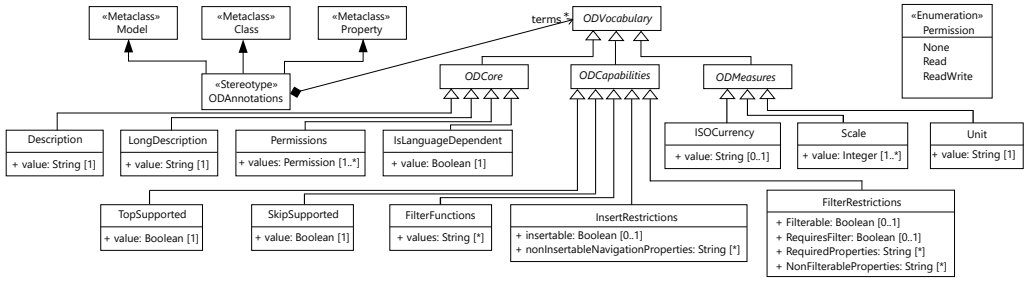


Figure 3.20: OData profile: annotation and vocabulary stereotypes.

key of the entity type (required for a an OData entity type). The stereotype `ODNavigationProperty` includes a name, a containment property, and a nullable property. The stereotype `ODNavigationPropertyBinding` extends also the metaclass `Property` and defines a navigation binding for the corresponding entity set.

To ensure the validity of the applied stereotypes, we have enriched the profile with a set of constraints written using OCL [CG12]. For instance, since the stereotypes related to properties and navigations properties extend all the metaclass `Property`, `ODPropertyConstraint` ensures that the stereotype `ODProperty` is applied to a UML property element representing an attribute.

Advanced Configuration of OData Services. OData defines annotations to specify additional characteristics or capabilities of a metadata element (e.g., entity type, property) or information associated with a particular result (e.g, entity or property). For example, an annotation could be used to define whether a property is read-only. Annotations consist of a term (i.e., the namespace-qualified name of the annotation), a target (the element to which the term is applied), and a value. A set of related terms in a common namespace comprises a vocabulary. Our profile supports the three standardized vocabularies defined by OData, namely: the core vocabulary, capacity, and measures.

Figure 3.20 shows an excerpt of the profile defined for representing annotations. The stereotype `ODAnnotations` extends the metaclasses `Model`, `Class`, and `Property`, and has an association with `ODVocabulary`, thus allowing adding annotations according to the vocabularies. `ODVocabulary` is the root class of the hierarchy of vocabularies supported by the OData profile (i.e., core, capabilities, and measures vocabularies). OData profile defines (i) the `ODCore` hierarchy which includes the core vocabularies such as documentation (e.g., the class `Description`), permis-

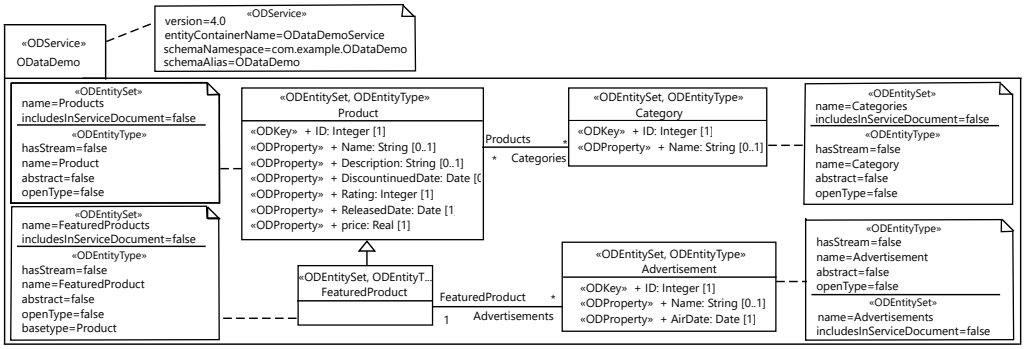


Figure 3.21: UML class diagram of the running example annotated by the generator.

sions (i.e., the class `Permissions`, and localization (i.e., the data type `IsLanguageDependent`); (ii) the `ODCapabilities` hierarchy which is used to explicitly specify Web API capabilities (e.g., `TopSupported` for query capabilities or `InsertRestriction` for data modification); and (iii) the `ODMeasures` hierarchy to describe monetary amounts and measured quantities (e.g., `ISOCurrency`).

3.2.2.2 Default Profile Generation

Our OData profile can be used to annotate any new or preexisting UML class diagram. Nevertheless, to simplify the application of our profile, we have also developed a model-to-model transformation that given an standard UML model, returns an annotated one by relying on a set of default heuristics that embed our knowledge on typical uml-to-odata design decisions. This annotated model can be regarded as just an initial option to bootstrap the process that the designer can then modify at will.

Table 3.3 summarizes our mapping strategies. From left to right, the columns of the table show (i) the involved UML element; (ii) the conditions to apply an stereotype (if any), (iii) the stereotype to be applied; and (iv) the values of the stereotype properties. In a nutshell, each class is mapped to an entity type and is exposed as entity set, each attribute is mapped to a property, and each navigable association is mapped to a navigation property. Figure 3.21 shows the running example including some of the generated OData profile annotations. This first version of the class diagram can later be customized and used in other model-driven processes to fast prototyping

Table 3.3: Rules of the OData profile annotation generator.

UML ELEMENT	CONDITION	STEREOTYPE	VALUE
m: Model	-	ODService s	- s.version = "4.0" - s.entityContainerName= m.name+"Service" - s.schemaNamespace = "com.example."+m.name - s.schemaAlias = m.name
c: Class	-	ODEntityType et	-et.name = c.name -if c.abstract == true then et.abstract = true -if c.generalization contains t then et.basetype=ot (ot is the entity type of t)
		ODEntitySet es	- es.name = the plural form of c.name
p: Property	p is an class attribute OR a data type attribute	ODProperty op	- op.name = p.name - if p.lower == 1 then op.nullable = false -op.defaultValue = p.default
	p is an class attribute marked as key	ODKey ok	- ok.name = p.name
	p is an navigable association end	ODNavigationProperty np	- np.name = p.name - if p.lower == 1 then np.nullable = false - if p.aggregation == composite then np.containsTarget = true
		ODNavigationProperty Binding npb	- npb.path = p.name - npb.target = the name of the corresponding entity set of the association end
dt: DataType	-	ODComplexType ct	- ct.name = dt.name - if dt.abstract == true then ct.abstract = true - if dt.generalization contains t then ct.base=ot (ot is the complex type of t)
pt: PrimitiveType	-	ODPrimtiveType opt	- opt.value = the corresponding primitive type of pt.name
e: Enumeration	-	ODEnumType oe	- oe.name = e.name

OData Web APIs.

3.2.3 *Tool Support*

We created two Open Source Eclipse plugins for OData, namely: (1) ODATAMM, which provides an implementation for the OData metamodel; and (2) ODATAPROFILE, which provides an editor for the OData profile.

ODATAMM. The ODATAMM plugin provides the tools and SDK for the OData metemodel. Similar to the the OpenAPI metamodel, OData metamodel relies on the EMF framework to define the metamodel and to generate an OData SDK. Thus, OData metamodel has been implemented as an *Ecore* model which has been used to generate a Java API to manage OData models. The plugin includes also menu which allow exporting an OData model as a Metadata document. As you will see in Chapter 7, we rely on the the ODATAMM plugin and OData metamodel to generate OData services. The ODATAMM plugin can be found in our GitHub repository[26].

ODATAPROFILE. Similar to OPENAPIPROFILE, the ODATAPROFILE plugin provides an editor based on PAPYRUS allowing to annotate UML class diagrams with OData stereotypes. The plugin provides also a contextual menu allowing to apply and initialize the OData profile following the rules described before. The ODATAPROFILE plugin can be found in our GitHub repository[27].

3.3 SUMMARY

In this chapter we have presented a set of modeling resources for the OpenAPI specification and OData protocol. For both, we have presented a metamodel and UML profile, thus giving more flexibility to the modeler in order to choose which technique suits him/her best. We have also presented the tools which implement these resources and help manipulate them. This is the first step to boost the model-based development of REST APIs with OpenAPI or OData, offering developers the opportunity to leverage on the plethora of modeling tools to define forward and reverse engineering to generate, visualize and manipulate REST APIs.

[26] https://github.com/SOM-Research/odata-generator/tree/master/metamodel

[27] https://github.com/SOM-Research/OData

CHAPTER

4 *APIfication* of Models

This chapter presents EMF-REST, an approach that leverages on MDE techniques to *APIfy* EMF models (i.e., provide REST APIs for EMF models). EMF, *de facto* modeling framework in ECLIPSE, is the backstone of a plethora of modeling environments and frameworks. However, most of these environments and frameworks are usually bound to desktop-based scenarios, thus restricting their usage. EMF-REST tries to elevate this situation by promoting model management in distributed environments using REST APIs and therefore advancing towards the portability of modeling tools to the Web. EMF-REST takes advantage of model and Web-specific features such as model validation and security, respectively.

This chapter is structured as follows. Section 4.1 presents the EMF-REST approach. Section 4.2 describes the running example we will use to illustrate our approach. Section 4.3 describes how we devised the mapping between EMF and REST principles, while Section 4.4 describes the additional EMF-REST features. Section 4.5 presents the technical architecture of the generated REST API. Section 4.6 describes the steps we followed to generate the API. Section 4.7 discusses some related work. Finally, Section 4.8 summarizes this chapter.

4.1 OUR APPROACH

Figure 4.1 shows an overview of the EMF-REST approach. As can be seen, EMF-REST takes as input a metamodel (i.e., *Ecore* model) and relies on

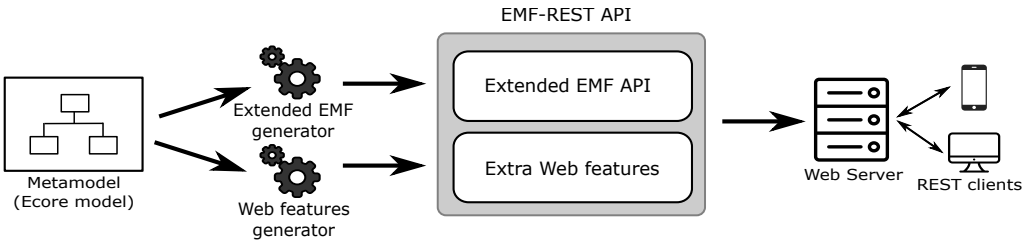


Figure 4.1: EMF-REST global approach.

two generators to derive an EMF REST API, namely: *the extended EMF generator* and *the Web features generator*. On the one hand, *the extended EMF generator* extends the EMF generation facility in order to adapt the generated Java API to the Web. On the other hand, *the Web features generator* generates extra Web features to enable the encapsulation of the generated API as a REST API. The result of the generation process is a-ready-to-deploy REST API where users can use HTTP requests to manage instances of the input metamodel. The mapping between EMF with REST will be presented in Section 4.3, while the technical details of the generated API and the generation process will presented in Sections 4.5 and 4.6, respectively.

In the following we will present the running example which we will use to illustrate the EMF-REST approach.

4.2 Running Example

To illustrate our approach, we will use a running example to create a REST API aimed at managing Interaction Flow Modeling Language (IFML) models. IFML is an OMG standard language designed for expressing the content, user interaction and control behavior of the front-end applications developed for systems such as computers and mobile phones [BF+14].

Figure 4.2 shows an excerpt of the IFML metamodel. As can be seen, an IFML model is represented by the `IFMLModel` element. This element contains an interaction flow model (i.e., `interactionFlowModel` reference) which defines the user view of an application. The `InteractionFlowModel` represents an interaction flow model and includes a set of interaction flow elements (i.e., `interactionFlowElements` reference). The `InteractionFlowElement` element represents an abstract interaction flow element. The elements `Event`, `ViewElement`, `ViewComponentPart` are specializations of this element and allow the definition of an event, a view element, and a view component part, respectively. The `OnSubmitEvent` is

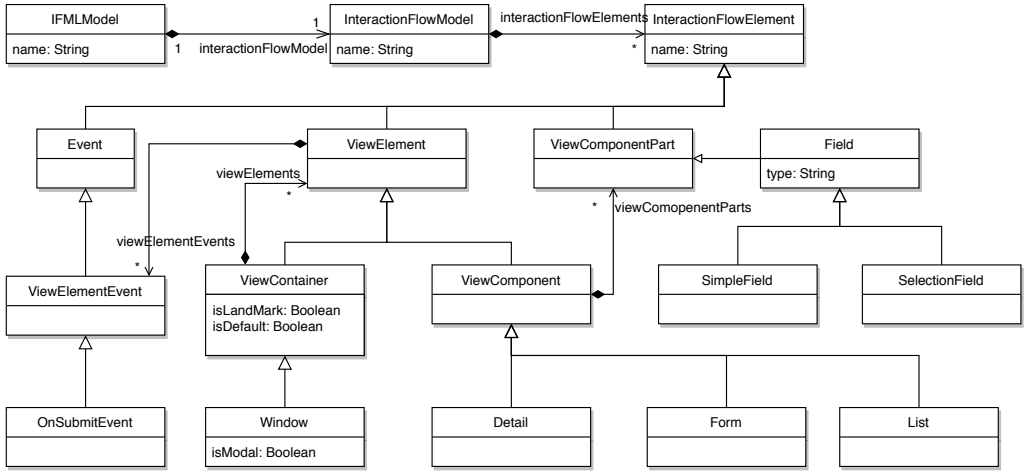
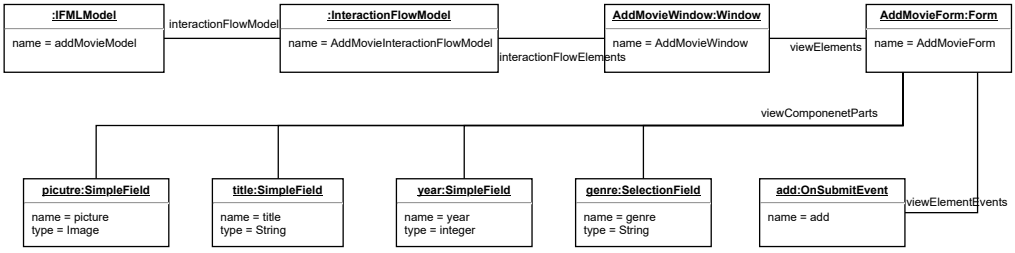


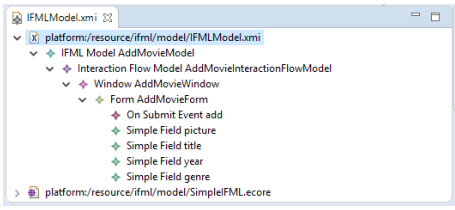
Figure 4.2: Simple *Ecore* model of an IFML subset.

a specialization of `ViewElementEvent` (a specialization of `Event`) and allows the definition of a submission event. The elements `ViewContainer` and `ViewComponent` are two specializations of `ViewElement` and allow the definition of a view container and a view component, respectively. The `Window` element is a specialization of `ViewContainer` and allows the definition of a window container. The elements `Detail`, `Form`, and `List` are specializations of `ViewComponent` and allow the definition of a component to represent the details of an element, a form, and a list of elements, respectively. A view component may contain a set of view component parts (i.e., `viewComponentParts` reference). Finally, the elements `SimpleField` and `SelectionField` are specializations of `Field` (a specialization of `ViewComponentPart`) and allow the definitions of the fields in a form.

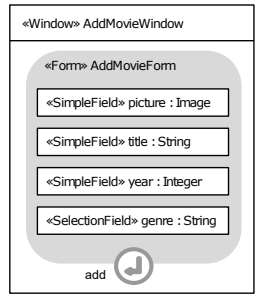
Figure 4.3 shows an IFML model for a form to add a movie to a collection. Figure 4.3.a. shows the object diagram of the model, while Figures 4.3.b. and 4.3.c. show the Eclipse tree view of the model and the concrete IFML model using IFML syntax, respectively. The model is composed of: (i) a `Window` container named *AddMovieWindow*, (ii) a `Form` component named *AddMovieForm*, (iii) a list of fields of types `SimpleField` and `SelectionField` representing the elements of the form, and finally (iv) a `ViewElementEvent` of type `OnSubmitEvent` allowing to submit the form. In what follows we will see how we would allow creating the *AddMovieForm* form by calling a REST API generated from the IFML model following the REST architecture style.



(a)



(b)



(c)

Figure 4.3: IFML model example: (a) object diagram, (b) abstract syntax tree, (c) concrete IFML syntax.

4.3 MAPPING EMF AND REST

The first step to build EMF-REST is to align EMF with REST. We rely on EMF to represent the models from which the REST APIs are generated. As models and their instances are managed by the corresponding APIs provided by the framework (i.e., *Ecore* and *EObject* APIs, respectively), we need to define a mapping between such APIs and REST. In the following we explain how we map EMF with REST. In particular, we will explain how to address model elements, how to represent them, and how the use HTTP's uniform interface to manipulate them. REST architectural style was presented in Section 2.1.

4.3.1 *Identification of Resources*

Models in EMF are addressed via a URI, which is a string with a well-defined structure as shown in Expression (4.1). This expression contains three parts specifying: (1) a *scheme*, (2) a *scheme-specific part* and (3) an optional *fragment*. The scheme is the first part separated by the *":"* character

and identifies the protocol used to access the model (e.g., *platform*, *file* or *jar*). In Eclipse we use *platform* for URIs that identify resources in Eclipse-specific locations, such as the workspace. The scheme-specific part is in the middle and its format depends on the scheme. It usually includes an *authority* that specifies a host, the *device* and the *segments*, where the latter two constitute a local path to a resource location. The optional fragment is separated from the rest of the URI by the # character and identifies a subset of the contents of the resource specified by URI, as we will illustrate below. Expression (4.2) shows an example of a platform-specific URI which refers to the *AddMovie* model, represented as a file *AddMovie.xmi* contained in a project called *project* in Eclipse workspace. It is important to note that in EMF model instances include a reference to the *Ecore* model they conform to.

```
[scheme:][scheme-specific-part][#fragment]          (4.1)
platform:/resource/project/AddMovie.xmi             (4.2)
```

We map the previous URI to a Uniform Resource Locator (URL) as follows. The base URL pattern of a model instance is defined by Expression (4.3). In the pattern, the part *https://[application-Link]/rest* is the URL of the Web application, *modelId* is the identifier of the model (i.e., the *Ecore* model) and *ModelInstanceId* is the identifier of the model instance being accessed (the XMI file). The URL (4.4) represents an example to retrieve the IFML model used in the example. As can be seen, while the URI can address a file representing a model instance (where a reference to the *Ecore* model is included), the URL requires indicating the identifier of both the *Ecore* model and the model instance.

```
https://[applicationLink]/rest/[ModelId]/[ModelInstanceId]   (4.3)
https://example.com/rest/IFMLModel/AddMovie                  (4.4)
```

This URL acts as the entrypoint for a particular model instance and points to its root element, which is normally the case in EMF. When the model instance has more than one root, we point at the first.

Once pointing to the root of a model instance, addressing a particular element of the model in the EMF is done by using the part *fragment* in Expression (4.1). The navigation is done using the reference names in the *Ecore* model. For instance, the concept *IFMLModel* has the reference *interactionFlowModel* to access the *InteractionFlowModel*. Using the EMF

API, the URI is shown in Expression (4.5), while using the Web API, the URL is shown in Expression (4.6).

```
platform:/resource/project/AddMovie.xmi#//@interactionFlowModel
```
(4.5)

```
https://example.com/rest/IFMLModel/AddMovie/interactionFlowModel
```
(4.6)

Depending on the cardinality of the reference this will return a specific element – if it is single-valued (like in the case of `interactionFlowModel`) – or a collection of elements – if it is multi-valued. Accessing a specific element contained in a collection can be done using (i) the identifier of the element or (ii) its index in the list. Also, when navigating through the references contained in elements being subclasses of a hierarchy, the appropriate filtering is done on the fly. For instance, the URI in Expression (4.7) retrieves the element representing *title* in EMF, while in EMF-REST it is done using the request in Expression (4.8). Note how the latter navigates through the reference `viewElements`, which is only included in *ViewContainer* element. To identify an element, we rely on the *identifier* flag provided by *Ecore*, which allows setting the attribute acting as identifier for a given class[1].

```
platform:/resource/project/AddMovie.xmi#title
```
(4.7)

```
https://example.com/rest/IFMLModel/AddMovie/
interactionFlowElements/AddMovieWindow/viewElements/
AddMovieForm/viewComponentsParts/title
```
(4.8)

On the other hand, the request in Expression (4.9) will retrieve the first element of the collection of *viewComponentsParts* in the EMF API. In our approach, it is done by adding the parameter *index* in the URL as illustrated in the request in Expression (4.10).

```
platform:/resource/project/AddMovie.xmi\#//.../@
viewComponentsParts.0
```
(4.9)

```
https://example.com/rest/IFMLModel/AddMovie/
interactionFlowElements/AddMovieWindow/viewElements/
AddMovieForm/viewComponentsParts?index=0
```
(4.10)

4.3.2 *Manipulation of Resources Through Representations*

By default, EMF persists models using the XMI representation format. Our approach relies also on XMI to save the models internally, however,

[1]When the *identifier* flag is not used, the fallback behavior looks for an attribute called *id*, *name* or having the *unique* flag activated.

Listing 4.1: Partial JSON representation of the example model

```
{
  "form":{
    "name":"addMovieForm",
    "viewComponentParts":{
      "simpleField":[{
        "uri":"https://example.com/rest/IFMLModel/AddMovie/
            interactionFlowElements/AddMovieWindow/viewElements/
            AddMovieForm/viewComponentsParts/picture"},{
        "uri":"https://example.com/rest/IFMLModel/AddMovie/
            interactionFlowElements/AddMovieWindow/viewElements/
            AddMovieForm/viewComponentsParts/title"},...],
      ...
    },
    "viewElementEvents":{
      "onsubmitevent":{"uri":"https://example.com/rest/IFMLModel/
          AddMovie/interactionFlowElements/AddMovieWindow/
          viewElements/AddMovieForm/viewelementevents/add"}
    }
  }
}
```

models are offered to clients using both JSON and XML formats in order to comply with the representation-oriented constraint of the REST architecture (see Section 2.1 in Chapter 2).

For JSON, we adhere to the following structure. Model concepts are represented as JSON objects containing key/value pairs for the model attributes/references. Keys are the name of the attribute/reference of the concept and values are their textual representation in one of the data types supported in JSON (i.e., string, boolean, numeric, or array). For attributes, their values are mapped according to the corresponding JSON supported data type or String when there is not a direct correspondence (e.g., float-typed attributes). When the attribute is multi-valued, its values are represented using the array data type. For references, the value is the URI of the addressed resource within the server (if the reference is multi-valued, the value will be represented as an array of URIs). Listing 4.1 shows an example of the content format in JSON. Note that references containing a set of elements from model hierarchies are serialized as a list of JSON objects corresponding to their dynamic type (see `viewComponentParts` reference including `SimpleField` and `SelectionField` JSON objects).

In XML, model concepts are represented as XML elements including an XML element for each model attribute/reference. Attribute values are included as string values in the XML element representing such attribute, references are represented according to their cardinality. If the reference

Listing 4.2: Partial XML representation of the example model

```
<form>
  <name>AddMovieForm</name>
  <viewComponentParts>
    <simpleField>
      <uri>https://example.com/rest/IFMLModel/AddMovie/
          interactionFlowElements/AddMovieWindow/viewElements/
          AddMovieForm/viewComponentsParts/picture</uri>
    </simpleField>
    <simpleField>
      <uri>https://example.com/rest/IFMLModel/AddMovie/
          interactionFlowElements/AddMovieWindow/viewElements/
          AddMovieForm/viewComponentsParts/title</uri>
    </simpleField>
    ...
  </viewComponentParts>
  <viewElementEvents>
    <onSubmitEvent>
      <uri>https://example.com/rest/IFMLModel/AddMovie/
          interactionFlowElements/AddMovieWindow/viewElements/
          AddMovieForm/viewElementEvents/add</uri>
    </viewElementEvents>
  </viewElementEvents>
</form>
```

is single-valued, the resulting XML element will include only the URI of the addressed resource in the server. On the other hand, if the reference is multi-valued, the resulting XML element will include a set of XML elements including the URIs addressing the resources. Listing 4.2 shows an example of the content format in XML format.

4.3.3 *Uniform Interface*

EMF supports loading, unloading and saving model instances after their manipulation. In our approach, these operations are managed by the application server. Models are loaded (and unloaded) dynamically as resources when running the application managing the Web API, and they are saved after each operation, thus conforming to the REST statelessness behavior.

To manipulate model instances, EMF enables the basic CRUD (i.e., Create, Read, Update and Delete) operations over model instances by means of either the EMF generated API or the *EObject* API. We map the same CRUD operations into the corresponding HTTP methods (*POST*, *GET*, *PUT*, and *DELETE*). For instance, Listing 4.3 shows the code to modify the name of the form called *AddMovieForm* using EMF generated API for

Listing 4.3: Update the attribute of a concept using EMF generated API.

```
...
addMovieFormObj.setName("toto"); // addMovieFormObj is of type Form
...
```

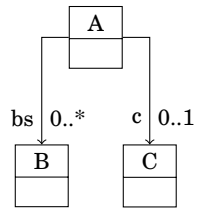
Listing 4.4: HTTP request and JSON representation to update the name of the addressed form.

```
1 PUT https://example.com/rest/IFMLModel/AddMovie/
      interactionFlowElements/AddMovieWindow/viewElements/AddMovieForm
2 {"form":{
3          name:"toto"
4          }
5 }
```

the *AddMovie* model. The same operation can be done on our Web API by sending the PUT HTTP request containing the JSON representation of the new *Form* model element, as shown in Listing 4.4.

Table 4.1 shows how each CRUD operation is addressed along with several URL examples. The first column of the table describes the operations.

Table 4.1: Supported operations in the generated API.

OPERATION	HTTP METHOD	URL	MODEL
CREATE and add element to the collection	POST	`.../a/bs`	
READ all the elements from the collection	GET		
READ the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	GET		
UPDATE the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	PUT	(1) `.../a/bs/<id>` (2) `.../a/bs?index=<i>` (3) `.../a/c`	
DELETE the element (1) identified by <id>, (2) in the <i> position of the collection, or (3) the element c	DELETE		

As can be seen, the first two rows represent operations over collections, enabling adding new elements (see first row) and reading their content (see second row). The rest of the rows describe operations over either individual elements of a collection (see cases 1 and 2 of these operations) or elements contained in a single-valued reference (see case 3). The second column shows the correspondent HTTP method for each operation while the third column presents the corresponding URL for each case. Finally, the last column includes a small model to better illustrate the cases considered in the table.

4.4 Additional EMF-REST Features

We provide also support for validation and security aspects in the generated REST Web API.

4.4.1 *Validation*

Support for validating the API data calls is pretty limited in current Web technologies. The most relevant one for our scenario would be the Bean Validation specification to enforce the validation of *Java Beans*. However, this specification can only ensure that fields follow certain constraints (e.g., a field is not `null`) and cannot satisfy complex validation scenarios for model integrity (e.g., a *form* must have at least one *field*). To cope with this issue, we rely on OCL [WK99] to define constraints as annotations in the model elements. OCL, which was presented in Chapter 2 (see Section 2.3), provides a declarative language to create queries and constraints over MOF-based models. We employ OCL to define constraints as annotations in the model elements.

OCL annotations can be attached to concepts in the model as invariants. An example on the IFML example model is shown in Figure 4.4. As can be seen, concepts include a set of invariants inside the annotation *OCL* plus the annotation *Ecore / constrains* which specifies the invariants to execute. Invariants are checked each time a resource is modified (i.e., each time the Web API is called from a Web-based client using the *POST*, *PUT* or *DELETE* methods). This validation scheme is imposed to comply with the stateless property of REST architectures, however, it may involve some design constraints when creating the model. In those cases where models cannot be validated each time they are modified (e.g., creating model elements requires several steps to fulfill cardinality constraints), we allow this validation process to be temporary deactivated. The results of the validation

process are mapped into the corresponding HTTP response messages (i.e., using status codes).

4.4.2 *Security*

While there is little support for security definition and enforcement from the MDE side, we have plenty of support from web technologies. In particular, our approach allows designers to provide some security annotations on the model that are then translated into security restrictions as described below. As part of the generation, we also create a separated admin view where additional security information (like users and passwords) can be maintained.

In order to secure a Web application, we have to: (i) ensure that only authenticated users can access resources, (ii) ensure the confidentiality and integrity of data exchanged by the client/server, and (iii) prevent unauthorized clients from abusing data. In order to address the previous requirements, we rely on a set of security protocols and services provided by *Java EE* which enable encryption, authentication and authorization in Web APIs, as we will explain in the following.

ENCRYPTION: The Web defines Hypertext Transfer Protocol Secure (HTTPS) protocol to add the encryption capabilities of Secure Sockets Layer (SSL)/Transport Layer Security (TLS) to standard HTTP communication. We enforce the use to HTTPS to communicate with the API.

AUTHENTICATION: We rely on basic authentication to provide the authentication mechanism since it is simple, widely supported, and secure by using HTTPS. The basic authentication involves sending a Base64-encoded username and password within the HTTPS request header to the server.

AUTHORIZATION: While the authentication is enabled by the protocol/server, the authorization is generally provided by the application, which knows the permissions for each resource operation. We use a simple role-based mechanism to support authorization in the generated Web API. Roles are associated to users (i.e., authentication) and operations in the Web API (i.e., authorization). In our approach roles are assigned to resources by adding annotations to the model. Fig-

Figure 4.4: Annotations on an excerpt of the example model.

ure 4.4 illustrates the use of these annotations (e.g., see annotation *Ecore*/*roles* in the *Form* concept).

4.5 EMF-REST API ARCHITECTURE

To implement the features described in the previous sections, we devised the application architecture presented in Figure 4.5. This architecture can then be seamlessly accessed with a variety of clients.

The Web application is split into three main components according to the functionality they provide: (1) content management, (2) validation and (3) security. The application relies on EMF as modeling framework and uses the following additional frameworks/specifications for each component, respectively: (1) Java Architecture for XML Binding (JAXB) to enable the content format support, (2) Eclipse OCL framework to provide validation before updating the model, (3) Java Persistence API (JPA) to provide security support by storing the system users and their permissions in an embedded database. The Web application also leverages on Enterprise Java Bean (EJB), Context Dependency Injection (CDI) and Java API for Representational State Transfer (JAX-RS) specifications. EJBs enable rapid and simplified development of distributed, transactional, secure and portable applications. They are in charge of loading the EMF resources from the persistent storage and providing the necessary methods to manage the resources (e.g., obtaining objects from the resource, removing objects) in a secure and transactional way. These EJBs are then injected into JAX-RS services using CDI technology. Thus, JAX-RS is used to expose EMF resources as Web services. In the remaining of the section we describe how all these technologies are used in each component.

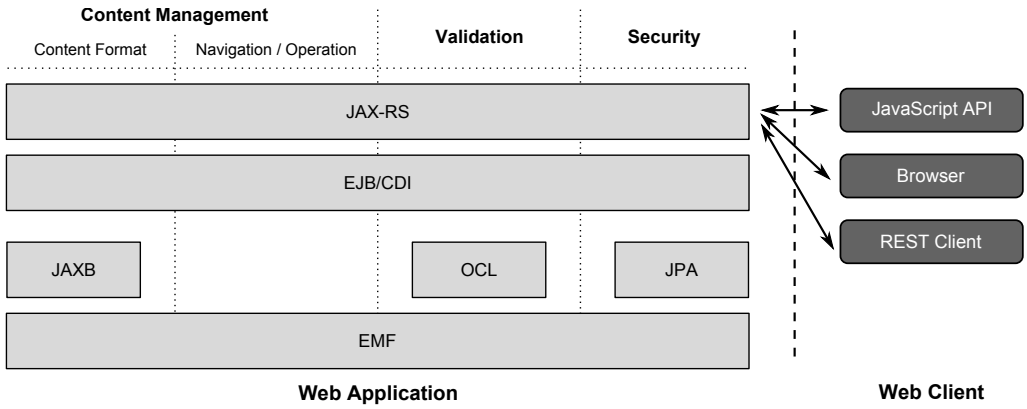


Figure 4.5: Architecture of the generated application.

4.5.1 *Content Management*

This component addresses the mapping between EMF and REST. It is in turn split into two subcomponents: (i) content format, and (ii) navigation/operation.

Regarding the content format, we enrich the EMF generated API with JAXB[2] annotations, which enable the support for mapping Java classes to XML/JSON (i.e., marshalling/unmarshalling Java object into/from XML/JSON documents). Listing 4.5 shows an example of the use of JAXB annotations to produce the corresponding representation in JSON (as shown in Listing 4.1) and XML (as shown in Listing 4.2). As can be seen, each concept class is mapped to an `XmlRootElement` element, while either `XmlElement` or `XmlElementWrapper` elements are used to map the attributes or references of the class, respectively. Other annotations are used to deal with the references and inheritance. For instance, `XmlJavaTypeAdapter` and `XmlAnyElement` are used to associate a reference of an element with the corresponding representation.

Navigation and operations are enabled by using JAX-RS, which provides a set of Java APIs for building Web services conforming to the REST style. Thus, this specification defines how to expose POJOs as Web resources, using HTTP as network protocol. For each concept (e.g., `IFMLModel`) a resource will be created (e.g., `IFMLModelResource`) annotated with `@Path` (e.g., `@Path("IFMLModel")`). The `@Path` annotation has the value that represents the relative root URI of the addressed resource. For instance, if the base URI of the server is `http://example.com/rest/`, the resource will be available

[2] `https://jaxb.java.net/`

Listing 4.5: Part of the `ViewComponent` concept.

```
@XmlRootElement (name="viewcomponent")
@XmlSeeAlso ({ViewComponentProxy.class,//...
})
public  class ViewComponentImpl extends ViewElementImpl
  implements ViewComponent {
  //...
  @XmlElementWrapper(name =  "viewComponentParts")
  @XmlAnyElement(lax=true)
  @XmlJavaTypeAdapter(value=ViewComponentPartAdapter.class)
  public EList<ViewComponentPart> getViewComponentParts() {
    if (viewComponentParts == null) {
      viewComponentParts = new
          EObjectContainmentWithInverseEList<ViewComponentPart>(
          ViewComponentPart.class, this,
          IfmlPackage.VIEW_COMPONENT__VIEW_COMPONENT_PARTS,
          IfmlPackage.VIEW_COMPONENT_PART__VIEW_COMPONENT);
    }
    return viewComponentParts;
  }
  //...
}
```

under the location `http://example.com/rest/IFMLModel`. To produce a particular response when a request with GET, PUT, POST and DELETE is intercepted by a resource, resource methods are annotated with `@GET`, `@PUT`, `@POST` and `@DELETE` what are invoked for each corresponding HTTP verb.

4.5.2 *Validation*

Our approach leverages on Eclipse OCL[3] to validate the data by means of annotations including the constrains to check the model elements. The generated API relies on the provided APIs for parsing and evaluating OCL constraints and queries on *Ecore* models. When constraints are not satisfied, the validation process will fire an exception that will be mapped by JAX-RS into an HTTP response including the corresponding message indicating the violated constraint.

4.5.3 *Security*

We rely on the combination of Java EE and JAX-RS for the authentication and authorization mechanisms by using the concept of role, while encryiption is provided by using HTTPS. To enable authentication, the deployment descriptor of the WAR file (i.e., `WEB-INF/web.xml`) has been modified to include the security constraints (i.e., `<security-constraint>`)

[3] `http://www.eclipse.org/modeling/mdt/?project=ocl`

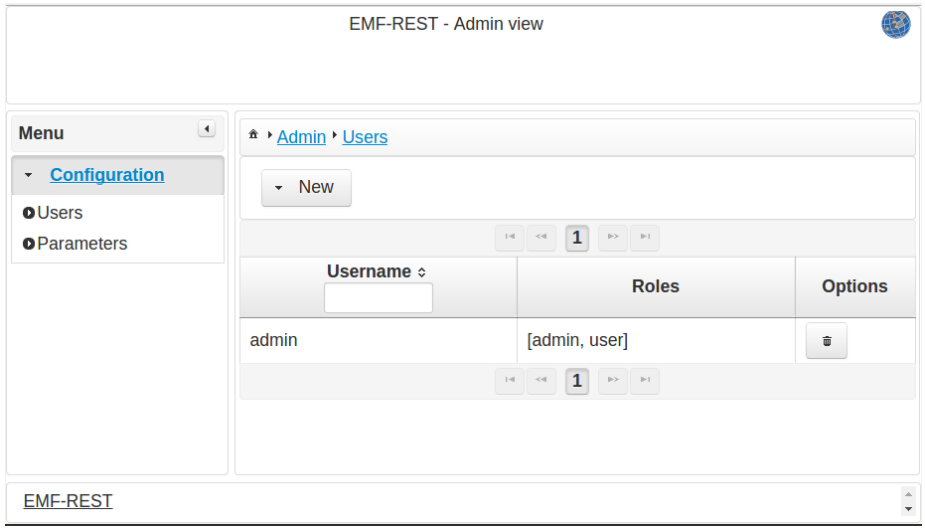


Figure 4.6: EMF-REST screenshot: admin view.

defining the access privileges. Assigning permissions for HTTP operations based on the roles provided in the model is done by using the `@RolesAllowed` annotation. For example, as shown before, Figure 4.4 shows that the role allowed for the *Form* concept is `admin`.This will restrict access to the resource to the users having the role `ADMIN`. To express this in the generated API, the annotation `@RolesAllowed({"ADMIN"})` is placed on top of `FormResource`. If no role is assigned to a concept, a `@PermitAll` annotation is placed on the resource class meaning that all security roles are permitted to access this resource. Note that security roles assigned to a resource are not inherited by its sub-resources.

To manage the list of users and their roles, we generate an admin view that allows the manager of the API to add, edit and remove users. All created users have a default role (i.e., *user*) allowing them to access unannotated concepts. The manager can assign more roles to a user in order to grant him/her access to a specific resource. Figure 4.6 shows a screenshot of the generated admin view to manage users and roles.

4.6 Code Generation and Tool Support

In order to generate the REST APIs we created a Java tool available as an Open Source Eclipse plugin[4]. Figure 4.7 shows the steps followed by the tool to generate the application starting from an initial *Ecore* model.

[4] `https://som-research.uoc.edu/tools/emf-rest/`

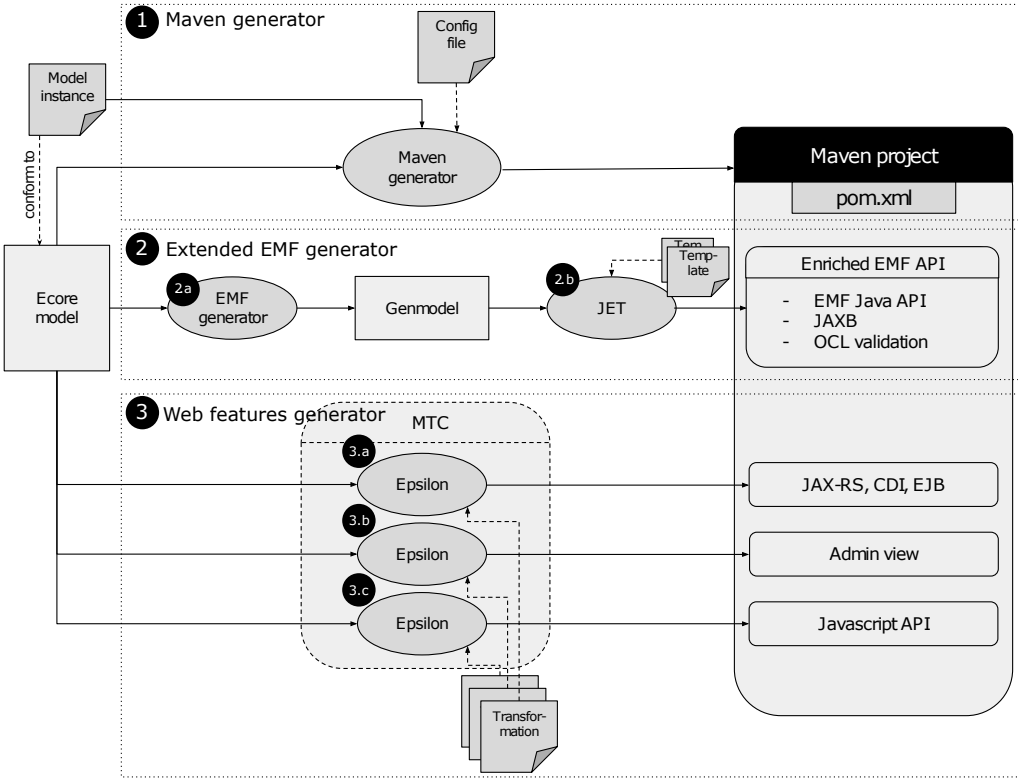


Figure 4.7: EMF-REST generation process.

Step 1 of the process generates a Maven-based[5] project that serves as a skeleton of the application. Maven allows a project to be built by using the Project Object Model (POM) file, thus providing a uniform build system. The POM is initialized with the required library dependencies described in the previous section.

Step 2 describes *the extended EMF generator*. The EMF code generation facility has been extended to include the required support for JAXB and validation. In particular, the JET templates used by EMF to generate Java code have been extended to produce the code corresponding to the JAXB annotations and the required methods to execute the OCL validation process.

Step 3 describes *the extended EMF generator*. This step performs a set of model-to-text transformations using Epsilon Generation Language (EGL) to generate the required Web features elements, including: (1) the JAX-RS, CDI and EJB implementation classes, (2) the admin view developed and (3)

[5] `http://maven.apache.org/`

a simple JavaScript API to facilitate Web developers to build clients for the generated Web API. For each part of the application (e.g., JAX-RS resources, etc.), an EGL transformation template has been implemented to generate the appropriate behavior according the input *Ecore* class. Since this step requires several transformations, the MTC tool [AC13] has been used to orchestrate the flow of the EGL templates.

4.7 Related Work

Several efforts have been made to bring together MDE and Web Engineering. This field is usually referred to as Model Driven Web Engineering (MDWE) and proposes the use of models and model transformations for the specification and semiautomatic generation of Web applications [SWK06; KK12; QN10; CFB00; MCG10]. Mainly, data models, navigation models and presentations models are used for this purpose.

Some of these works provide support for the generation of Web services as well, but support for generation of REST APIs is limited [Riv+13a; TV13a; Max+07; Zol+17; Hau+14; Ter+17]. Moreover, these approaches require the designer to specifically model the API itself using some kind of tool-specific DSL from which then the API is (partially) generated. To the best of our knowledge, only Texo[6] provides an extension to EMF in order to support the creation of REST APIs but providing a proprietary solution which requires extending the user's metamodel with meta-data to drive the API generation. Instead, our approach is able to generate a complete REST API from any metamodel.

4.8 Summary

In this chapter we have presented EMF-REST, an approach to generate REST APIs out of EMF models which we illustrated using the IFML metamodel. We described how we map EMF concepts to REST and the process of generating REST APIs to manage EMF models. The generated APIs rely on well-known libraries and standards, and also provide extra features such as validation and security. We believe our approach fills an important gap between modeling and Web technologies. In particular, our approach enables MDE practitioners to add new capabilities to their modeling process such as collaboration.

[6] `https://wiki.eclipse.org/Texo`

CHAPTER

5 Discovering REST APIs Specifications

In the previous chapter we have presented an approach to *APIfy* EMF models by generating REST APIs to manage such models. These generated APIs do not provide a specification to describe its features but rely on their *Ecore* definitions. In fact, despite their popularity, REST APIs do not typically come with any specification of the functionality they offer, thus hindering their integration. This chapter aims at improving this situation by presenting APIDISCOVERER, an approach to automatically discover REST API specifications, in particular the OpenAPI one, from API call examples.

The remainder of this chapter is structured as follows. Section 5.1 describes the running example. Section 5.2 presents the overall approach and then Sections 5.3 and 5.4 describe the discovery process and the generation process, respectively. Section 5.5 describes the validation process and limitations of the approach. Section 5.6 presents the related work. Section 5.7 describes the tool support, and finally, Section 5.8 summarizes this chapter.

5.1 RUNNING EXAMPLE

This section introduces the running example we will use to illustrate our approach. The example is based on the Petstore API introduced in Chapter 2 (see Section 2.2). The Expression 5.1 shows the request to retrieve the pet with the identifier 123 while Listing 5.1 shows the returned response with

Listing 5.1: JSON representation of a Pet instance.

```
{
  "id": 123,
  "category": {
    "id": 1,
    "name": "dogs"
  },
  "name": "doggie",
  "photoUrls": ["http://example.com"],
  "tags": [
    {
      "id": 1,
      "name": "black"
    }
  ],
  "status": "available"
}
```

status code 200 including that pet information.

$$\underbrace{\texttt{GET}}_{\text{HTTP method}} \quad \underbrace{\texttt{http}}_{\text{protocol}} \texttt{://} \underbrace{\texttt{petstore.swagger.io}}_{\text{host}} \quad \underbrace{\texttt{/v2}}_{\text{basePath}} \quad \underbrace{\texttt{/pet/123}}_{\text{relativePath}} \tag{5.1}$$

A request includes a method (e.g., `GET`), a URL (e.g., `http://petstore.swagger.io/v2/pet/123`) and optionally a message body (empty for this example). The URL in turn includes: (i) the transfer protocol, (ii) the host, (iii) the base path, (iv) the relative path, and (v) the query (indicated by the first question mark, empty for this example). The relative path and the query are optional. A response includes a status code (e.g., 200) and optionally a JSON response message.

Listing 5.2 shows a snippet of the OpenAPI definition of the Petstore API showing the operation `getPetById` which corresponds to this call example. As can be seen, the specification indicates that the `GET` operation of the path `/pet/{petId}` allows retrieving a pet by his ID. This operation includes one path parameter (i.e., `petId`) and returns an instance of `Pet`. Our goal is to analyze the call example 5.1 to infer the OpenAPI definition reflecting the behavior the Petstore API.

Listing 5.2: A snippet of the OpenAPI definition of the Petstore API showing the `getPetById` operation.

```
{
   "swagger":"2.0",
   "host":"petstore.swagger.io",
   "basePath":"/v2",
   "schemes": ["http"],
   "paths":{
      "/pet/{petId}"":{
         "get":{
            "parameters":[
               {"name":"petId",
               "in":"path"
               ,...}],
            "responses":{
              "200":"schema":{
                "$ref":"#/definitions/Pet"},
                ...},
         },...
      },...
   },
   "definitions":{
      "Pet":{
         "type":"object",
         "properties":{
            "id":{
               "type":"integer",
               ...},
            "category":{
              "$ref":"#/definitions/Category"},
            "name":{
               "type":"string",
               ...},
               ...
      },...
   },
   ...
  }
}
```

5.2 Our Approach

We define a two-step process to discover OpenAPI definitions from a set of API call examples. Figure 5.1 shows an overview of our approach.

The process takes as input a set of API call examples. For the sake of simplicity, we assume examples are provided beforehand and later in Section 5.7 we describe how we devised a solution to provide them both manually and relying on other sources. These examples are used to build an OpenAPI model (see Figure 5.1a) in the first step of the process. Each example is

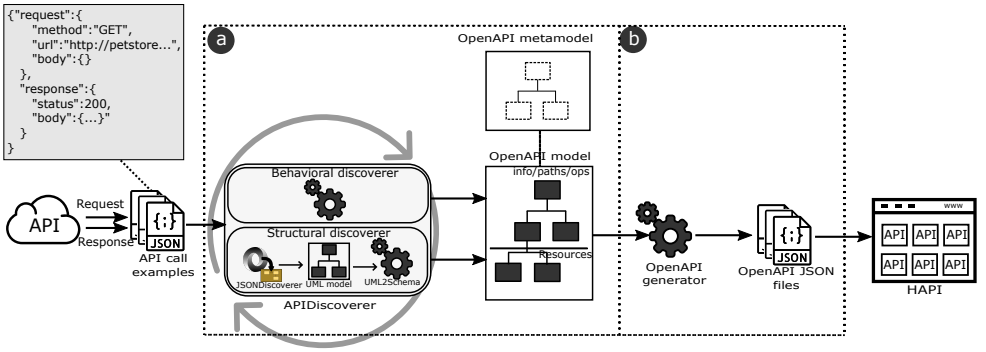


Figure 5.1: The APIDISCOVERER approach.

analyzed with two discoverers, namely: (i) behavioral and (ii) structural; targeting the corresponding elements of the API definition. The output of these discoverers is merged and added incrementally to an OpenAPI model, conforming to the OpenAPI metamodel presented in Section 3.1. The second step transforms these OpenAPI models to valid OpenAPI JSON documents (see Figure 5.1b).

To represent the API call examples, we rely on a JSON-based representation of the request/response details. Both, the request and the response messages are represented as JSON objects (i.e., `request` and `response` fields in left upper box of Figure 5.1). The request object includes fields to set the method, the URL and the JSON message body; while the response object includes fields for the status code and the JSON response message. This JSON format helps simplify the complexity of directly using raw HTTP requests and responses (which would require to perform HTTP traffic analysis) and facilitate the provision of examples. As we will discuss later, we also provide tool support to provide API call examples and even to (semi)automatically derive them from other sources, like existing documentation.

As a final step, the resulting OpenAPI documents may optionally be added to HAPI, our community-driven hub for REST APIs, where developers can search and query them. In the following sections we describe the discovery process. The example providers, APIs importers, and HAPI will be explained in Section 5.7.

5.3 The Discovery Process

The discovery process takes as input a set of API call examples and incrementally generates an OpenAPI model conforming to our OpenAPI metamodel using two types of discoverers: (i) behavioral and (ii) structural. The former generates the behavioral elements of the model (e.g., paths, operations) while the latter focuses on the data types elements. In the following we explain the steps followed by these two discoverers.

5.3.1 *Behavioral Discoverer*

The behavioral discoverer analyzes the different elements of the API call examples (i.e., HTTP method, URL, request body, response status, response body) to discover the behavioral elements of the metamodel.

Table 5.1 shows the applied steps. *Target elements* column displays the created/updated elements in the OpenAPI model while *Source* column shows the elements of an API call example triggering those changes. The *Action* column describes the applied action at each step and the *Notes* column displays notes for special cases. These steps are applied in order and repeated for each API call example. A new element is created only if such element does not already exist in the OpenAPI model. Otherwise, the element is retrieved and enriched with the new discovered information. Note that the discovery of the schema structure will be assessed by the structural discoverer (see step 6).

Figure 5.2 shows the generated OpenAPI model for the API call in the running example. The discovery process is applied as follows. Step 1 creates an `API` element and set its attributes (i.e., `schemes` to `SchemeType::http`, `host` to `petstore.swagger.io`, and `basePath` to `/v2`). Step 2 creates a `Path` element, sets its only attribute `realtivePath` to `/pet/{petId}` (the string `'123'` was detected as identifier), and adds it to the `paths` references of the `API` element. Step 3 creates an `Operation` element, sets its `produces` attribute to `application/json`, and adds it to the `get` reference of the previously created `Path` element. Step 4 creates a `Parameter` element, sets its attributes (i.e., `name` to `petId`, `location` to `path`, and `type` to `JSONDataType::integer`), and adds it to the `parameters` reference of the previously created `Operation` element. Step 5 creates a `Response` element, sets its attributes (i.e., `code` to `200` and `description` to `OK`), and adds it to the `response` reference of the previously created `Operation` element. Finally step 6 creates a `Schema` element, sets only its name to `Pet`, and adds it to the `definitions` reference of the `API` element. The rest of the `Schema`

Table 5.1: APIDISCOVERER: steps of the behavioral discoverer applied for each REST API call example.

STEP	Source	Target elements	ACTION	NOTES
1	*<host>*, *<basePath>*, *<protocol>*	`a:API`	-`a.schemes`= *protocol* -`a.host`= *host* -`a.basePath`= *basePath*	If the path contains many sections (e.g., `/one/two/...`) the base path is set to the first section (e.g., `/one`) otherwise it is set to "/".
2	*<relativePath>*	`pt:Path`	-Add `pt` to `a.paths` -`pt.relativePath`= *relativePath*.	If *relative path* contains an identifier, it is replaced with a variable in curly braces to use path parameters. A pattern-based approach is used to discover identifiers.[a].
3	*<httpMethod>*, *<RequestBody>*, *<ResponseBody>*	`o:Operation`	-`pt.{httpMethod}`= `o` -If *requestBody* is of type JSON then add `"application/json"` to `o.consumes` otherwise keep `o.consumes` empty. -If *responseBody* is of type JSON then add `"application/json"` to `o.produces` `o.produces` otherwise keep `o.consumes` empty.	*{httpMethod}* is the reference of `pt` which corresponds to *<httpMethod>* (e.g., *get* or *post*).
4	*<query>*, *<relativePath>*, *<requestBody>*	`pr:Parameter`	-Add `pr` to `o.parameters` -Set `pr.type` to the inferred type[b] -Set `pr.location` to: (i) `path` if parameter is in *relativePath* (ii) `query` if parameter is in *query* (iii) `body` if parameter is in *requestBody*	Apply this rule for all the detected parameters. The discovery of the schema of the body parameter is launched in step 6
5	*<ResponseCode>*	`r:Response`	-Add `r` to `o.responses` -`r.code`= *responseCode* -`r.description`= correspondent description of the response.	The discovery of the schema of the response body is launched in step 6.
6	*<RequestBody>*, *<ResponseBody>*	`s:Schema`	-Add `s` to `a.definitions`. -Set the `s.name`= the last meaningful section of the path. -If the schema is in *<RequestBody>*, set `pr.schema` to `s` where `pr` is the body parameter created in step 4. -If the schema is in *<ResponseBody>*, set `r.schema` to `s` where `r` is the response created in step 5. -Launch the structural discoverer.	We apply this rule only if *requestBody* or *responseBody* contains a JSON object.

[a] We apply an algorithm which detects if a string is a UID (e.g., hexadecimal strings, integer).

[b] When a conflict is detected (e.g., a parameter was inferred as integer and then as string), the most generic form is used (e.g., string).

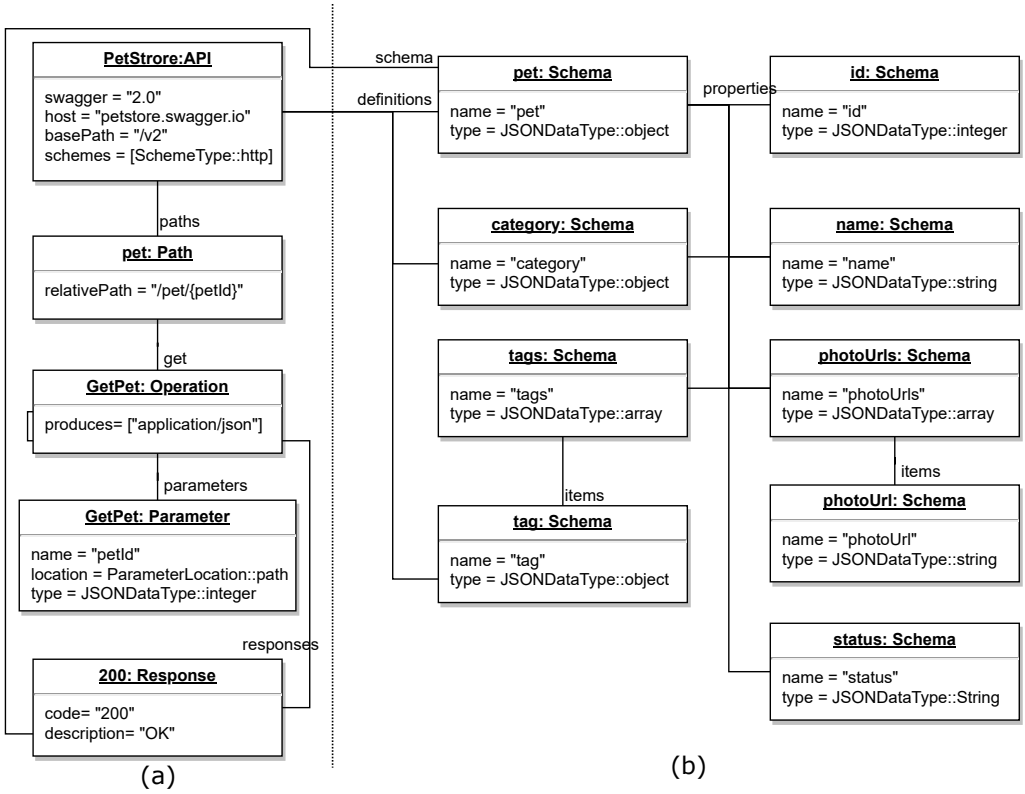


Figure 5.2: APIDISCOVERER: the discovered OpenAPI model for the Petstore API example: (a) behavioral discovery, (b) structural discovery.

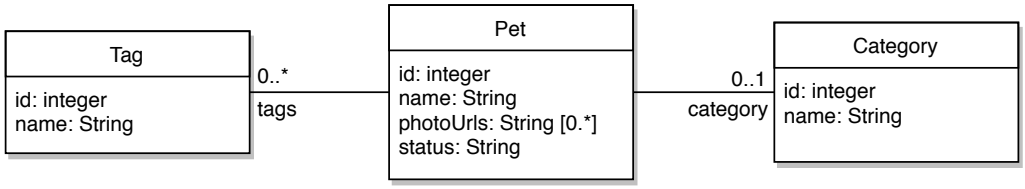


Figure 5.3: Generated UML model from the API call example.

element will be completed by the structural discoverer.

5.3.2 *Structural Discoverer*

The structural discoverer instantiates the part of the OpenAPI model related to data types and schema information. This process is started after the behavioral discovery when the API call includes a JSON object either

Table 5.2: Transformation rules from UML to Schema

SOURCE	TARGET: CREATE	TARGET: UPDATE	ATTRIBUTES INITIALIZATION
`Class`	`c: Schema`	- Add `c` to the `api.definitions`.	- `c.type` = `Object` - `c.name` = The corresponding class name
`Attribute (1)`	`a: Schema`	-Add `a` to `c.properties` where `c` is the correspondent schema of the class containing the attribute.	- `a.type` = the `JSONDataType` correspondent to the type of the attribute - `a.name` = the attribute name
`Attribute (*)`	`a: Schema, i: Schema`	-Add `a` to `c.properties` where `c` is the correspondent schema of the class containing the attribute.	- `a.type` = `array` - `a.items`= `i` - `i.type`= the `JSONDataType` correspondent to the type of the attribute - `i.name` = the attribute name
`Association (1)`	-	-Add `tc` to `c.properties` where `c` is the correspondent schema of the source class of the association and `tc` the correspondent schema of the target class of the association.	-
`Association (*)`	`a: Schema`	-Add `a` to `sc.properties` where `sc` is the correspondent schema of the source class of the association	- `a.type` = `array` - `a.items`= `tc` where `tc` is the correspondent schema of the target class of the association.

in the request body or the response message that will be used to enrich the definition of the discovered `Schema` elements.

We devised a two-step process where we first obtain an intermediate UML-based representation from the JSON objects and then we perform a model-to-model transformation to instantiate the actual schema elements of the OpenAPI metamodel. This intermediate step allows us to benefit from JSONDISCOVERER [CC16], which is the tool used to build a UML class diagram, and to use this UML-based representation to bridge easily to other model-based tools if needed. Then, classes, attributes, and associations of the UML class model are transformed to `Schema` elements. Table 5.2 shows the transformation rules applied to transform UML models to `Schema` elements. *Source* column shows the source elements in a UML model while *Target: create* and *Target: update* columns display the created/updated elements in the OpenAPI model. The *Attribute initialization* column describes the transformation rules.

Note that elements are updated/enriched when they already exist in the OpenAPI model. This particularly happens when different examples represent the same `schema` elements, as JSON schema allows having optional

parts in the examples.

Figure 5.3 shows the UML class model discovered by JSONDISCOVERER for the API response shown in Listing 5.1. This class model is transformed to actual `Schema` elements applying the discovery process as follows. `Tag`, `Pet`, and `Category` classes are transformed to `schema` elements of type `Object`. Single-valued attributes (e.g., `name`, `id`) are transformed to `Schema` elements where `type` is set to the corresponding primitive type. The `photoUrls` multivalued attribute and `tags` multivalued association are transformed to `Schema` elements of type `array` having as `items` a `Schema` element of type `String` and `Tag`, respectively. Finally, attributes and associations are added to the `properties` reference of the corresponding `Schema` element.

5.4 THE GENERATION PROCESS

The generator creates an OpenAPI-compliant JSON file from an OpenAPI model by means of a model-to-text transformation. This generator was previously presented in Chapter 3 alongside the OpenAPI metamodel. As said in Section 3.1, elements such as `Schema`, `Parameter`, and `Response` can be declared in different locations and reused by other elements. While the `declaringContext` reference is used to define where to declare the object, the `ref` attribute (inherited form `JSONPointer` class) is used to reference this object from another element. By default the discovery process sets the declaring context to the containing class of the element (e.g., parameters in operations).

Listing 5.3 shows the generated JSON file for the OpenAPI model shown in Figure 5.2. Note that the declaring context of the `Pet` schema element is set to `API`, which resulted in listing the `Pet` element in the `definitions` object. Consequently, the attribute `ref` is set to `#/definitions/Pet` and will be used to reference `Pet` from any another element (as in the `response` object).

5.5 VALIDATION AND LIMITATIONS

To ensure the quality of the OpenAPI definitions we generate, we have first enriched the OpenAPI metamodel with a set of well-formedness constraints written in OCL (e.g., to guarantee the uniqueness of the parameters in a call). These constraints are checked during the discovery process to validate the generated OpenAPI specification against the constraints published in the official OpenAPI specification document. Note that this is in itself a

Listing 5.3: The generated OpenAPI definition of the Petstore call example.

```
{ "swagger":"2.0",
   "info":{},
   "host":"petstore.swagger.io",
   "basePath":"/v2",
   "tags":[ "pet" ],
   "schemes":[ "http" ],
   "paths":{
         "/pet/{petId}":{
           "get":{
            "produces":["application/json"],
            "parameters":[{"name":"petId","in":"path","type":"
               integer"}],
            "responses":{
               "200":{
                  "description":"OK",
                  "schema":{"$ref":"#/definitions/Pet"
                  }}}}
   }
   },
   "definitions":{
      "Pet":{
         "type":"object",
         "properties":{
            "id":{"type":"integer"},
            "category":{
              "$ref":"#/definitions/Category"
                  },
            "name":{
              "type":"string"
              },
            "photoUrls":{
              "type":"array",
              "items":{
                "type":"string"
                }
              },
            "tags":{
              "type":"array",
              "items":{
                "$ref":"#/definitions/Tag"
                }
              },
            "Status":{
              "type":"string"
              }
            }
      },
      ...
    }
  }
```

useful contribution with regard to other syntax checkers for API documents that offer a limited support in terms of constraint checking.

Additionally, we have validated our approach by manually comparing the results of our generated OpenAPI with the original specification for a number of APIs providing already such information. This has been an iterative process but we would like to highlight the latest tests, comprising the following five APIs: (i) REFUGE RESTROOMS[1], a web application that seeks to provide safe restroom access for transgenders; (ii) OMDB[2], an API to obtain information about movies; (iii) GRAPHHOPPER[3], a route optimization API to solve vehicle routing problems; (iv) PASSWORDUTILITY[4], an API to validate and generate passwords using open source tools; and finally (v) the Petstore API. Several factors influenced the choice of these APIs to serve for our testing purposes. Beside having an OpenAPI specification, these APIs did not involve fees or invoke services (e.g., SMS APIs), they managed JSON format (to test our structural discoverer) and were concise (to keep limited the number of examples required).

For these APIs, our approach was able to generate on average 80% of the required specification elements and did not generate any incorrect result. Mainly, the missing information was due to the structure of the call examples which cannot cover advanced details such as: (i) the enumerations used for some parameters, (ii) the optionality or not of the parameters, (iii) form parameters, and (iv) the headers used in some operations. Furthermore, the quality of the results depend on the number and the variety of the API call examples used to discover the specification. Our experience so far shows that the number of examples should be higher than the number of operations of an API covering all the parameters. However, more experiments are required to identify the ideal balance between the quality of the result and the number of needed experiments.

Note that even if the result is not complete, it is still useful. Even for APIs that do provide an OpenAPI as starting point. For instance, for Refuge Restrooms, we were able to discover both the operations and data model of the API even if the latter was not part of the original specification. The complete set of examples and generated APIs are available in our repository[5].

[1] `http://www.refugerestrooms.org/api/docs/`
[2] `http://www.omdbapi.com/`
[3] `https://graphhopper.com/`
[4] `http://passwordutility.net`
[5] `https://github.com/SOM-Research/APIDiscoverer`

5.6 Related Work

Several tools supporting the OpenAPI initiative have been developed, thus making making the OpenAPI specification a more valuable artifact. Tools such as Swagger UI[6] and ReDoc[7] can be used to generate document pages. Other tools such as Swagger Editor[8] and Restlet Studio[9] helps developers manually design APIs with visual tools and generate SDKs for different platforms. Regarding the discovery, SmartBear has recently released a commercial tool called API Inspector[10] which generates OpenAPI definitions from historical calls. While their approach is similar to ours, it only focuses on the behavioral level (e.g., paths, operations, parameters) and neglects the data structures, thus making most of the generated definitions incomplete. Our approach may join this tool ecosystem by inferring the OpenAPI definitions not only on the behavioral level, but also on the structural one.

Research-wise, there is a limited number of related efforts and barely any targeting specifically REST or Web APIs in general. Some research efforts (i.e., [Mot+11; Ser+08]) focus on the analysis of service interaction logs to discoverer message correlation in business processes. Other works (i.e., [Rod+15; SP04; QBC13]) are more proactive and try to suggest possible compositions based on a WSDL (or similar) description of the service. Nevertheless, they all focus on the interaction patterns and do not generate any description of Web APIs specification (or the initial WSDL document for previous approaches) themselves. SpyREST [SAM15] is a closer work to ours. It proposes a proxy server to analyze HTTP traffic involved in API calls to generate API documentation. Still, the generated documentation is intended to be read by humans and therefore does not adhere to any formal API specification language.

Other research efforts limit themselves to discover the data model underlying an API, specially by analyzing the JSON documents it returns. For instance, the works in [Kle+15] and [RMM15] analyze JSON documents in order to generate their (implicit) schemas. However, they are specially bounded to NoSQL databases and are not applicable for Web APIs. On the other hand, JSONDiscoverer [CC16] generates UML class diagrams from the JSON data returned after calling a Web API. We use this tool in our

[6] https://swagger.io/tools/swagger-ui/
[7] http://rebilly.github.io/ReDoc/
[8] https://editor.swagger.io/
[9] https://studio.restlet.com
[10] https://swagger.io/tools/swagger-inspector/

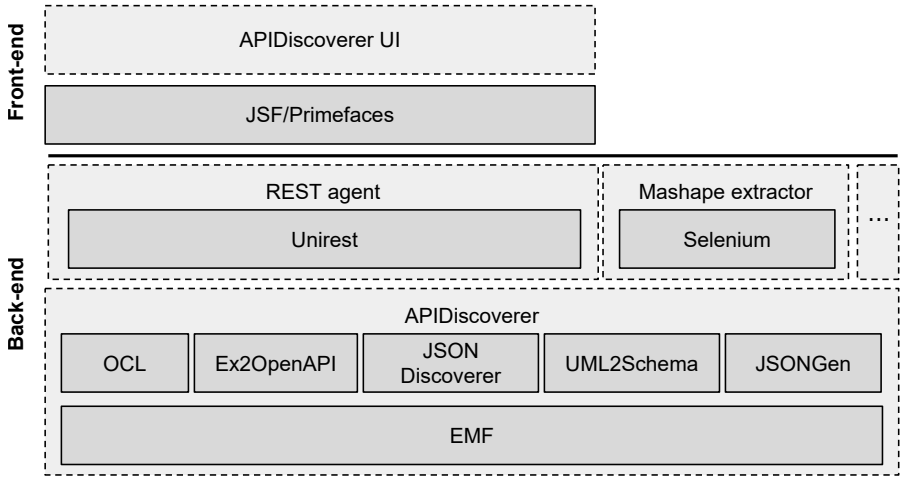


Figure 5.4: APIDISCOVERER architecture.

structural discoverer phase.

5.7 TOOL SUPPORT

Figure 5.4 shows the underlying architecture of our discovery tool. Our tool includes a front-end, which allows users to collect and run API call examples to trigger the launch of the core APIDISCOVERER process (see *APIDiscoverer UI*); and a back-end, which includes all the components to parse the calls and responses, generate the intermediate models, etc. Our tool has been implemented in Java and is available as an Open Source application[11].

More specifically, APIDISCOVERER is a Java Web application that can be deployed in any Servlet container (e.g., APACHE TOMCAT[12]). The application relies on JavaServer Faces (JSF), a server-side technology for developing Web applications; and Primefaces[13], a UI framework for JSF applications. Figure 5.5 shows a screenshot of the APIDISCOVERER interface. The central panel of APIDISCOVERER contains a form to provide API call examples either by sending requests or using our JSON-based representation format. The former requires providing the request and obtaining a response from the API. As result, a JSON-based API call example is shown on the

[11] https://github.com/SOM-Research/APIDiscoverer
[12] http://tomcat.apache.org/
[13] http://www.primefaces.org

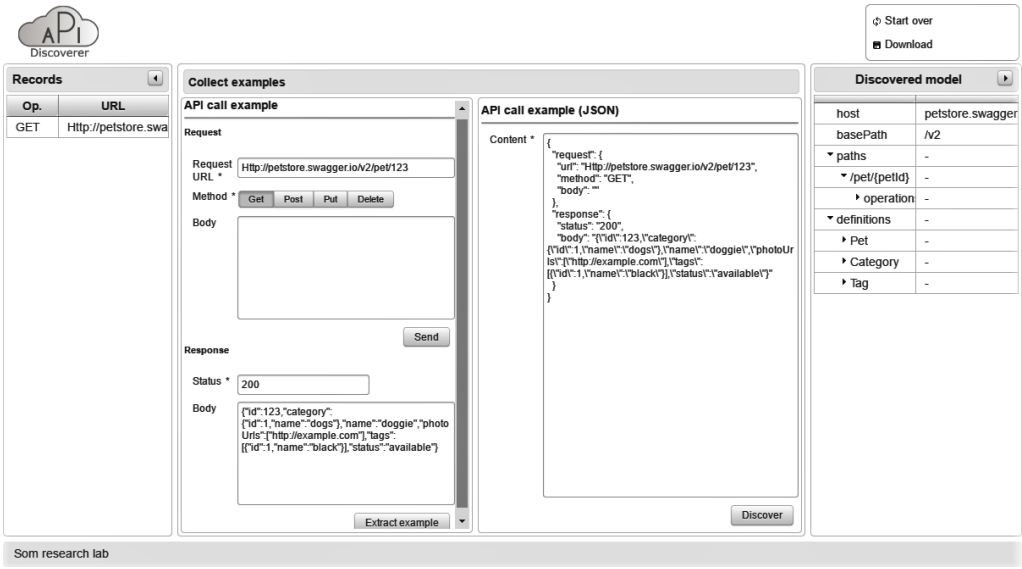


Figure 5.5: Screenshot of the UI of APIDISCOVERER.

right. The latter only requires providing the JSON-based API call example. API call examples are then used by APIDISCOVERER to obtain/enrich the corresponding OpenAPI model. The example history is shown on the left panel and an intermediate OpenAPI model is shown on the right panel. The OpenAPI model is updated after each example with the new information discovered by the last request. Finally, a button in the top panel allows the user to download the final OpenAPI description file.

The main components of the back-end are (1) a REST agent and (2) the core APIDISCOVERER. The REST agent relies on UNIREST[14], a REST library to send requests to APIs to build and collect API call examples. APIDISCOVERER relies on a plethora of web/modeling technologies, namely, (i) EMF[15] as a modeling framework to implement the OpenAPI metamodel, (ii) the Eclipse OCL to validate models and (iii) the JSONDISCOVERER to discover models from JSON examples. Additionally, we have implemented the required components (i) to discover OpenAPI elements from API call examples (see *Ex2OpenAPI* in Figure 5.4), (ii) to transform UML models to a list of schema elements using model-to-model transformations (see *UML2Schema* in Figure 5.4), and (iii) to generate an OpenAPI description file from an OpenAPI model by using model-to-text transformations (see *JSONGen* in Figure 5.4).

[14] `http://unirest.io`

[15] `http://www.eclipse.org/modeling/emf/`

Beyond these key components, we have also developed *MashapeDiscoverer*, a proof-of-concept to show how the API call examples can be derived from other sources like available examples in the API documentation (in this specific case, from APIs in the MASHAPE marketplace[16], a documentation portal with over 2,000 APIs) by using SELENIUM[17] to crawl the documentation pages and extract the relevant examples information (i.e, entrypoints, parameters, response examples).

Additionally, we have created HAPI[18], a public REST Web API directory and an open source community-driven project, which stores the discovered Web APIs. Besides allowing users to download the Web API specifications, this directory invites developers to contribute using the well-known pull-request model of GitHub. In order to enrich HAPI, we have also created two OpenAPI importers for APIS.GURU and APIS.IO that use their dedicated Web APIs[19]. This allows easily adding APIs with an already predefined specification to HAPI.

5.8 SUMMARY

This chapter presented a model-driven approach to discover OpenAPI definitions for REST APIs. From a set of API call examples, our approach relies on two discoverer (i.e., the behavioral discoverer, the structural discoverer) to generate an OpenAPI model describing the operations and data types of the target API. At the end of the process, our approach generates a valid JSON OpenAPI definition from the intermediate OpenAPI model. The generated OpenAPI definitions are stored in a shared directory where anybody can access and contribute.

[16] `https://market.mashape.com`
[17] `http://docs.seleniumhq.org/projects/webdriver/`
[18] `https://github.com/SOM-Research/hapi`
[19] `https://apis.guru/api-doc/` and `http://www.apis.io/apiDoc`

CHAPTER

6 Testing REST APIs

In the previous chapter we have presented an approach to generate OpenAPI definitions from API call examples, thus benefiting from the different advantages of the OpenAPI specification presented previously. One arising problem is how to ensure that API specifications reflect correctly the behavior of their corresponding APIs. This problem is not only related to generated specifications but also to official ones which should reflect their API implementations. This chapter tries to solve this problem by presenting a model-driven approach to generate test cases for REST APIs relying on OpenAPI specification, with the goal to ensure that API implementations conform to their specifications. Unlike existing approaches which mainly focus on nominal test cases (i.e., using correct input) while neglecting fault-based ones (i.e., using incorrect input) and require parameter inputs, our approach covers both test cases scenarios and provides parameter inputs.

This chapter is organized as follows. Sections 6.1 presents the background. Section 6.2 describes our approach. Sections 6.3, 6.4, 6.5, and 6.6 present the different steps of the test case generation process for the OpenAPI specification, namely: extracting OpenAPI models, inferring parameter values, generating test case definitions, and generating executable code, respectively. Section 6.7 describes the tool support. Section 6.8 presents the validation process. Section 6.9 presents related work and, finally, Section 6.10 presents a summary of the chapter.

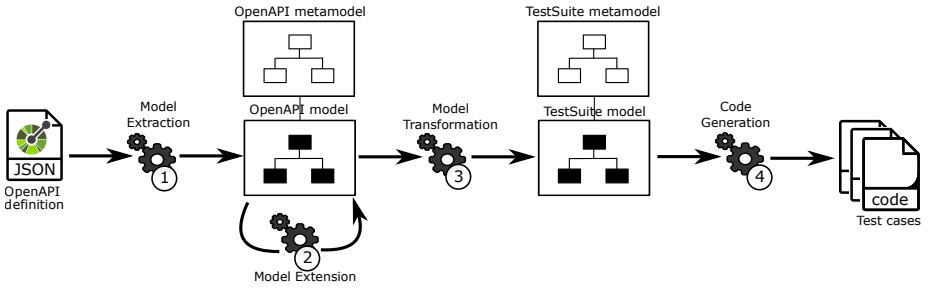


Figure 6.1: Test cases generation for the OpenAPI specification.

6.1 Background

API testing is a type of software testing that aims to validate the expectations of an API in terms of functionality, reliability, performance, and security. *Specification-based API testing* is the verification of the API using the available functionalities defined in a specification document [BHH13]. In specification-based REST API testing, test cases consist of sending requests over HTTP/S and validating that the server responses conform to the specification, such as the OpenAPI one. For instance, a test case could be created for the operation *findPetsByStatus* of the Petstore example (see Figure 3.6 in Chapter 3) by sending a GET request to `http://petstore.swagger.io/v2/pet/findByStatus?status=available` and expecting a response having the HTTP status code `200` and a JSON array conforming the schema defined in the specification. Automated specification-based API testing involves generating such test cases.

Fault-based testing is a special type of testing which aims to demonstrate the absence of *pre-specified* faults [Mor90]. Instead of detecting unknown faults, the aim of fault-based test cases is to prove that known faults do not exist, thus promoting reliability. For instance, the parameter *status* of the operation *findpetsByStatus* is of type enumeration and restricted to the values: `available`, `pending`, and `sold`. Therefore, a test case could be created for the operation *findPetsByStatus* using an incorrect value for the parameter *status* by sending a GET request to `http://petstore.swagger.io/v2/pet/findByStatus?status=test` and expecting a response having the status code `400 BAD REQUEST`.

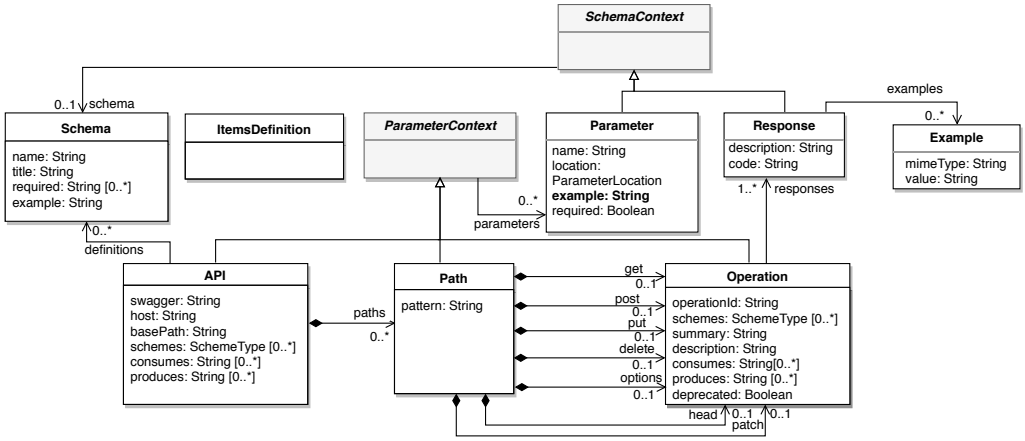


Figure 6.2: An excerpt of the extended OpenAPI metamodel.

6.2 Our Approach

We define an approach to automate specification-based REST API testing, which we illustrate using the OpenAPI specification, as shown in Figure 6.1. Our approach relies on model-based techniques to promote the reuse and facilitate the automation of the generation process. OpenAPI definitions are represented as models conforming to the OpenAPI metamodel (see Section 3.1), while test case definitions are stored as models conforming to the TestSuite metamodel. This metamodel allows creating test case definitions for REST API specifications and is inspired by the best practices in testing REST APIs.

Our approach has four steps. The first step extracts an OpenAPI model from the definition document of a REST API, by parsing and processing the JSON (or YAML) file. The second step enriches the created OpenAPI model with parameter examples, which will be used as input data in the test cases. The third step generates a TestSuite model from the OpenAPI model by inferring test case definitions for the API operations. Finally the last step transforms the TestSuite model into executable code (JUnit in our case). In the following sections, we explain each step in detail.

6.3 Extracting OpenAPI Models

The OpenAPI metamodel and extraction of OpenAPI models from OpenAPI definitions have been already addressed in Chapter 3. For the context of this approach, we extended the OpenAPI metamodel in order to support

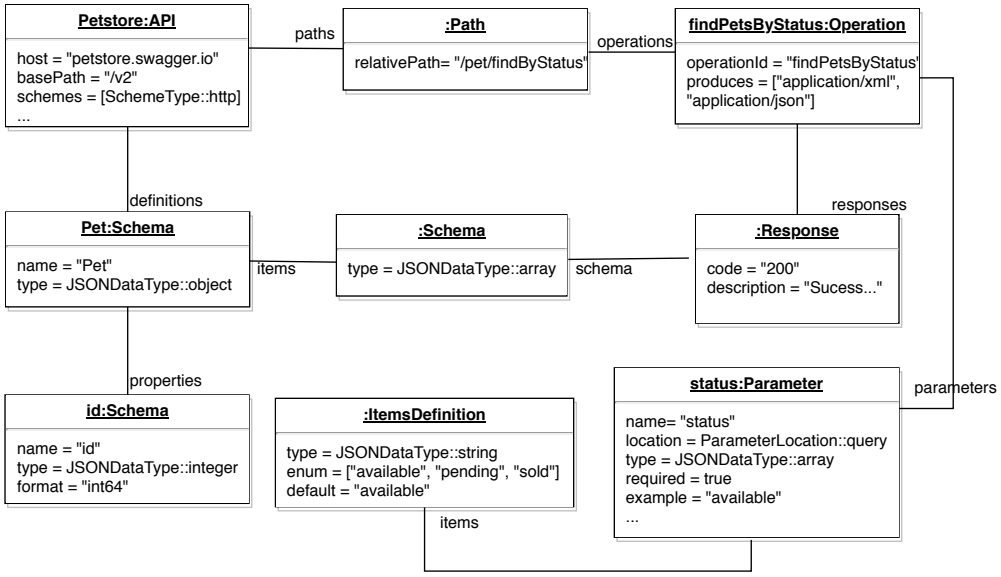


Figure 6.3: Excerpt of the OpenAPI model corresponding to the Petstore API including a parameter example.

the definition of parameter value examples. Such feature is not provided out-of-the-box in OpenAPI 2[1]. Therefore, we added the attribute `example` to the element `Parameter` in the OpenAPI metamodel to allow the definition of examples as depicted in Figure 6.2.

Figure 6.3 shows an excerpt of the generated OpenAPI model of the Petstore API including the *findPetsByStatus* operation. Note that this model also includes an inferred value for the parameter status (i.e., `example = available`). We present the inference rules in the next section.

6.4 Inferring Parameter Values

The goal of this step is to enrich OpenAPI models with the parameter values needed to generate test cases for an operation. Definition 1 describes a testable operation and PR 1, PR 2, and PR 3 are the applied rules to infer parameter values ordered by preference.

Definition 1 (Testable Operation)**.** An API operation can be tested if all the values of its required parameters can be inferred.

[1]Parameter examples in the OpenAPI specification 2 should be added as vendor extensions (i.e., using the prefix `x-`). This was clearly a limitation in this version of the specification and has be fixed in version 3

PR 1 (Simple parameter value inference)**.** A value of a parameter `p` could be inferred from: (1) examples (i.e., `p.example` and `p.schema.example`), (2) default values (i.e., `p.default` or `p.items.default` if `p` of type array) or (3) enums (i.e., the first value of `p.enum` or `p.items.enum` if `p` is of type array).

PR 2 (Dummy parameter value inference)**.** A value of a parameter `p` could be set to a dummy value (respecting the type) if a request to the operation of `p` including that value returns a successful response (i.e., a `2xx` response code class).

PR 3 (Complex parameter value inference)**.** A value of a parameter `p` could be inferred from the response of an operation `o` if: (1) `o` is testable; (2) `o` returns a successful response `r`; and (3) `r.schema` contains a property matching `p`[2].

The previous rules are applied in sequence to infer a parameter value. For instance, PR 1 was applied to the parameter *status* of the operation *findPetsByStatus* of the Petstore API since `items.default = available` (see Figure 3.6 in Chapter 3). PR 3 was applied to the parameter *petId* of the operation *getPetById* since a *petId* example can be inferred from the operation *findPetsByStatus*. Note that PR 2 and PR 3 stress the API (i.e., they involve sending several requests to the API) and may lead to biased results as the very own API under test is used to infer the values. Thus, the three rules are applied to infer the required parameters while only the first rule is used for the optional ones.

6.5 Extracting Test Case Definitions

In the following we explain the generation process of test case definitions from OpenAPI models. We start by introducing the TestSuite metamodel, used to represent test case definitions; and then we present the transformation rules to create test cases.

6.5.1 *The TestSuite Metamodel*

The TestSuite metamodel allows creating test case definitions for REST APIs. Figure 6.4 shows an excerpt of this metamodel. The `TestSuite` element represents a test suite and is the root element of the metamodel. This

[2]Some basic heuristics are applied in the parameter matching process (e.g., *petId* parameter is inferred from the *id* property of the schema *Pet*.)

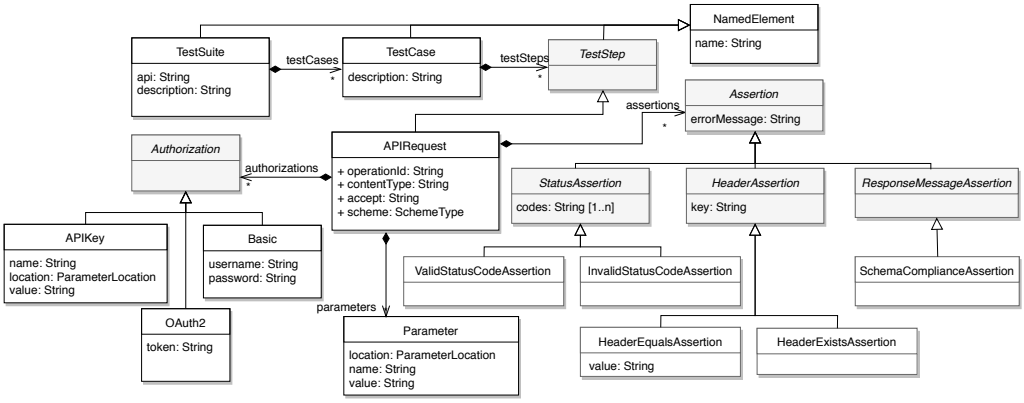


Figure 6.4: Excerpt of the TestSuite metamodel.

element includes a name, the URL of the REST API definition (i.e., `api` attribute), and a set of test case definitions (i.e., `testCases` reference).

The `TestCase` element represents a test case definition and includes a name, a description, and a set of test steps (i.e., `testSteps` references). The `APIRequest` element is a specialization of the `TestStep` element which represents the details of an API request to be sent and the logic to validate the returned response. It includes the target operation (i.e., `operationId` attribute), the content type (e.g., `application/json`), the accepted MIME type (i.e., `accept` attribute, e.g., `application/json`), and the transfer protocol of the request (i.e., `scheme` attribute, e.g., `http`). The values for the content type and accepted MIME type should adhere to RFC6838[3]. The `APIRequest` element also includes the parameters of the request (i.e., `parameters` reference), the request authorization method (i.e., `authorizations` reference), and the assertions to be validated for this API request (i.e., `assertions` reference).

A parameter of an API request is represented by the `Parameter` element and includes its location (i.e., `location` attribute), name, and value. `Parameter` elements are mapped to parameters of the REST API definition by their name and location.

The `Authorization` element represents an abstract authorization method in a REST API. It has three specializations, namely: `APIKey` for API key authorizations, `Oauth2` for OAuth 2.0 authorization using a client token, and `Basic` for basic authentication using a username and a password.

The `Assertion` element represents the root element of the hierarchy of assertions supported by the TestSuite metamodel. As can be seen assertions

[3] `https://tools.ietf.org/html/rfc6838`

are organized in three categories which define validations regarding: (1) the HTTP status code of the received HTTP response (i.e. the abstract element `StatusAssertion`); (2) the header of the received HTTP response (i.e., the abstract element `HeaderAssertion`); and (3) the message of the received HTTP response (i.e., the abstract element `ResponseMessageAssertion`).

The `ValidStatusCodeAssertion` and `InvalidStatusCodeAssertion` elements are specializations of the `StatusAssertion` element and allow checking that the HTTP status of the received response is within the defined list of codes, or not, respectively. The `HeaderEqualsAssertion` and `HeaderExistsElement` elements are specializations of the element `HeaderAssertion` and allow checking that the value of an HTTP header in the response is equals to the expected value, and whether an HTTP header exists in the response, respectively.

The `SchemaComplianceAssertion` element is a specialization of the `ResponseMessageAssertion` element which allows checking whether the the response message is compliant with the schema defined by the definition (e.g., check that the returned *Pet* instance is compliant with the `Pet` definition).

6.5.2 *OpenAPI to TestSuite Transformation*

We present now the generation rules we have defined to test that REST APIs conform to their OpenAPI definitions. We define two rules (i.e., GR 1 and GR 2) to generate test case definitions in order to assess that the REST APIs behave correctly using both correct and incorrect data inputs. GR 1 generates nominal test case definitions which assess that given correct input data, the API operations return a successful response code (i.e., `2xx` family of codes) and respect their specification. GR 2 generates faulty test case definitions which assess that given incorrect input data, the API operations return a client error response code (i.e., `4xx` family of codes).

GR 1 (Nominal test case definition)**.** If an operation `o` is testable then one `TestCase` testing such operation is generated such as `APIRequest` includes the inferred required parameter values (if any). Additionally, if `o` contains inferable optional parameters then another `TestCase` testing such operation is generated such `APIRequest` as includes the inferred required and optional parameter values. In both cases `APIRequest` includes the following assertions:

- ◇ `ValidStatusCodeAssertion` having a successful status code (i.e., `2xx` family of codes)

Table 6.1: Wrong data types generation rules.

PARAMETER TYPE	GENERATION RULE
`object`	an object violating the object schema
`integer/int32`	a random string or a number higher than $2^{31}-1$
`integer/int64`	a random string or a number higher than $2^{63}-1$
`number/float` `number/double` `string/byte` `string/datetime` `string/date`	a random string
`boolean`	a random string different from `true` and `false`

Table 6.2: Violated constraints generation rules.

CONSTRAINT	GENERATION RULE
`enum`	a string or a number outside of the scope of the enumeration
`pattern`	a string violating the regEx
`maximum/exclusiveMaximum`	a number higher than maximum
`minimum/exclusiveMinimum`	a number lower than minimum
`minLength`	a string with the length lower than `minLength`
`maxLength`	a string with the length higher than `maxLength`
`maxItems`	an array having more items than `maxItems`
`minItems`	an array having less items than `minItems`
`uniqueItems`	an array with duplicated values
`mutlipleOf`	a number not divided by `multipleOf`

- ◇ `SchemaComplianceAssertion`, if `o.responses` contains a response `r` such as `r.code` is a successful status code (i.e., `2xx` family of codes) and `r.schema = s`
- ◇ `HeaderExistsAssertion` having `key = h`, if `o.responses` contains a response `r` such as `r.code` is a successful code (i.e., `2xx` family of codes) and `r.headers` contains `h`

GR 2 (Faulty test case definition)**.** For each parameter `p` in an operation `o`, a `TestCase` testing such operation is generated for the following cases:

- ◇ **Required missing:** If `p` is required and not located in the path then `APIRequest` will not include a value of `p`.
- ◇ **Wrong data types:** If `o` is testable and `p` is not of type string then `APIRequest` will include the inferred required parameter values (if any) and a wrong value of `p` as described in Table 6.1.

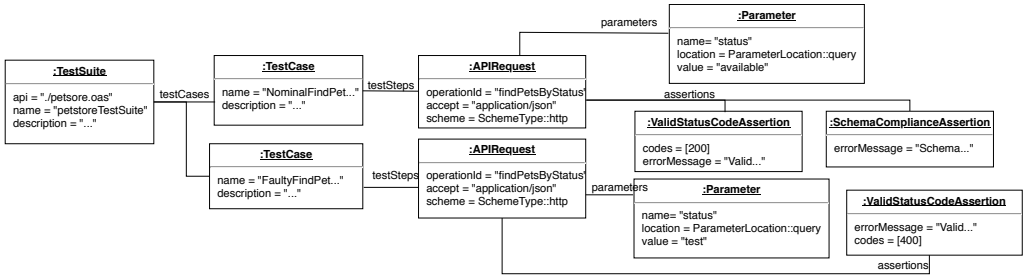


Figure 6.5: TestSuite model representing nominal and faulty test cases for the Petstore API.

- **Violated constraints:** If `o` is testable and `p` includes a constraint then `APIRequest` will include the inferred required parameter values (if any) and a value of `p` violating such constraint as described in Table 6.2.

In all cases `APIRequest` includes the following assertion:

- `ValidStatusCodeAssertion` having a client-error code (i.e., `4xx` family of codes).

Figure 6.5 shows an example of nominal and faulty test cases for the operation *findPetsByStatus* of Petstore represented as a model conforming to our TestSuite metamodel. Note that our approach does not consider dependencies between operations since such information is not provided in OpenAPI 2. With no knowledge about dependencies it is not possible to *undo* the changes and therefore each operation is treated in an isolated mode (i.e., test case side-effects are ignored, e.g, creating resource, deleting resources). Heuristics could be defined to detect such dependencies, however, they would assume that API providers follow the best practices in the design of their REST APIs and respect the semantics of HTTP protocol, which in practice is generally not the case [Rod+16]. This situation does not affect *read-only* APIs (e.g., Open Data APIs) but could affect *updateable* APIs. Thus, we created two generation modes: *safe* (i.e., only GET operations are considered to generate test cases), and *unsafe* (i.e., all operations are considered to generate test cases).

6.6 Code Generation

The final step of the process consists on generating test cases for a target platform. Since the test case definitions are platform-independent, any

programing language or testing tool could be considered. In Section 6.7 we will illustrate our approach for Java programming language and JUnit templates.

6.7 TOOL SUPPORT

We created a proof-of-concept plugin implementing our approach. The plugin extends the Eclipse platform to generate: (1) OpenAPI models from OpenAPI definition files; (2) TestSuite models from the generated OpenAPI models; and (3) Maven projects including the JUNIT templates implementations for the test case definitions in the TestSuite models. The plugin has been made available in our GitHub repository[4].

The OpenAPI and TestSuite metamodels are implemented using EMF. We created a set of Java classes to infer the parameter values as described in Section 6.4. We defined an ATL transformation to derive a TestSuite model from an input OpenAPI model following the generation rules described in Section 6.5. Finally, we used ACCELEO[5] to generate the JUNIT test cases.

Figure 6.6 shows a screenshot of the generated Maven project for the Petstore API including the corresponding tests for test cases showed in Figure 6.5. The generated classes rely on JUNIT[6] to validate the tests and UNIREST framework[7] to call the REST API. To test the schema compliance we infer the JSON schema from the API description and use the framework JSON SCHEMA VALIDATOR[8] to validate the *entity-body* of the response against the inferred schema. The actual implementation of the tool supports the authentication methods Basic and API Key.

6.8 VALIDATION

In order to validate our approach and the companion tool implementation, we address the following research questions:

RQ1 What is the coverage level of the generated test cases? Test cases cover a set of endpoints, operations, parameters and data definitions of the OpenAPI definition. Our goal is to provide high coverage degrees.

[4] https://github.com/SOM-Research/test-generator
[5] https://www.eclipse.org/acceleo/
[6] https://junit.org/
[7] http://unirest.io/
[8] https://github.com/java-json-tools/json-schema-validator

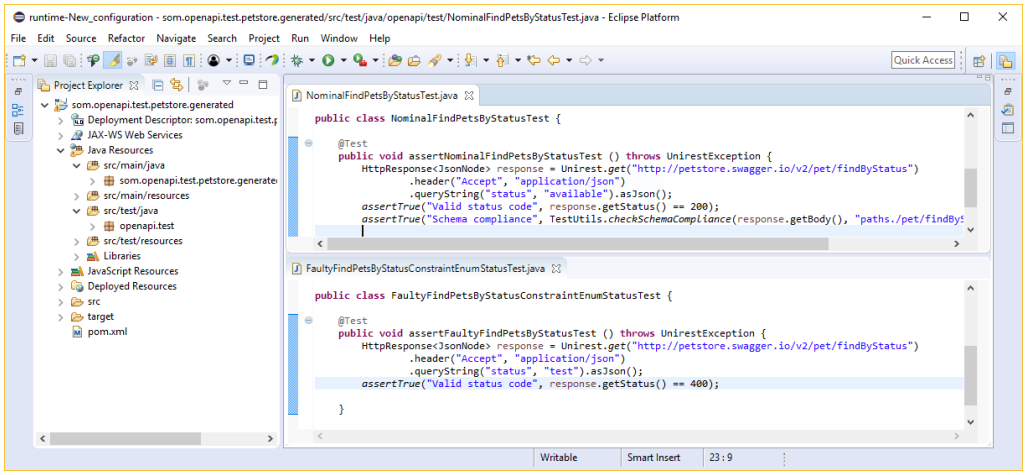


Figure 6.6: TESTGENERATOR: a screenshot of the generated Maven project of the Petstore API showing a nominal and a faulty test case.

RQ2 What are the main failing points in the definitions and implementation of real world REST APIs? We use our approach to study how current REST APIs perform and when they usually fail.

To answer these questions we ran our tool on a collection of OpenAPI definitions. We describe next how we created this collection and addressed the research questions.

6.8.1 *REST APIs Collection and Selection*

We created a collection of REST APIs by querying APIS.GURU, a site including 837 REST APIs described by OpenAPI definitions. To build our collection, we performed a two-phased filtering process to (1) select free, open and available REST APIs which are not services (i.e., they provide access to data models) and (2) remove those REST APIs with incorrect or invalid OpenAPI definitions. Next we detail these phases.

The first phase performs the selection process relying on the metadata of each API provided in the OpenAPI definition or in its external documentation. The selection criteria in this phase removes: (1) APIs providing access to functionalities and services (mostly applying fees) to manage specific environments such as IoT, Cloud, and messaging which are normally tested in controlled environments (654 APIs); (2) deprecated and/or unreachable APIs (15 APIs); and (3) APIs with restricted access (e.g., available for part-

Table 6.3: Coverage of the test cases in terms of operations, parameters, endpoints and definitions.

Elements	Count	Coverage			Coverage (%)		
		All	Nominal	Faulty	All	Nominal	Faulty
Operations	367	320	303	233	87%	82%	63%
Parameters	949	595	485	476	62%	51%	50%
Endpoints	356	289			81%		
Definitions	313	239			76%		

Table 6.4: Errors found in the test cases.

	Total	Nominal test cases		Faulty test cases	
		4xx/500	Schema	500	2xx
Number of APIs	37	9	11	11	20
%	40%	25%	30%	30%	55%

ners only, rate limited, etc.) (21 APIs). At the end of this phase our collection included 147 APIs.

The second phase analyzes each OpenAPI definition of the previously selected REST APIs and removes (1) APIs which showed semantic errors (i.e., validation failed against the OpenAPI specification) (21 APIs); (2) APIs relying on OAuth as authentication mechanism (15 APIs); (3) big APIs including more than 100 parameters to avoid biased results (14 APIs); and (4) not REST-friendly APIs (e.g., using formData, appending body to GET methods) (6 APIs). At the end of this phase, our collection included 91 OpenAPI definitions[9] belonging to 32 API providers. Most of the APIs in the collection provide support for *text processing* (25 APIs) and *Open Data* (21 APIs). Other APIs target other domains such as *transport* (6 APIs), *media* (5 APIs), and *entertainment* (4 APIs). The REST APIs described by these definitions have on average 4 operations and 10 parameters. The collection included 71 REST APIs that require authentication, thus API keys were requested to use them. Furthermore, the OpenAPI definitions of 51 of the selected APIs were polished to follow the security recommendations (i.e., using the *SecurityDefinition* element provided by the specification instead of parameters).

To minimize the chances of subjectivity when applying this process, we applied a code scheme as follows. The process was performed by one coder, who is the author of this thesis. Then a second coder, who is the second adviser of this thesis, randomly selected a sample of size 30% (251 Web

[9]The full list of collected APIs, including the application of the selection process, and the results for answering the research questions are available at `http://hdl.handle.net/20.500.12004/1/C/EDOC/2018/001`.

APIs) and performed the process himself. The intercoder agreement reached was higher than 92%. All disagreement cases were discussed between the coders to reach consensus.

6.8.2 *Results*

Once we built our collection, we ran our tool for each REST API to generate and execute the test cases.

RQ1. Coverage of the generated test cases. For the 91 APIs, we generated 958 test cases (445 nominal / 513 faulty). We analyzed these test cases to determine their coverage in terms of operations, parameters, endpoints and definitions. We defined the coverage as follows: an operation is covered when at least one test case uses such operation; a parameter is covered when at least one test case includes an operation targeting such parameter; an endpoint is covered when all its operations are covered; and a definition is covered when it is used in at least one test case. We report the coverage for both the total test cases and the nominal/faulty ones.

Table 6.3 summarizes our results. As can be seen, we obtain high coverage levels except for parameters. The value for nominal test cases for parameters is low since this kind of test cases relies on required parameters (i.e., testable operations) and only 30% of the parameters were required. On the other hand, the nature of the parameters and poor OpenAPI definitions affected the coverage of faulty test cases. Most of the parameters were of type *String* and do not declare constraints to validate their values, thus hampering the generation of faulty test cases. Richer OpenAPI definitions would have helped us to tune our inference technique and therefore increase the coverage for operations and parameters. Further analysis shown that many APIs provide constraints regarding their parameters in natural language instead of the method provided by the OpenAPI specification. For instance, the *data at work API*[10] includes a parameter named *limit* and described as “Maximum number of items per page. Default is 20 and cannot exceed 500”. Instead, using the tags *default* and *maximum* defined by the OpenAPI specification would have helped us generate more test cases for this parameter.

RQ2. Failing points in the definitions and implementation in real world REST APIs. To answer this question we analyzed the errors found when executing the generated test cases. Table 6.4 shows the results. As can be seen, 37 of the 91 selected REST APIs (40% of the APIs) raised

[10] `https://api.apis.guru/v2/specs/dataatwork.org/1.0/swagger.json`

some kind of errors, which we classified into two categories (nominal/faulty). The nominal test case errors included: those regarding the status code (i.e., obtaining an error status code when expecting a successful response, see column 3) and those regarding schema compliance (i.e., the object returned in the body of the response is not compliant with the JSON schema defined in the specification, see column 4). The faulty test case errors included two kind of errors regarding the status code where a client error response (i.e., 4xx code) was expected but either a server error response code (i.e., 500 code, see column 5) or a successful response code (i.e., 2xx code, see column 6) was obtained. The errors regarding nominal test cases are mainly related to mistakes in the definitions, while the faulty ones are related to bad implementations of the APIs.

The errors in the definitions vary from simple mistakes such a missing *required* field for a parameter (e.g., the parameter *q* in the operation *GET action/organization_autocomplete* of the *BC Data Catalogue API*[11]) to complex ones such as having a wrong JSON schema (e.g., *Data at Work API*).

The errors in the implementation of the APIs are characterized by sending 500 (i.e., internal server error) or 2xx (i.e., successful) status codes on an error condition instead of a 4xx status code. On the one hand, the 500 internal server error tells the client that there is a problem on the server side. This is probably a result of an unhandled exception while dealing with the request. Instead, the server should include a validation step which checks the client's input and sends a client side error code (e.g., 400 wrong input) on violated constraints with a text explaining the reasons. On the other hand, sending 200 status code on an error condition is a bad practice [RAR13]. Furthermore, four errors were linked to the limitation of OpenAPI to define mutually exclusive required parameters (e.g., The *Books API*[12]), resulting in inconsistent definitions. Such limitation is confirmed by Oostvogels et al. [ODD17].

Even though we do not consider links between operations and side effects, we believe our approach may help developers to identify the main error-prone points in REST API development. Thus, our experiments showed that extra attention should be paid when dealing with data structures and returning the proper HTTP error code to allow the client to recover and try again.

6.8.3 *Threats to Validity*

Our work is subjected to a number of threats to validity, namely: (1)

[11] `https://api.apis.guru/v2/specs/gov.bc.ca/bcdc/3.0.1/swagger.json`

[12] `https://api.apis.guru/v2/specs/nytimes.com/books_api/3.0.0/swagger.json`

internal validity, which is related to the inferences we made; and (2) external validity, which discusses the generalization of our findings. Regarding the internal validity, many definitions in APIS.GURU have been created or generated by third-parties which may have consequences on the quality of such definitions. Therefore, different results can be expected if all definitions are provided by the API owners. Furthermore, some providers are over-represented in APIS.GURU which may lead to biased results. As for the external validity, note that the nature and size of the sample may not be representative enough to generalize the findings of the experiments.

6.9 Related Work

Many approaches proposed both nominal (e.g., [Bai+05; Bar+09; HM08; OX04]) and faulty (e.g., [XOL05; ZZ05]) specification-based test cases generation for the classical SOAP Web APIs by relying on their WSDL definitions. Research works targeting test cases generation for REST APIs, on the other hand, are relatively limited. For instance, [FB15], [CK09], and [Ben+14] propose different approaches for automated test case generation, but relying on manual definition of a model, a test specification DSL, and a JSON schema, respectively, thus making their approaches bound to their ad-hoc specifications. [Arc17] proposes an approach to generate white-box integration test cases for REST APIs using search algorithms and relying on the OpenAPI specification. However, in such scenario, the developer must have full access to the code of the API to generate the test cases. On the other hand, our approach does not require providing any model or definition other than OpenAPI.

Beyond the research context, several commercial and open source tools provide testing facilities for REST APIs. Some of these tools propose automated testing for REST API specifications, such as the OpenAPI specification. These tools can be divided into three categories, namely: (1) integrated environments, (2) command line tools, and (3) development frameworks. The integrated environment testing tools provide a GUI which can be used to create and run test cases (e.g., SOAPUI/READYAPI![13], RUNSCOPE, or APIFORTRESS[14]). Command line tools propose a command line interface for configuring and running the test cases (e.g., DREDD or GOT-SWAG[15]). Finally, development frameworks provide testing facilities for different

[13] https://www.soapui.org/
[14] http://apifortress.com/
[15] https://github.com/mobilcom-debitel/got-swag

programming languages (e.g., HIPPIE-SWAGGER[16], SWAGGER REQUEST VALIDATOR[17], or SWAGGER TESTER[18]). Certainly, these tools prove the importance of specification-based testing for REST APIs, but they all require a considerable effort to configure them and to provide input data. Furthermore, these tools do not provide fault-based testing out the box.

6.10 SUMMARY

In this chapter we have presented a model-driven approach to automate specification-based REST API testing by relying on OpenAPI. We used the OpenAPI metamodel presented in Chapter 3 and created the TestSuite metamodel to define test cases for REST APIs. Models conforming to these metamodels are used to generate the test cases. The TestSuite metamodel is reusable, thus allowing defining test suites for any REST API. As a proof-of-concept, we created a plugin implementing our approach for the Eclipse platform. Our experiments show that the generated test cases cover on average 76.5% of the elements included in the definitions and that 40% of the tested APIs contain bugs either in their specification or server implementation.

[16] https://github.com/CacheControl/hippie-swagger
[17] https://bitbucket.org/atlassian/swagger-request-validator
[18] https://swagger-tester.readthedocs.io/en/latest/

CHAPTER

7 Generating REST APIs

In Chapter 4 we covered the *APIfication* of models by enabling model management using REST APIs. Models also play a fundamental role in the design and conception of Web APIs. In this Chapter we present a model-driven approach to generate REST APIs from conceptual data models. We rely on OData protocol for the realization of the API due to its adaptability to data-centric REST APIs and its querying capability. In fact, the OData protocol enables the creation of data-centric Web services, where URL-accessible resources are defined according to an entity model and can be queried by web clients using standard HTTP messages. However, designing OData services is tedious and time-consuming, specially for relational databases where extra effort is needed to align the service with the database query capabilities. While Chapter 3 covered the specification of OData services (using a metamodel and a UML profile), in this chapter we present a model-driven approach to (semi)automate the generation of ready-to-deploy REST APIs targeting relational databases.

This chapter is structured as follows. Section 7.1 describes our approach and Section 7.2 shows the running example used to illustrate our approach. Section 7.3 describes how we drive OData models from UML models. Sections 7.4 and 7.5 show the database schema generation process and the OData service generation process, respectively. Section 7.6 describes the tool support. Section 7.7 presents the related work. Finally, Section 7.8 summaries this chapter.

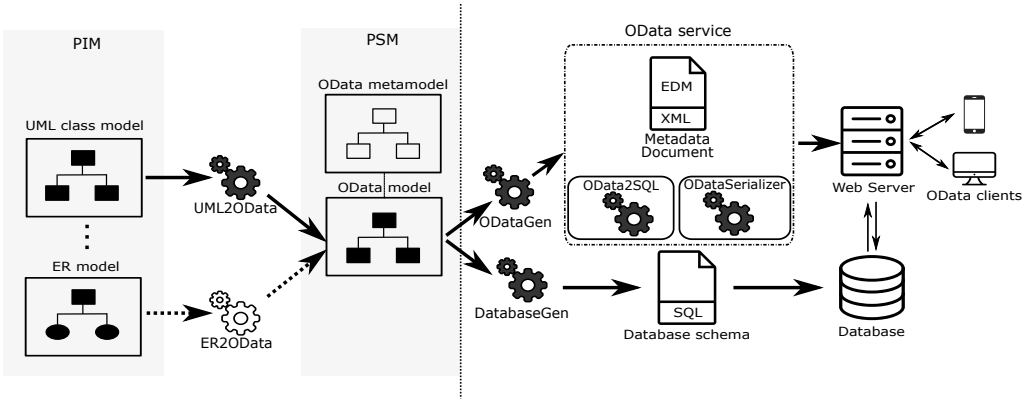


Figure 7.1: The OData service specification and generation approach.

7.1 Our Approach

We propose a model-driven approach where OData models drive the generation of OData services using a relational database as storage solution. These OData models could be directly specified but typically they will be derived from an input UML or ER model describing the domain. Figure 7.1 shows an overview of our approach.

On the left-hand side of Figure 7.1, and following the MDA terminology of OMG (see Chapter 2, Section 2.3), we have the UML and ER models at the PIM level while the OData metamodel would belong to the PSM level as a refinement of the previous one. The mapping between the PIM and PSM level is rather straightforward, as we will show. We will focus on UML models (see *UML2OData* transformation) and ER models (see *ER2OData* transformation) but a similar approach can be followed for another kind of PIM model.

On the right-hand side of Figure 7.1, we see how OData models are used to generate: (1) an OData service wrapped in a Web application to be deployed in a server (see *ODataGen* transformation); and (2) the corresponding database schema to initialize the database (see *DatabaseGen* transformation). The OData service includes: (a) the OData metadata document, which defines the Entity Data Model (EDM) for the data exposed by the service [PHZ14e]; (b) the logic to transform OData requests into SQL statements according to the query language defined by OData protocol [PHZ14d] (see *OData2SQL* component); and (c) an OData serializer, which defines the serialization mechanism according to OData JSON format [HPB14] and

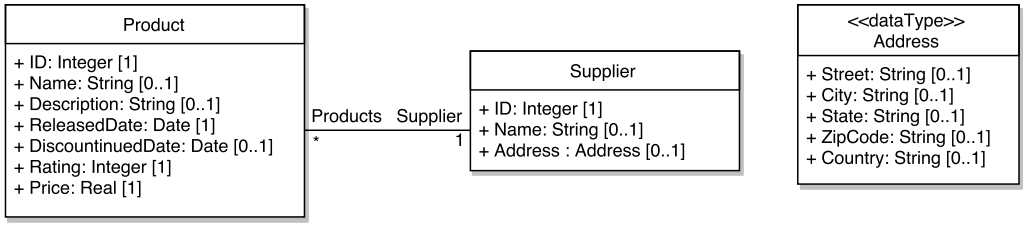


Figure 7.2: UML model of the running example.

OData Atom format [ZPH14] (see *ODataSerializer* component).

OData defines three levels of conformance for an OData Service, namely: minimal, intermediate and advanced (cf. OData protocol [PHZ14c], Section 13). Each level defines a set of requirements and recommendations that a service should fulfill in order to conform to this level. The OData service generated with our approach fully conforms to the OData Intermediate Level and partially to the OData Advanced Conformance Level, as we will present later.

The elements generated in each step could be customized by the user in order to either include other details not captured in our generation process or remove extra generated elements. For instance, an OData model generated from a PIM model could be enriched by adding other OData elements (e.g., OData annotations) or remove unwanted generated elements (e.g., an OData entity type generated from an unwanted UML class). Also, the generated application could be refined in order to customize the OData service or integrate other web functionalities not related to OData such as authentication.

7.2 Running Example

To illustrate our approach, we use as running example the UML class diagram shown in Figure 7.2 representing a data model to manage online stores. This example is inspired by the official reference example of OData[1]. The model includes two classes, namely: *Product*, which represents products; and *Supplier*, which represents the supplier of a product. The address of a supplier is defined using the data type *Address*. The bidirectional association between products and suppliers allows navigating from a product to a supplier (the association end *supplier*), and from a supplier to a list of products (the association end *products*).

[1] `http://services.odata.org/V4/OData/OData.svc/$metadata`

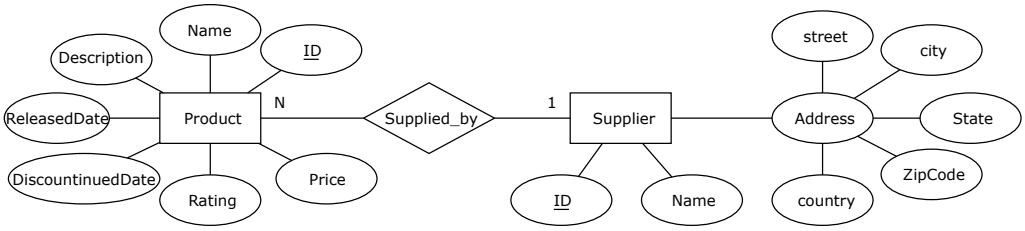


Figure 7.3: ER model of the running example.

We will also illustrate our approach using an ER model. Figure 7.3 shows the ER model representing the data model of the online store using the widely adopted Elmarsi & Navathe's notation [EN10] which is close to the classical notation defined by Chen. This model includes two entities, namely: *Product*, which represents products; and *Supplier*, which represents the supplier of a product. `Supplied_by` represents a binary relationship between the entities *Product* and *Supplier* meaning that a product is supplied by one supplier and a supplier can provide many products. Given either a UML or an ER model, our approach generates a ready-to-deploy OData Service exposing the OData metadata document representing the data model and serving client requests for both querying and updating data, which requires transforming OData requests to SQL statements and representing the data according to OData protocol (i.e., OData JSON [HPB14] and OData Atom [ZPH14] formats).

The OData metadata document is expressed using CSDL [PHZ14e]. Listing 7.1 shows an excerpt of the generated metadata document for the data model shown in Figure 7.3. The `Schema` element describes the entity model exposed by the OData Web service and includes the entity types `Product` and `Supplier`, and the complex type `Address`. Each type includes properties and navigation properties to describe attributes and relationships, respectively. The `Schema` element includes also an `EntityContainer` element defining the entity sets exposed by the service and therefore the entities that can be accessed. Web clients use this document to understand how to query and interact with the service. For instance, the request `GET http://host/service/Products?$filter=Price le 2.6` including the `$filter` option, should retrieve the list of *products* having the price less or equals to 2.6. Listing 7.2 shows the result of this request in OData JSON format.

Next sections will describe how our approach can be used to go from the original model to the deployed OData services following a model-driven approach.

Listing 7.1: A simple OData Metadata Document for the products service.

```
<edmx:Edmx xmlns:edmx="http://docs.oasis-open.org/odata/ns/edmx"
    Version="4.0">
  <edmx:DataServices>
    <Schema xmlns="http://docs.oasis-open.org/odata/ns/edm"
        Namespace="com.example.ODataDemo" Alias="ODataDemo">
      <EntityType Name="Product">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Description" Type="Edm.String"/>
        <Property Name="ReleasedDate" Type="Edm.DateTimeOffset"
            Nullable="false"/>
        <Property Name="DiscontinuedDate" Type="Edm.DateTimeOffset"
            />
        <Property Name="Rating" Type="Edm.Int16" Nullable="false"/>
        <Property Name="Price" Type="Edm.Double" Nullable="false"/>
        <NavigationProperty Name="Supplier" Type="ODataDemo.
            Supplier" Partner="Products"/>
      </EntityType>
      <EntityType Name="Supplier">
        <Key><PropertyRef Name="ID"/></Key>
        <Property Name="ID" Type="Edm.Int32" Nullable="false"/>
        <Property Name="Name" Type="Edm.String"/>
        <Property Name="Address" Type="ODataDemo.Address"/>
        <NavigationProperty Name="Products" Type="Collection(
            ODataDemo.Product)" Partner="Supplier" />
      </EntityType>
      <ComplexType Name="Address">
      <Property Name="Street" Type="Edm.String"/>...
      </ComplexType>
      <EntityContainer Name="DemoService">
        <EntitySet Name="Products" EntityType="ODataDemo.Product">
          <NavigationPropertyBinding Path="Supplier" Target="
              Suppliers"/>
        </EntitySet>
        <EntitySet Name="Suppliers" EntityType="ODataDemo.Supplier"
            >
          <NavigationPropertyBinding Path="Products" Target="
              Products"/>
        </EntitySet>
      </EntityContainer>
    </Schema>
  </edmx:DataServices>
</edmx:Edmx>
```

Listing 7.2: An example of collection of *products* in OData JSON format.

```
{
    "@odata.context": "$metadata#Products",
    "value": [
        {
            "ID": 1,
            "Name": "Milk",
            "Description": "Fresh milk",
            "ReleasedDate": "1992-01-01",
            "DiscountinuedDate": null,
            "Rating": 4,
            "Price": 2.40
        },
        {
            "ID": 2,
            ...
            "Price": 2.25
        }
    ]
}
```

7.3 Specification of OData Services

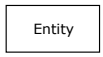
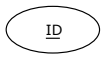
This section describes the specification of OData services using the OData metamodel presented in Chapter 3 (see Section 3.2). OData models can be automatically derived from UML models by means of a model-to-model transformation (see *UML2OData* transformation in Figure 7.1). We rely on plain UML models to generate OData models, thus no knowledge of OData is to be required. A similar approach, relying on the UML profile for OData presented in Chapter 3 (see Section 3.2), could also be followed to drive a custom transformation for enriched UML models.

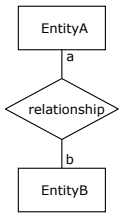
Table 7.1 shows a subset of the main transformation rules from UML metamodel elements to OData metamodel elements where the second column displays the source UML elements, third column shows the conditions to trigger the transformation, firth column shows the created/updated OData elements, and the last column shows the initialization values for the OData elements. Similarly, Table 7.2 shows a subset of the main transformation rules from ER metamodel elements (i.e., entities, attributes, relationships) to OData metamodel elements where the first column displays the ER elements (using the ER notation for the sake of clarity), second column shows the created/updated OData elements, and the last column shows the initialization values for the OData elements. In both scenarios, instances of the elements `ODEntityType`, `ODComplexType`, and `ODEnumType` are added

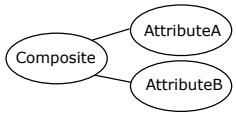
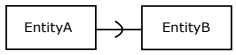
Table 7.1: UML to OData model transformation rules.

Ref.	Source elements	Conditions	Target elements	Initialization details
1	`c: Class`	-	`et: ODEntityType` `es: ODEntitySet`	- `et.name = c.name` - **if** `c.abstract = true` **then** `et.abstract` ← `true` - **if** `c.generalizations` contains a class `cc` **then** `et.baseType` ← `t` where `t` is the corresponding `EntityType` of `cc` - `et.properties` ← *(cf. rules to transform attributes, rows 3 and 4)* - `et.navigationProperties` ← *(cf. rule to transform navigable association ends, i.e., row 5)* - `es.name` ← the plural form of `c.name` - `es.entityType` ← `et` - `es.navigationPropertyBindings` ← *(cf. rule to transform navigable association ends, i.e., row 5)*
2	`dt: DataType`	-	`ct: ODComplexType`	- `ct.name` ← `dt.name` - **if** `ct.abstract = true` **then** `dt.abstract` ← `true` - **if** `dt.generalizations` contains a data type `dd` **then** `ct.baseType` ← `t` where `t` is the corresponding `ODComplexType` of `dd` - `ct.properties` ← *(cf. rules to transform attributes, i.e., rows 3 and 4)*
3	`p: Property`	`p` is a class attribute or a data type attribute	`op: ODProperty`	- `op.name` ← `p.name` - `op.type` ← `t` where `t` is the corresponding type of the attribute - **if** `p` is multivalued **then** `op.multivalued` ← `true`
4		`p` is a class attribute marked as `ID`	`pk: ODPropertyKeyRef`	- `pk.property = op`
5		`p` is a navigable association end	`np: ODNavigationProperty` `npb: ODNavigation-PropertyBinding`	- `np.name` ← `p.name` - `np.type` ← `t` where `t` is the corresponding entity type of `p.type` - **if** `p.aggregation = Composite` **then** `np.containsTarget` ← `true` - **if** `p` is multivalued is **then** `np.multivalued` ← `true` - `npb.path` ← `p.name` - `npb.target` ← `t.name` where `t` is the corresponding entity set of `p.type`
6	`e: Enumeration`	-	`oe: ODEnumType`	- `oe.name` ← `e.name` - `oe.members` ← *(cf. rule to transform literals, i.e., row 7)*
7	`el: EnumerationLiteral`		`om: ODMember`	- `om.name` ← `el.name`

Table 7.2: ER to OData model transformation rules.

ER CONSTRUCTORS	TARGET ELEMENTS	INITIALIZATION DETAILS
	`et: ODEntityType`, `es: ODEntitySet`	- `et.name = "Entity"` - `es.name = "Entities"` - `es.entityType = et`
	`p: ODProperty`	- `p.name = "Attribute"` - `p.type = t` (`t` is the corresponding primitive type of the attribute)
	`p: ODProperty` `pk: ODPropertyKeyRef`	- `p.name = "ID"` - `p.type = t` (`t` is the corresponding primitive type of the attribute) - `pk.property = p`
	`p: ODProperty`	- `p.name = "Attribute"` - `p.type = t` (`t` is the corresponding primitive type of the attribute) - `p.multivalued = true`

	`nb:`
	`ODNavigationProperty`,
	`na: ODNavigationProperty`
		- `nb.name = "EntityB"` (`nb` is a property of the corresponding entity type of `EntityA`)
		- `nb.type = t1` (`t1` is the corresponding entity type of `EntityB`
		- if `b==N` then `nb.multivalued = true`
		- `na.name = "EntityA"` (`na` is a property of the corresponding entity type of `EntityB`)
		- `na.type = t2` (`t2` is the corresponding entity type of `EntityA`
		- if `a==N` then `na.multivalued = true`

	`c: ODComplexType`	- `c.name = "Composite"`
	`b: ODEntityType`	- `b.baseType = t` where `t` is the corresponding `EntityType` of `EntityA`

to the element `ODSchema` once they are created, and likewise instances of `ODEntitySet` are added to the element `ODEntityContainer`. Furthermore, each instance of the elements `ODProperty` and `ODNavigationProperty` are added to its corresponding `ODEntityType`, `ODComplexType` element, and likewise each instance of `ODNavigationPropertyBinding` is added to its corresponding `ODEntitySet`. Figure 7.4 shows an excerpt of the generated OData model for the `Product` entity (with only the `ID` attribute) of our running example (see Figures 7.2 and 7.3).

7.4 DATABASE SCHEMA GENERATION

OData is designed to work on a variety of data stores. In particular, the protocol does not necessarily assume a relational data model. We defined

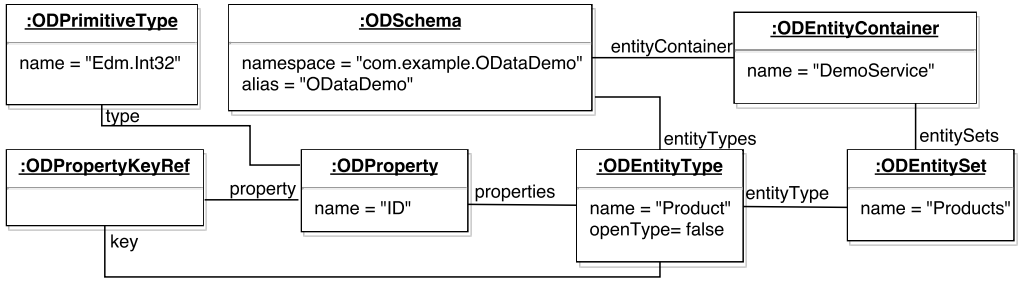


Figure 7.4: An excerpt of the generated OData model for the running example.

an algorithm to generate a relational database schema from an OData data model, which we describe in this section (see *DatabaseGen* transformation in Figure 7.1). This algorithm is heavily inspired by the typical transformation rules to derive database schemas from UML models (e.g., GENMYMODEL[2], UMLTOX[3]) or ER models (e.g. see Fidalgo et al. [Fid+12], ER2SQL[4]).

Algorithm 1 illustrates the OData data model to database schema generation process. As can be seen, the algorithm takes as input an instance of `ODSchema` and returns a DDL script representing the data model. The first part of the algorithm (i.e., from line 1 to line 37) iterates over the contained entity types and complex types then generates a `CREATE` command for each element not having a super type (i.e., `baseType = null`). The algorithm adds an extra column `id` for each complex type to define a primary key (cf. line 5) and an extra column `discriminator` for each complex or entity type having subtypes to identify the concrete type of the element (cf. line 8). Furthermore, for each single valued property or navigation property the algorithm generates a column statement (cf. line 12). Moreover, the algorithm generates a `CREATE` command for each multivalued property (cf. line 21) and many to many navigation property (cf. line 29). The second part of the algorithm (i.e., from line 38 to line 46) iterates over the navigation properties of each entity type and complex type then generates an `ALTER TABLE` command to declare a foreign key. The algorithm relies on the functions TABLENAME, COLUMNTYPE, COLUMNNAME, GETNULL and REFERENCE (see footnotes on Algorithm 1). Note that for the sake of simplicity, the algorithm assumes that the key of each entity type is represented by a single property. Listing 7.3 shows an excerpt of the DDL script to create the

[2] `https://www.genmymodel.com/`
[3] `https://github.com/jcabot/UMLtoX`
[4] `http://er2sql.sourceforge.net/`

Algorithm 1 APIGENERATOR: DDL schema generation process.

Input:
s where s is an instance of `ODSchema`
Output:
q where q is the DDL script of the database

```
1: S ← s.entityTypes ∪ s.complexTypes
2: for i = 1 to S.length do
3:     if S[i] does not have a super type then
4:         q ← q + "CREATE TABLE" + TABLENAME(S[i]) + "("
5:         if S[i] is instance of ODComplexType then
6:             q ← q + "id INT not null,"
7:         end if
8:         if S[i] is a super type then
9:             q ← q + "discriminator VARCHAR(255),"
10:        end if
11:        P ← S[i].properties ∪ S[i].navigationProperties
12:        for j = 1 to P.length do
13:            if P[j] is single valued then
14:                q ← q + COLUMNNAME(P[j]) + COLUMNTYPE(P[j]) + GETNULL(P[j]) + ","
15:            end if
16:        end for
17:        q ← q + "PRIMARY KEY (" + PRIMARY(S[i]) + "));"
18:    end if
19:    for j = 1 to S[i].properties.length do
20:        if S[i].properties[j] is multivalued then
21:            q ← q + "CREATE TABLE" + TABLENAME(S[i].properties[j])
22:            +"(id INT not null," + COLUMNNAME(S[i].properties[j])
23:            +COLUMNTYPE(S[i].properties[j]) +","
24:            +TABLENAME(S[i]) + "_id" + PRIMARYTYPE(S[i]) +","
25:            +"PRIMARY KEY (id));"
26:        end if
27:    end for
28:    for j = 1 to S[i].navigationProperties.length do
29:        if S[i].navigationProperties[j] is many to many then
30:            q ← q + "CREATE TABLE"
31:            +TABLENAME(S[i].navigationProperties[j]) + "(" + "id INT not null,"
32:            +TABLENAME(S[i]) + "_id," + PRIMARYTYPE(S[i])+","
33:            +TABLENAME(S[i].navigationProperties[j].partner)+"_id,"
34:            +PRIMARYTYPE(S[i].navigationProperties[j].partner)+"," + "PRIMARY KEY (id));"
35:        end if
36:    end for
37: end for
38: for i = 1 to S.length do
39:    for j = 1 to S[i].navigationProperties.length do
40:        if S[i].navigationProperties[j] is single valued then
41:            q ← q + "ALTER TABLE" + TABLENAME(S[i]) +"ADD FOREIGN KEY"
42:            +COLUMNNAME(S[i].navigationProperties[j])
43:            +"REFERENCES" + REFERENCE(S[i].navigationProperties[j]) + ";"
44:        end if
45:    end for
46: end for
```

TABLENAME gets a table name according to database recommendations representing the input element.

COLUMNTYPE gets a database primitive type representing the type of the input element.

COLUMNNAME gets a column name according database recommendations representing input element.

GETNULL generates `NOT NULL` statement if the input parameter is required.

PRIMARY gets the key column of the input element

PRIMARYTYPE gets the type of the key column of the input element

REFERENCE generates the target of `REFERENCES` statement of the input element.

Listing 7.3: A simple DDL file of the running example.

```
CREATE TABLE product (
    id INT not null,
    name VARCHAR(255),
    description VARCHAR(255),
    releaseddate DATE not null,
    discountinueddate DATE,
    rating INT not null,
    price DECIMAL (10,2) not null,
    supplier_id INT,
    PRIMARY KEY (id));
CREATE TABLE supplier (
    id INT not null,
    name VARCHAR(255)
    address_id INT,
    PRIMARY KEY (id));
CREATE TABLE address (
    id INT not null,
    street VARCHAR(255),...
    PRIMARY KEY (id));
ALTER TABLE product
ADD FOREIGN KEY (supplier_id) REFERENCES supplier(id);
ALTER TABLE supplier
ADD FOREIGN KEY (address_id) REFERENCES address(id);
```

database schema corresponding to the running example after applying the algorithm.

7.5 OData Service Generation

In this section we describe the generation process of OData services from OData models (see the *ODataGen* transformation in Figure 7.1). Our process includes the generation of (1) the metadata document, (2) the mapping between OData requests and SQL statements (see *OData2SQL* component in Figure 7.1), and (3) the de/serialization process (see *ODataSerializer* component in Figure 7.1).

7.5.1 *OData Metadata Document Generation*

This process transforms an OData model into an OData metadata document by means of a model-to-text transformation. This document helps clients discover the data schema exposed by the service and therefore build OData queries.

The generation process is nearly straightforward as our metamodel follows the OData CSDL specification and only special attention had to be paid when generating references among elements. Thus, the transformation iterates over OData model elements and generates their XML representation (e.g., `Schema` for the element `ODSchema`, `EntityContainer` for the `ODEntityContainer`, etc.) taking into account its specification (e.g., name, type, etc.). The resulting document is an XML file represented using the CSDL language [PHZ14e] and can be retrieved by appending `$metadata` to the root URL of an OData service. An example of this file has been previously shown in Listing 7.1.

7.5.2 *OData Requests to SQL Statements Transformation*

The OData specification defines standard rules to query data via HTTP `GET` requests and perform data modification actions via HTTP `POST`, `PUT`, `PATCH`, and `DELETE` requests. A URL of an OData request has three parts [PHZ14d]: (1) the service root URL, which identifies the root of an OData service; (2) a target resource path, which identifies a resource to query or update (e.g., products, a single product, supplier of a product); and (3) a set of query options.

$$\underbrace{\texttt{GET}}_{\text{HTTP method}} \quad \underbrace{\texttt{http://host/service}}_{\text{service root URL}} / \underbrace{\texttt{Suppliers(1)/Products}}_{\text{target resource path}} ? \underbrace{\texttt{\$top=2\&orderby=Name}}_{\text{query options}} \tag{7.1}$$

To transform OData requests to SQL statements we consider the HTTP method, which specifies whether the request is either a query or a data modification action; the resource path, and the query options part. In the following we describe how these elements drive the SQL statement generation process, in particular, how we deal with the target resource paths, query options, and data modification actions.

Target Resource Path URL Transformation. The target resource path in an OData request can address (1) a collection of entities, (2) a single entity, (3) and a property, which we will illustrate by means of examples. Table 7.3 shows a set of examples of requests relying on our running example for all the CRUD operations. The second column shows the used HTTP methods. The third column explains the type of the request. The fourth and fifth columns show an example of the OData request including the resource path and request body, and the corresponding SQL query to the database, respectively.

Example 1 illustrates a request to access to a collection of entities (i.e.,

Table 7.3: Examples of OData requests and the corresponding SQL statements.

Ref	HTTP method	Description	Resource path example	SQL query
1	GET	Request a collection of entities	`GET http://host/service/Products`	`SELECT * FROM product p`
2		Request a single entity by ID	`GET http://host/service/Products(1)`	`SELECT * from product p WHERE p.id = 1`
3		Request an individual property	`GET http://host/service/Products(1)/Price`	`SELECT p.price from product p WHERE p.id = 1`
4		Request an entity collection by following a navigation from an entity to another related entity	`GET http://host/service/Suppliers(1)/Products`	`SELECT p.* from product p JOIN supplier s on p.supplier_id = s.id WHERE s.id = 1`
5	POST	Create a new entity	`POST http://host/service/Products { texttt"ID": 1; "Name": "Milk", "Description": "Fresh milk",...}`	`INSERT INTO product (id, name, description, ...) VALUES (1, 'Milk', 'Fresh milk', ...)`
6	PATCH	Update an entity	`PATCH http://host/service/Products(1) { "Description": "Very fresh milk" }`	`UPDATE product SET description = 'Very fresh milk' WHERE id = 1`
7	PUT	Update a navigation property	`PUT http://host/service/Products(1)/Supplier/$ref { "@odata.id": "http://host/service/Suppliers(2)" }`	`UPDATE product SET supplier_id = 2 WHERE id = 1`
8	DELETE	Delete an entity	`DELETE http://host/service/Products(1)`	`DELETE FROM product WHERE id = 1`

collection of products), which is transformed into a SELECT SQL statement for the corresponding table of the entity exposed by the addressed entity set. Example 5 also illustrates a request to add a new entity into the target collection of entities.

Example 2 shows how a single entity can be accessed by adding the entity key as path segment (i.e., the product of ID 1), which requires adding a WHERE clause to the SELECT statement. Likewise, examples 6 and 8 illustrate how to update and delete a single entity, respectively.

Example 3 shows how to access an entity property (i.e., the price of the product), which requires adding the corresponding column name to the SELECT statement.

Finally, example 4 illustrates a request to access to a collection of entities by navigating from an entity to another one (i.e., the collection of products of a specific supplier), which requires adding one or more JOIN clauses to the SELECT statement depending on the cardinalities of the navigation properties (i.e., one to many, many to many) and the hierarchy of the resource.

To perform the transformation of the target resource path into the corresponding SQL statement we devised Algorithm 2. The algorithm takes as input the set of entity types referenced in the URL, the set of the properties used to navigate from one entity to another in the path, a set of path segments to specify the key of a particular entity and the name of the final property to retrieve. Last two parameters are optional.

The algorithm is divided into three parts. The first part (i.e., from line 1 to line 4) retrieves the database tables corresponding to each entity. The second part (i.e., from line 5 to line 31) iterates over the entities and constructs the SELECT statement and the JOIN clauses depending on the number and the type of the navigations (i.e., one-to-one, one-to-many, many-to-one, many-to-many). Finally, the third part (i.e., from line 32 to line 38) constructs the WHERE clause. The algorithm relies on the functions TABLENAME, TABLEALIAS, COLUMNNAME, and HASNEXT (see footnotes on Algorithm 2).

The first row of Table 7.4 illustrates the execution of this algorithm for the URL shown on the left. This URL identifies the price of the product 1 belonging to the supplier 1. The corresponding input parameters are: $E = \{E_1 : Supplier, E_2 : Product\}$, $N = \{N_{1,2} : Products\}$, $x = \{x_1 : 1, x_2 : 3\}$, and $p = Price$. The resulting SQL query is shown on the right.

Query Transformation. OData allows querying data via HTTP GET requests to the resources addressed by a resource path (as shown before). OData queries can include a set of options which are string parameters

Algorithm 2 APIGENERATOR: Resource path URL transformation.

Input:
$E = \{E_1, E_2, ..., E_n\}$ where E_i is the entity at the index i and n is the total number of entities
$N = \{N_{1,2}, N_{2,3}, ..., N_{n-1,n}\}$ where $N_{i-1,i}$ is the navigation property from E_{i-1} to E_i.
$x = \{x_1, x_2, ..., x_n\}$ where x_i is the key of the entity E_i if presented in the URL or $\varnothing$ otherwise
p where p corresponds to the name of a particular property to retrieve or $\varnothing$
Output:
q where q is an SQL query representing the resource of the input

1: **for** $i = 1$ to n **do**
2: $T_i \leftarrow$ TABLENAME(E_i)
3: $A_i \leftarrow$ TABLEALIAS(E_i)
4: **end for**
5: **if** $p <> \varnothing$ **then**
6: $q \leftarrow$ +"SELECT" + A_n +"." + COLUMNNAME(p) + "FROM" + T_n + A_n
7: **else**
8: $q \leftarrow$ + "SELECT" + A_n + ".* FROM" + T_n + A_n
9: **end if**
10: $i \leftarrow n$
11: **while** $i <> 1$ **do**
12: **if** $N_{i-1,i}$ is many to one **then**
13: Let $C_{i-1}^{i_{fk}}$ be the column representing the foreign key of T_i in T_{i-1} and C_i^{pk} the primary key of T_i
14: $q \leftarrow q$ + "JOIN" + T_{i-1} + A_{i-1} + "ON" + A_{i-1} +"." + $C_{i-1}^{i_{fk}}$
15: + "=" + A_i + "." + C_i^{pk}
16: **else**
17: **if** $N_{i-1,i}$ is one to many **then**
18: Let $C_i^{i-1_{fk}}$ be the column representing the foreign key of T_{i-1} in T_i and C_{i-1}^{pk} the primary key of T_{i-1}
19: $q \leftarrow q$ + "JOIN" + T_{i-1} + A_{i-1} + "ON" + A_i + "." + $C_i^{i-1_{fk}}$ + "=" + A_{i-1} + "." + C_{i-1}^{pk}
20: **else**
21: **if** $T_{i,i-1}$ is many to many **then**
22: Let $J_{i,i-1}$ be the association table between T_i and T_{i-1}, $A_{i,i-1}$ the alias name $J_{i,i-1}$, $C_{i,i-1}^{i_{fk}}$ the column representing the foreign key of T_i in $J_{i,i-1}$, and $C_{i,i-1}^{i-1_{fk}}$ the column representing the foreign key of T_{i-1} in $J_{i,i-1}$
23: $q \leftarrow q$ + "JOIN" + $J_{i,i-1}$ $A_{i,i-1}$ + "ON" + $A_{i,i-1}$ +"."
24: + $C_{i,i-1}^{i_{fk}}$ "=" + A_i + "." + C_i^{pk} "JOIN" + T_{i-1} A_{i-1}
25: + "ON" + $A_{i,i-1}$" + "." + $C_{i,i-1}^{i-1_{fk}}$ "=" + A_{i-1} + "."
26: + C_{i-1}^{pk}
27: **end if**
28: **end if**
29: **end if**
30: $i \leftarrow i - 1$
31: **end while**
32: $q \leftarrow q$ + "WHERE"
33: **for** $i = 1$ to n **do**
34: $q \leftarrow q + A_e$ + "." + C_e^{pk} + "=" + x_i^e
35: **if** HASNEXT(x_i^e) **then**
36: $q \leftarrow q$ + "AND"
37: **end if**
38: **end for**

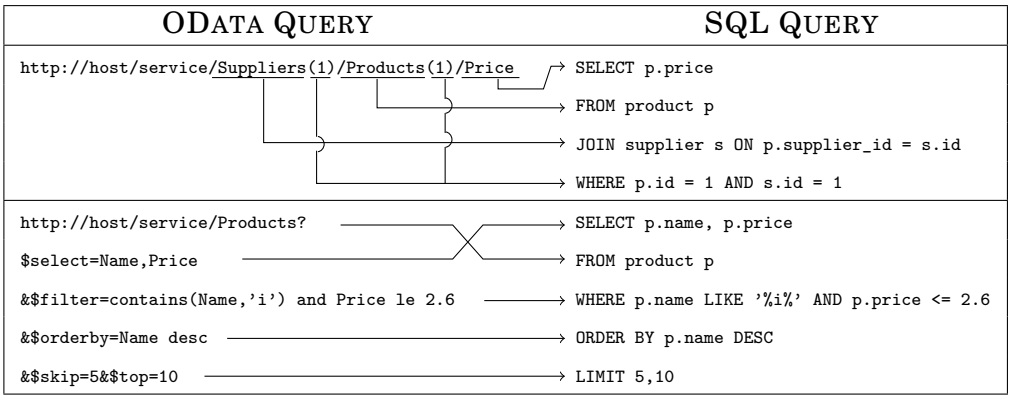
TABLENAME gets a table name according to database recommendations representing the input element.

TABLEALIAS gets an alias name for the input element.

COLUMNNAME gets a column name according database recommendations representing input element.

HASNEXT returns `true` if there is a key in the queue and `false` otherwise.

Table 7.4: Example of OData request to SQL mapping.



prefixed by a $ symbol that control the amount and order of the returned data for a resource. Query options can be used to refine the result of an OData request and therefore are also considered in our transformation. Table 7.5 shows the query options provided by OData and the corresponding mapping rules to generate the SQL code. For each query option, the table provides a small description, the mapping rule with SQL, an example, and the corresponding SQL query.

OData also defines (1) logical and arithmetic operators (e.g., `eq` for equals or `add` for addition) to use with the `$filter` query option, (2) utility functions for string, date and time management (e.g., `concat`, `hour()` or `now()`), and (3) combining query options to create advanced queries. Our approach covers the operators and functions supported by MySQL database[5].

The second row of Table 7.4 shows an example of query option mapping. On the left, the Table shows a URL example to retrieve 10 records after the position 5 ordered by name from the collection of products which have a name containing the character `i` and a price less or equals to `2.6`; while on the right, the Table lists the corresponding SQL query.

Data Modification Transformation. To perform data modification actions, OData relies on the HTTP `POST`, `PUT`, `PATCH` and `DELETE` requests. To support data modification actions, we generate a set of controllers which process the client requests and generate the corresponding SQL statements according to the specification.

Table 7.3 shows the usage of HTTP methods in OData illustrated with examples. To create an entity, the client must send a `POST` request contain-

[5]More details can be found at `https://github.com/SOM-Research/odata-generator`.

Table 7.5: OData System query options and their corresponding SQL rules.

OPTION	DESCRIPTION	SQL RULE	QUERY OPTION EXAMPLE	SQL EXAMPLE
`$filter`	Filter a collection of resources that are addressable by a request url	Add a `WHERE` clause including the corresponding operator to the filter expression	`http://host/service/Products?$filter=Name eq 'Milk'`	`SELECT * FROM product p WHERE p.name LIKE 'Milk'`
`$expand`	Include relative resource in line with retrieved resources	For each retrieved resource, create a `SELECT` statement to retrieve the relative resource	`http://host/service/Suppliers(1)?$expand=Products`	`SELECT * FROM product p WHERE supplier_id = 1`
`$select`	Request a specific set of properties	Add the corresponding column names to the `SELECT` statement	`http://host/service/Products?$select=Name, Price`	`SELECT p.name, p.price FROM product p`
`$orderby`	Request resources in either ascending order using `asc` or descending order using `desc`	Add the `ORDER BY` clause to the `SELECT` statement	`http://host/service/Products?$orderby='Name'desc`	`SELECT * FROM product p ORDER BY p.name DESC`
`$top` `$skip`	`$top` requests the number of items to be included and `$skip` requests the number of items to be skipped	Add the supported clause by the database system (e.g., `LIMIT` for MySQL)	`http://host/service/Products?$top=10$skip=5`	`SELECT * FROM product p LIMIT=5,10`
`$count`	Request a count of the resources	Add the `COUNT` function to the `SELECT` statement	`http://host/service/Product?$count`	`SELECT COUNT(*) FROM product`
`$search`	Request entities matching a free text search	Add a set of `LIKE` operators to the `WHERE` clause to much the search text with all the text columns of the corresponding table	`http://host/service/Product?$search='Milk'`	`SELECT * FROM product p WHERE p.name LIKE 'Milk' OR p.description LIKE 'Milk'`

ing a valid representation of the new entity to the URL of a collection of entities (i.e., *EntitySet*, e.g., *Products*). This request is transformed to an `INSERT` statement (see example 5). To update an entity, the client must send a `PATCH` request containing a valid representation of the properties to update to the URL of a single entity (e.g., *Products(1)*). This request is transformed to an `UPDATE` statement (see example 6). To update a navigation property, the client must send a `PUT` request containing the URL of the new related

entity to the URL of the reference[6] of a single-valued navigation property (e.g., *Products(1)/Supplier/$ref*). This request is transformed to an `UPDATE` statement (see example 7). Finally to remove an entity, the client must send a `DELETE` request to the URL of an individual entity (e.g., *Products(1)*). This request is transformed to a `DELETE` statement. Next, we explain the serialization and deserialization mechanisms of the requests.

7.5.3 *OData Serializer and Deserializer Generation*

This process generates a serializer and a deserializer for OData objects supporting both the OData JSON [HPB14] and Atom [ZPH14] formats.

The serializer applies a model-to-text transformation to the result of OData requests (i.e., entity collection, entity and property) in order to generate the textual representation according to OData format conventions. For instance, in the case of JSON, an entity collection is transformed to a JSON array holding the entities while an entity is represented by a JSON object containing a list of key/value pairs representing its properties. A similar process is followed for the Atom representation format. OData representation formats also support different levels (and content) of metadata (i.e., `full`, `minimal` or `none`) [HPB14; ZPH14], which can be configured in the header of any OData request. The generated serializer also takes into account this setting and generates the JSON or Atom representation accordingly. An example of a collection of *products* in OData JSON format was previously shown in Listing 7.2. As can be seen, apart from the properties of the entity, the JSON object also includes as metadata the annotation `odata.context` which indicates the root context URL of the payload.

The deserializer processes and parses the body of the OData requests `POST`, `PUT` and `PATCH` in order to generate the details of the `INSERT` and `UPDATE` SQL statements, accordingly. For instance, in the case of JSON (see the examples 5, 6 and 7 in Table 7.3), each key and value in the JSON object are transformed to the corresponding field in the corresponding table and the value for such field, respectively, in the generated SQL statement. A similar process is followed for the Atom representation format.

7.6 TOOL SUPPORT

Our approach is available as a proof-of-concept plugin for the Eclipse platform[7]. The plugin extends the platform to provide contextual menus to

[6] A reference is specified by adding `$ref` to the resource path of the navigation property

[7] `https://github.com/SOM-Research/odata-generator`

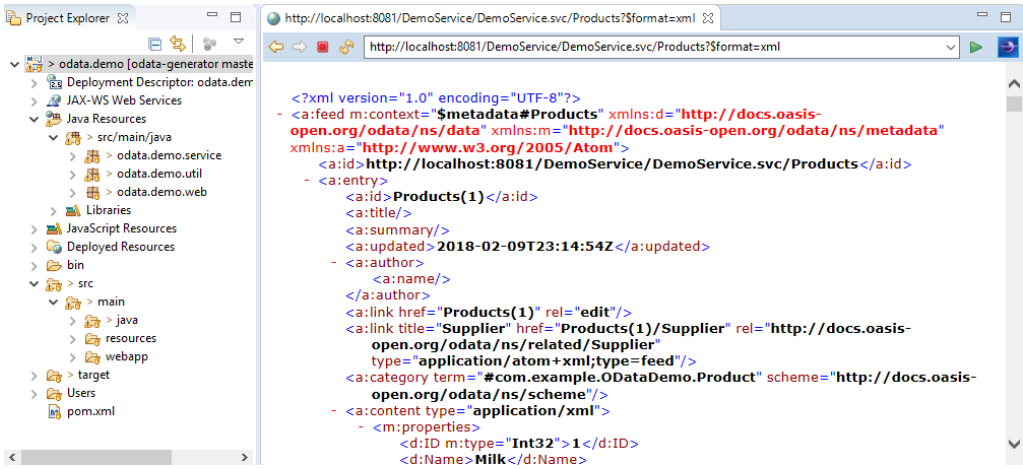


Figure 7.5: APIGENERATOR: a screenshot of the generated application for the running example.

obtain OData models from existing UML models and, given an OData model instance: (1) generate the metadata document conforming to OData specification; (2) generate the DDL of the database; and (3) generate an OData service based on a Maven-based project.

The UML to OData model transformation relies on UML2[8] which provides an EMF-based implementation of the UML 2.5 OMG metamodel; while the code generators are based on ACCELEO[9], an implementation of the MOF Model-To-Text Transformation Language (MTL) specification from OMG, to define the logic and generate the different OData artifacts.

The generated Web application includes a properties file with the configuration of the database. The OData service implementation relies on APACHE OLINGO[10] to provide support for OData query language and serialization; and JOOQ[11] which provides a DDL to build SQL queries. The implementation includes controllers to analyze and deserialize the requests, transform them into SQL queries, execute the queries, and serialize the result to be sent back to the client.

Figure 7.5 shows a screenshot of the generated application for the running example. The figure includes the structure of the Maven project (see left panel) and a browser showing the result of request in Atom format (see right panel). The service currently implements the support for: (1) resource path

[8] https://wiki.eclipse.org/MDT/UML2
[9] https://www.eclipse.org/acceleo/
[10] http://olingo.apache.org
[11] https://www.jooq.org

URL transformation; (2) data querying using the query options `$filter`, `$top`, `$skip` and `$orderby`; and (3) data modification as described in the previous section. The generated application returns `501, "Not Implemented"` for any unsupported functionality as required by the protocol. The complete generated application can be found in our repository[12]. The repository includes also the steps to install the plugin, generate the OData service, and deploy the generated application in a Servlet container.

7.7 Related work

Model-driven approaches have been widely used in the Web Engineering field to generate different kinds of web applications (e.g., [Sch+08; VP11; Fra99; SCL14; Hau+14; Riv+14; Rod+13; Val+07]). While existing approaches already provide methodologies and tools to cover a variety of technologies (e.g., web services, ubiquitous applications), specific support for Web APIs is rather limited. For instance, Porres et al. [PR11] and Tavares et al. [TV13b] propose to model REST APIs using UML and a REST metamodel, respectively, but only generate a WADL document [Had06] describing the behavior of a REST API and do not generate the API implementation. A few exceptions to generate REST APIs are: (1) EMF-REST presented in Chapter 4 (but for generating web modeling environments); (2) ODaaS [SCL14] (for the exploitation of existing Open Data and social media streams); (3) ELECTRA [Riv+14] and MockAPI [Riv+13b] (for fast prototyping of mockup APIs); (4) the work by Rodríguez-Echeverría et al. [Rod+13] (deriving REST APIs from legacy Web applications); and (5) MicroBuilder [Ter+17] and the work by Haupt et al. [Hau+14] (using ad-hoc DSLs for the specification and the realization of REST APIs). None of them, though, have explicit support neither for modeling OData nor for automatically generating OData Services from high-level models.

Some SDKs provide support for developing OData applications for a target platform (e.g., RESTier[13], Apache Olingo[14], SDL OData Frameworks[15]). These frameworks are handy for developers but require knowledge to deal with the intricacies of their architecture[16] to create OData applications.

[12] `https://github.com/SOM-Research/odata-generator`

[13] `https://github.com/OData/RESTier`

[14] `https://olingo.apache.org/`

[15] `https://github.com/sdl/odata`

[16] There are 129 open issues in GitHub regarding RESTier and StackOverFlow lists 396 questions regarding Olingo.

Support for generating OData applications is so far limited to commercial tools like CLOUD DRIVERS[17], ODATA SERVER[18] or SKYVIA CONNECT[19]. Still, these solutions only offer ways to expose OData services from already existing data sources such as databases but not to create new OData services from scratch nor to configure the full support to OData specification. In fact, OData services generated from relational databases just mirror the data structure (i.e., tables and relationships to entities and navigation entities, respectively), thus not leveraging on OData protocol which supports richer data structures (e.g., hierarchies, complex type or multivalued properties) and capabilities. For instance, we tested the trial version of ODATA SERVER to create an OData server for the MySQL database of our running example. Besides the limitation regarding the use of richer data structures (e.g., `Address` was transformed to an `Entity` and not a `ComplexType`), we also detected other issues related to Open API capabilities: (1) there is no entity container and therefore clients are not able to query the data; (2) the foreign keys are plain properties instead of navigation properties; (3) data is read-only.

Other tools such as SIMPLE-ODATA-SERVER[20] and JAYDATA[21] allow generating a basic OData server but require providing both an OData Entity model of the desired application and the corresponding database. Also, they only support a subset of the query options offered by OData protocol. On the other hand, our approach has advanced support for OData protocol and provides the database implementation of the data model out of the box.

7.8 SUMMARY

In this chapter we have presented a model-driven approach to specify and generate OData services. UML and ER models are used to generate the required artifacts to deploy OData services relying on a relational database as storage solution. The generation process covers the specification of the OData metadata document, the database schema, the resolution of URL requests into SQL statements and a de/serialization mechanism for the exchanged messages.

[17] `http://www.cdata.com/odata/`
[18] `https://rwad-tech.com/`
[19] `https://skyvia.com/connect/`
[20] `https://github.com/pofider/node-simple-odata-server`
[21] `https://github.com/jaystack/jaydata`

CHAPTER

8 Composing REST APIs

In this chapter we present APICOMPOSER, a lightweight model-driven approach to automatically compose data-oriented REST APIs (i.e., REST APIs exposing data). APICOMPOSER completes the picture of our model-driven approach and enables the composition of REST APIs by relying on the their definitions. EMF-REST and APIGENERATOR could be used to provide the input APIs, while APIDISCOVERER could be used to discover API definitions, and APITESTER could be used to ensure the correctness of such definitions. APICOMPOSER takes as input a set of OpenAPI definitions which are then processed to create a global API exposed as an OData application.

The rest of this chapter is organized as follows. Section 8.1 describes our approach, while Sections 8.2 and 8.3 explain its main steps. Section 8.4 illustrates our approach using an example. Section 8.5 presents our tool support. Section 8.6 discusses some related works. Finally, Section 8.7 summarizes this chapter.

8.1 OUR APPROACH

We propose a model-driven approach to compose data-driven REST APIs. From a set of initial REST APIs, our approach creates a global API exposing a unified data model merging the data models of the initial APIs. The global model is exposed as an OData application, thus allowing end-users to use the OData query language to get the information they need in an easy and

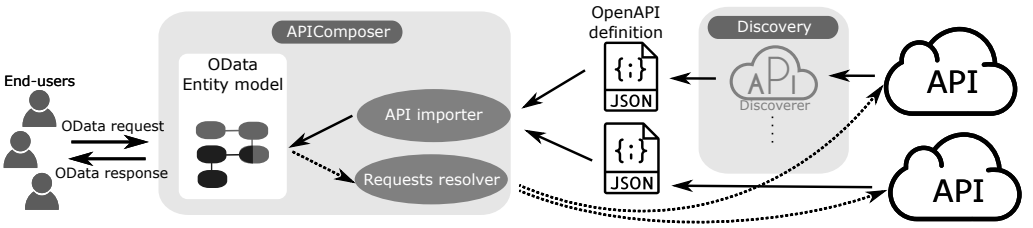


Figure 8.1: Overview of the APICOMPOSER approach.

standard way. Each OData query is translated into a combined sequence of requests to the underlying APIs which are then executed, combined and sent back to the user in OData format.

Figure 8.1 shows an overview of our approach. APICOMPOSER takes as input the OpenAPI definitions of the REST APIs to be composed. Such definitions may be (i) supplied by the API provider, (ii) generated using tools such as APIDiscoverer (see Chapter 5) which is able to infer OpenAPI definitions from API call examples; or (iii) derived from other API definition formats (e.g., API Blueprint or RAML) using tools such API Transformer[1].

Our approach relies on a model-based infrastructure which includes two components, namely: (i) *API importer*, in charge of integrating a new REST API to the global API; and (ii) *Requests resolver*, responsible for processing the user requests and returning the queried data. We explain each component in the following sections.

8.2 API IMPORTER

Figure 8.2 shows the *API importer* process. For each input OpenAPI definition, the *API importer* first generates an equivalent model conforming to our OpenAPI metamodel (see step 1 in Figure 8.2). We previously introduced this metamodel alongside the extraction process in Chapter 3.

The second step of the process (see step 2 in Figure 8.2) performs a model-to-model transformation to generate a UML model, which emphasizes the data schema of the input API to facilitate the matching process later on. This process consists on iterating over the data structures in the OpenAPI model (i.e., the *schema* elements) to generate the adequate UML elements (i.e. classes, properties and associations elements). This process relies on our tool OPENAPITOUML[2] presented in Chapter 3.

[1] http://apimatic.io/transformer
[2] https://github.com/SOM-Research/openapi-to-uml

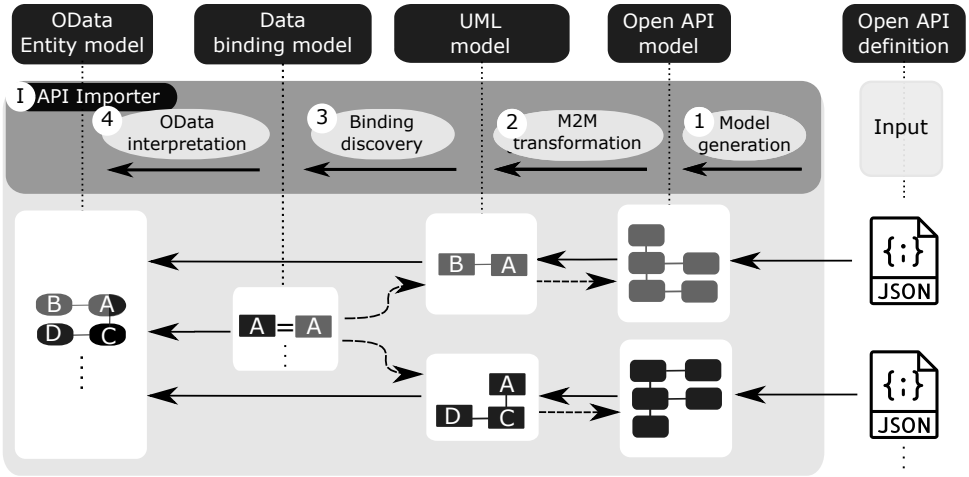


Figure 8.2: Composition process.

The third step (see step 3 in Figure 8.2) analyzes the UML models to discover matching elements and creates bindings to express the matches between them. The binding model conforms to the binding metamodel which allows creating traceability and binding elements for the data elements in the UML models.

Figure 8.3 shows an excerpt of the binding metamodel. The `BindingModel` element is the root element of the binding metamodel and includes a set of binding elements (i.e., `bindingElments` reference). The `ClassBinding`, `PropertyBinding`, and `AssociationBinding` elements allow defining bindings to `Class`, `Property`, and `Association` elements in a UML model, respectively. Each element includes a preferred name (i.e., the `preferredName` attribute inherited from the `BindingElement` element) and a set of binded elements (i.e., the `binded` references).

We currently support a simple two-step matching strategy to define the bindings between elements. The first step finds matching candidates based on their names and types. Then, the second step validates the matches by calling the REST APIs and comparing data related to each candidate. Our experience showed that such strategy is sufficient for APIs coming from the same provider/domain, which share the same concept names across their APIs. However, our approach can be extended in order to support more advanced matching strategies specially for cross-domain composition by relying on, for instance, database schema integration approaches [Bor+07] or the new approaches to add semantic descriptions to OpenAPI [CD17; Mus+16]. Also, a designer can manually curate the initial automatic result.

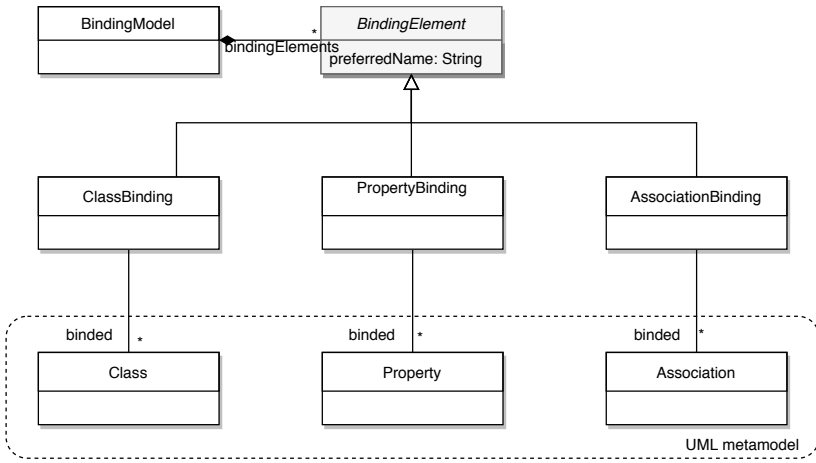


Figure 8.3: Excerpt of the binding metamodel.

Finally, the last step creates an OData metadata document from: (i) the generated UML models, and (ii) the binding model. This document includes an OData entity model created by merging all the data models of the input REST APIs and resolving the bindings between them. Thus, the creation process iterates over all the data elements in the UML models and creates a new element in the entity model if there is not a binding linking such element to another element, or merging both elements otherwise. The OData metadata document is the standard way OData provides to let end-users know how to query data using the OData query language.

8.3 Requests Resolver

The *Requests resolver* is an OData application exposing the created metadata document, and in charge of processing the end-user queries and building the query response based on the bindings and OpenAPI models generated during the import phase. Such process involves two steps, namely: query resolution and response resolution.

The query resolution phase interprets first the OData query in order to determine the target resource to retrieve (i.e., a collection of entities, a single entity or a property) and the options associated with the query (e.g., filter or ordering). The resolver transforms then the query into a set of API calls by tracing back the origin of each element thanks to the binding model. From the binding model we navigate first to the UML models then to the OpenAPI models. These OpenAPI models contain all the necessary details

to generate the actual calls[3] as they contain the same information as the original OpenAPI definitions.

On the other hand, the response resolution phase is in charge of providing the result to the end-user by combining the different API answers in a single response conforming to the OData entity model defined in the OData metadata document. In the next section we will illustrate our approach using an example.

8.4 EXAMPLE

To illustrate our approach, we consider the following REST APIs: BATTUTA[4], which allows retrieving the regions and cities of a country; and RESTCOUNTRIES[5], which allows getting general information about countries such as their languages, currencies and population. Our goal with this example is to create a global API combining both APIs. Thanks to the global API, users will be able to query both kinds of country information (either geographical, general or both) in a transparent way, (i.e., without having to specify in each OData query what API/s the query should read from). As a preliminary step, we generated the OpenAPI definitions describing BATTUTA and RESTCOUNTRIES APIs using APIDiscoverer. We used the resulting definitions as inputs for our approach.

Figure 8.4 illustrates the results of applying our composition mechanism on these APIs. Figures 8.4.a.1 and 8.4.a.2 show parts of the OpenAPI definitions of BATTUTA and RESTCOUNTRIES APIs, respectively. As explained in the previous section, the first step of the process generates an OpenAPI model describing the input definition, while the second step generates UML model where the data aspects have been refined and highlighted.

Figure 8.4.b.1 and 8.4.b.2 show the generated UML models for BATTUTA and RESTCOUNTRIES APIs, respectively. As can be seen, the data model for the BATTUTA API includes the classes *Country*, *Region* and *City*, while the model for the RESTCOUNTRIES API includes the classes *Country*, *RegionalBlock*, and *Currency*. Figure 8.4.c shows the binding model including a *ClassBinding* element for the *Country* entities of both data models, identified as a valid matching concept.

[3]We created a set of heuristics which map operations to entity elements.

[4]`https://battuta.medunes.net/`

[5]`https://restcountries.eu/`

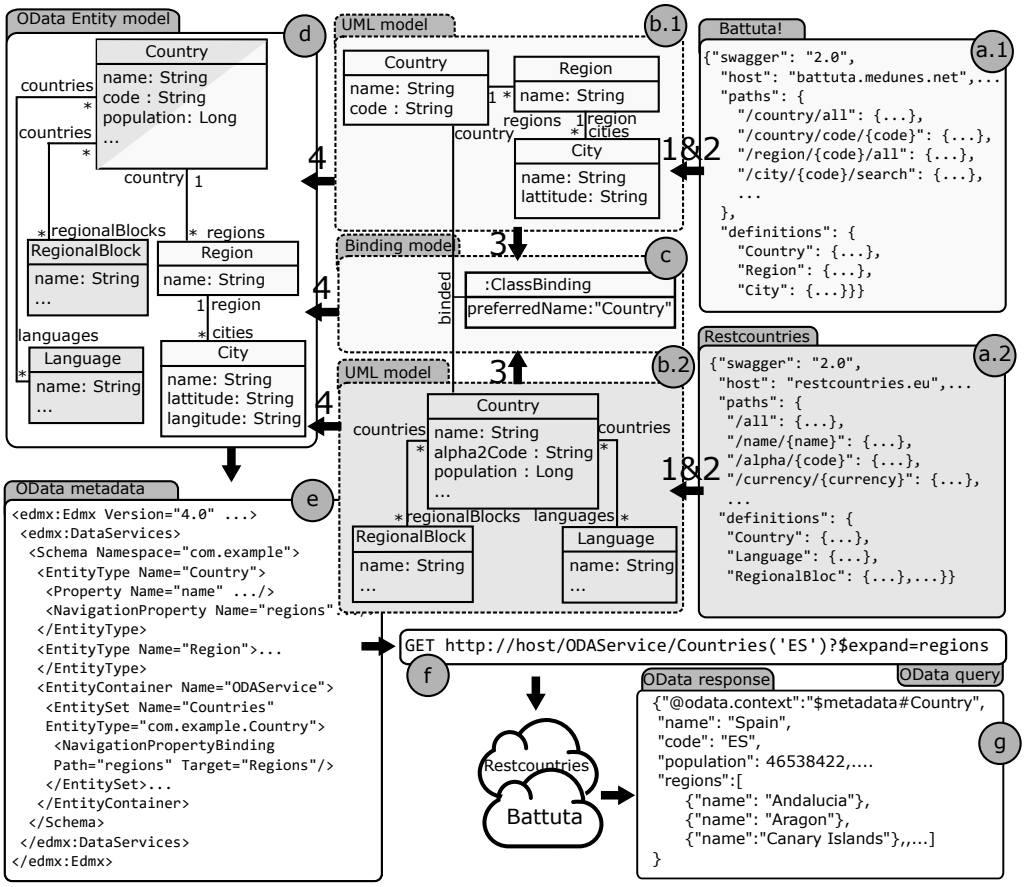


Figure 8.4: APICOMPOSER illustrative example.

Figure 8.4.d shows the OData Entity model created by joining the elements of both data models and resolving the match between the *Country* entities. As can be seen, the *Country* class is shared between both APIs and includes properties and relationships coming from both APIs. Figure 8.4.e shows an excerpt of the Metadata document of the OData Entity model. This document can be retrieved by appending `$metadata` to the URL of the OData application and allows end-users to understand how to query the data.

Figure 8.4.f shows an example of an OData request to retrieve the details of Spain and its regions using the query option `$expand`[6]. This request relies on the concept binding for `Country`, which allows process the request using RESTCOUNTRIES API (mainly for information about the country) and BATTUTA API (for information about the regions). Thus, the

[6] `$expand` specifies that the related resources have to be included in line with retrieved one.

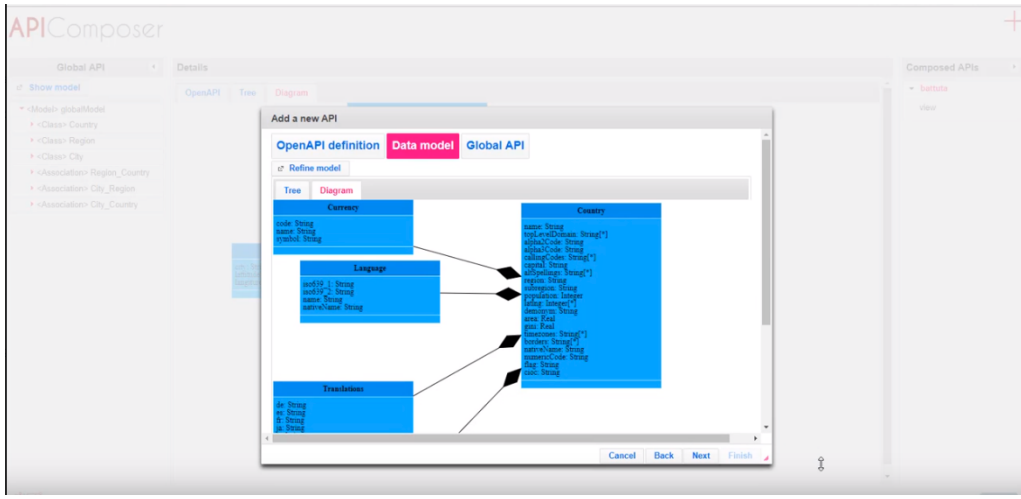


Figure 8.5: Screenshot of APICOMPOSER: *API importer* wizard.

request is traced back to both RESTCOUNTRIES and BATTUTA APIs (i.e., the operations `/alpha/{code}` and `/region/{code}/all`, respectively), which are therefore queried. Figure 8.4.g shows the response in OData format.

8.5 TOOL SUPPORT

We created a proof-of-concept tool implementing our approach which we made available as an Open Source application[7]. Our tool has been implemented as a Java web application which can be deployed in any Servlet container (e.g., APACHE TOMCAT[8]). The application relies on JSF, a server-side technology for developing Web applications; and PRIMEFACES[9], a UI framework for JSF applications; to implement a wizard guiding the user through the steps of the API importer and displaying the different models. The OpenAPI metamodel, the extended OpenAPI metamodel, and the binding metamodel have been implemented using the EMF. OData implementation relies on APACHE OLINGO[10] to provide support for OData entity model, OData query language, and serialization. Figure 8.5 shows a screenshort from APICOMPOSER highlighting the *API importer* wizard.

[7] `https://github.com/SOM-Research/api-composer`
[8] `http://tomcat.apache.org/`
[9] `http://www.primefaces.org`
[10] `http://olingo.apache.org/`

8.6 Related Work

Most of the previous works on REST APIs composition are tight to specific API description languages [Gar+16]. For instance, some of them relied on WADL and HTML for RESTful Services (hREST) to describe the behavior of REST APIs, and Web Service Modeling Ontology (WSMO) and Semantic Annotation of Web Resources (SA-REST) to add semantic annotations (e.g., [Pau09; DRZ10; LG10]). However, none of them gained a broad support mainly because those languages were not successfully adopted [Gar+16]. Verborgh et al. [Ver+17] proposes an approach that relies on hypermedia to compose REST APIs. However, in practice, most REST APIs do not provide hyperlinks to navigate through them. We decided to rely on the OpenAPI specification, which can be seen as a reference solution for REST APIs. The emergence of OpenAPI definitions has motivated initiatives to annotate OpenAPI definitions with semantic descriptions [CD17; Mus+16] and identify APIs for selection [BGD17]. Our approach differs from these works by putting OpenAPI specification at the core of the composition strategy, but we can profit in the future from these works (e.g., by considering semantic descriptions for concept matching).

Our approach focuses on the composition of data-oriented APIs, which allows us to rely on the family of approaches proposed for JSON data [CC14] and in the database world for schema matching and merging [RB01; Bor+07; BLN86]. To the best of our knowledge, only the work by Serrano et al. [Ser+17] proposes a similar approach to ours but theirs require annotating REST APIs with Linked-Data ontologies and uses SPARQL to query to composed APIs.

8.7 Summary

This chapter presented a model-based approach to automatically compose and orchestrate data-driven REST APIs. We relied on the OpenAPI and OData specifications to describe the resources of REST APIs and provide query support, respectively. Our approach parses OpenAPI definitions to extract data models, expressed as UML models, which are combined following a pragmatic matching strategy to create a global data model representing the union of all the data for the input APIs. The global model is exposed as an OData application, thus allowing users to easily perform queries using the OData query language. Queries on the global model are automatically translated into queries on the underlying individual APIs.

CHAPTER

9 Conclusions and Future Work

This chapter draws the conclusions of this thesis in Section 9.1 and presents new ideas for future work in Section 9.2.

9.1 CONCLUSIONS

This thesis presented a model-driven approach to facilitate the design, implementation, composition, and consumption of REST APIs. Our approach mainly targeted (i) the OpenAPI specification which has become the preferred format to define REST APIs, and (ii) OData which is focused on data-centric REST APIs and has gained momentum because of the emergence of Open Data initiatives.

We presented a set of model-based representations and modeling tools for OpenAPI and OData. For both, we provided a metamodel and a UML profile, thus giving users the flexibility to choose the representation that suits them best. By targeting the Eclipse platform, users can rely on a plethora of model-based tools to perform tasks such as (i) model-to-model transformations (e.g., ATL[1], ECLIPSE QVT OPERATIONAL[2]), (ii) code generation (e.g., ACCELEO[3], EGL[4]), (iii) model validation (e.g., ECLIPSE OCL[5], EPSILON VALIDATION

[1] https://www.eclipse.org/atl/
[2] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml
[3] https://www.eclipse.org/acceleo/
[4] https://www.eclipse.org/epsilon/doc/egl/
[5] https://projects.eclipse.org/projects/modeling.mdt.ocl

LANGUAGE (EVL)[6]), (iv) model weaving (e.g., EMF VIEWS[7]), and (v) model comparison (e.g., EMF COMPARE[8]). These resources set the foundations for the rest of the contributions of this thesis.

We presented EMF-REST, our solution to enable model management via REST APIs, thus unlocking modeling tasks which currently rely on heavy desktop environments. EMF-REST paves the way to develop model-based solutions relying on the Cloud as well as an enhanced collaborative support for Web-based modeling tools.

We presented APIDISCOVERER, our example-driven approach to discover Web API specifications, thus helping developers increase the exposure of their APIs without fully writing API specifications and benefit from OpenAPI tooling infrastructure (e.g., generating documentation, generating SDKs). These specifications can be stored in a public repository where developers can use them. We believe our process and repository is a significant step forward towards API reuse, helping developers to find and integrate the APIs they need to provide their software services.

We presented APITESTER, our approach to automate specification-based REST API testing by relying on OpenAPI. APITESTER covers both nominal test cases (i.e., with correct data input) and faulty test cases (i.e., with incorrect data input) to test APIs in correct scenarios and also in the presence of invalid data inputs. We validated our approach using the APIs listed in APIS.GURU[9]. Our experiments showed good coverage levels and revealed that errors are generally found errors in the specification of the APIs and their implementations.

We presented APIGENERATOR, our model-driven approach to generate OData REST APIs from conceptual data models. APIGENERATOR aims at bringing more agility to the Web development process by providing ready-to-run Web APIs out of data models. Also, our approach advances towards the definition of an MDE infrastructure for the generation of OData services, where developers can rely on a plethora of modeling tools to easily design, generate and evolve their web applications.

Finally, we presented APICOMPOSER which proposes a lightweight model-driven approach to compose and orchestrate data-centric REST APIs. APICOMPOSER relies on OpenAPI to describe the input APIs and OData to expose the global API combining all these APIs. The global model is exposed as an OData application, thus allowing users to easily perform queries using

[6] https://www.eclipse.org/epsilon/doc/evl/

[7] https://www.atlanmod.org/emfviews/

[8] https://www.eclipse.org/emf/compare/

[9] https://apis.guru/openapi-directory/

the OData query language. Queries on the global model are automatically translated into queries on the underlying individual APIs. In case users are not familiar with OData, OpenAPI definitions could also be easily derived from OData services using tools such as ODATA-OPENAPI[10]. Our approach advances towards the automatic composition of REST APIs.

Collectively, the presented contributions constitute an ecosystem of solutions which automate different tasks related to REST APIs development and consumption. We believe such contributions to be of a great value to Web APIs developers who want to bring more agility to their development tasks. Furthermore, they advance the state of the art of automated software engineering for REST APIs development and consumption.

9.2 FUTURE WORK

This section presents some ideas for future work. We will propose first some possible extensions of the contributions of this thesis. Later, we will present new ideas for possible research lines which could extend the work presented in this thesis.

9.2.1 *Current Contributions*

MODELING APIS. The resources presented in Chapter 3 could be extended to cover more than functionalities of REST APIs. For instance, we are interested in extending the OpenAPI metamodel and profile to add ontology and vocabulary concepts to describe the APIs on both syntactical level and semantic level. This will help identifying what the APIs offer and therefore prepare the ground for semantic-based composition of REST APIs. We aim to complete our resources with Quality of Service (QoS) and business plan aspects, which play a fundamental role in the API economy. This information will help in the selection process of REST APIs specially when many APIs offer similar functionalities.

We also envision to enrich our metamodel with OCL expressions with the goal of calculating metrics that could be used to ensure high-quality models. Such metrics could help in the recommendation of REST APIs based on their design quality (e.g., `GET` operations should not have `body` parameters) and the quality of their definitions (e.g., does the

[10] `https://github.com/oasis-tcs/odata-openapi`

definition include the data structures consumed and produced by the API?).

Additionally, we would like to investigate new ways to represent OpenAPI and OData models such as a graphical notation in order to visualize the functionalities of REST APIs and therefore facilitate understanding them. This may involve creating a graphical editor using model-based frameworks such as SIRIUS[11] or GRAPHITI[12] to create and visualize the diagrams.

We also plan to support the newly released version of OpenAPI (i.e., OpenAPI 3) as it is getting more adoption, and help migrating to this version. To do so, we will need to create a new metamodel for OpenAPI 3 and a transformation to derive OpenAPI 3 models from OpenAPI 2 models.

Regarding OData, we aim at extending our OData metamodel and OData profile to capture additional OData behavioral concepts such as functions and actions to enable the design and generation of more complex aspects.

EMF-REST. We plan to extend our EMF-REST approach in order to include a small configuration DSL to help designers parameterize the structure of the generated API (e.g., configuring the URIs to the resources, customizing the data structure consumed and produced by the API, configuring the part of the metamodel to be exposed). This will enable the decoupling between the generated REST API and the back-end model, thus allowing both to evolve separately.

We are also interested in exploring the benefits of using EMF-REST in combination with client-side modeling environments, for instance, in ECLIPSE, thus enabling developers to collaborate and deal with large EMF models in a transparent way (i.e., models in Eclipse that are remotely stored using an EMF-REST back-end). To this end, we would like to support a different persistence strategy that promotes scalability and allows the definition of very large models (we currently support the default persistence strategy provided by EMF using XMI files). For instance, we aim to investigate the use of alternative EMF persistence frameworks such as NEOEMF [Dan+17] and CDO[13] to store EMF models in graph databases. This integration may allow

[11] https://www.eclipse.org/sirius/

[12] https://www.eclipse.org/graphiti/

[13] https://www.eclipse.org/cdo/

EMF-REST generated APIs to use less memory (i.e., no need to load the entire model in memory), gain more speed (i.e., saving model into the desk may be costly) and deal with concurrency issues more efficiently.

We would also like to explore a possible extension of the EMF-REST approach that does not require the generation of the EMF Java API. To do so, we need to generate a layer that serves as a proxy for REST calls and delegate the execution of such calls to the EMF API, then retrieve the results and send them to the client as REST responses. This will enable the support of existing metamodels that already have their customized Java API which cannot be regenerated (e.g., UML2[14]).

APIDISCOVERER. We are interested in extending APIDISCOVERER in order to discover other aspects such as non-functional properties, semantic definitions, and the security mechanisms of the APIs under scrutiny, and to support non-JSON data (e.g., XML). The non-functional properties and semantic definitions will rely on the extension of the OpenAPI metamodel presented before. The discovery of non-functional properties will require performing non-functional tests such as load tests, while the discovery of semantic annotations will require studying the state of the art of ontology matching algorithms [ORG15] and the support for automatic semantic annotations of Web services [TM15]. Furthermore, the discovery of security mechanisms will require performing security tests to determine the parameters that grant the authorizations to the API (i.e., API keys, when such parameters are missing the API returns a `403 Forbidden` or `401 Unauthorized` response).

The discovery process *per se* could also be improved by extending our approach to support the generation of call examples based on the textual analysis of the API documentation websites, thus speeding up the process of interacting with the API to infer its specification. We plan to systematically apply our process to a large number of APIs (linked from other directories or repositories) in order to contribute to current repositories of APIs and therefore help developers facilitate the integration of more APIs.

Finally, we want to support also OpenAPI 3 and give the user the choice to choose between version 2 and 3 in the discovery process. The support for OpenAPI 3 will require a metamodel that represents

[14] `https://www.eclipse.org/modeling/mdt/?project=uml2`

the OpenAPI 3 specification. OpenAPI 3 includes some new features such as links between operations and an enhanced support for JSON schema (e.g., support of `oneOf`) which will present new challenges for APIDISCOVERER which, for instance, will have to match operation responses with operation parameters to find links between them.

APITESTER. We are interested in extending APITESTER with the goal to increase the coverage levels by improving our parameter inference technique. For instance, we can rely on natural language processing techniques to infer parameter values and constraints from the parameter descriptions. We could also rely on the OpenAPI metamodel extension proposed before to support semantic annotations and the APIDISCOVERER extension to discover semantic definitions to infer parameter values. The usage of search-based techniques to generate test cases could also be useful here.

We also plan to extend our approach in order to support OpenAPI 3 which will allow us to deal with dependencies between operations and side-effects caused by executing test cases. Thus, this would involve extending our TestSuite metamodel in order support (i) links between test steps which will require adding a property transfer mechanism, and (ii) *pre-* and *post-* requests. The former will allow us to support integrity test cases which evaluate the life-cycle of a particular resource in a REST API, while the latter will allow us to create *pre-* and *post-* operations which prepare data for a test case and manage side-effects, respectively.

We would like also to extend APITESTER to generate other kinds of test cases such as load tests and security test cases. This will require extending the TestSuite metamodel to support these test cases and creating new rules to generate such tests by relying on the proposed extension of the OpenAPI metamodel to support non-functional properties.

APIGENERATOR. We plan to extend our APIGENERATOR approach to support a number of features commonly used in any Web infrastructure such as authentication and encryption. To do so, we envision to develop a DSL that helps developers tune their API definition (e.g., authentication mechanisms, encryption). We need to combine this DSL with the OData metamodel to generate a REST API that takes into consideration these constraints and relies on well known frameworks to support them (e.g., SPRING SECURITY for authentication).

We are interested in extending our code generation facility to support more back-end technologies (e.g., .NET, NODE.JS) and data sources (e.g., NOSQL databases). We also plan to extend our APIGENERATOR approach to comply with the advanced OData conformance level, which implies adding support to other OData functionalities such as canonical functions. This will imply relying on the extension of the OData metamodel presented before.

We plan to add support for OCL to enrich the definition of the OData metamodel in order to define the logic behind the functions. We would like to extend our UML-to-OData transformation in order to support enriched UML models which include operations and OCL constraints to generate fine-grained OData models.

Finally, we envision to integrate our OData models with other web-based modeling languages like IFML that focus on the modeling of the user interaction with the web application. With this integration we aim to provide a rich modeling environment combining both front-end and back-end development.

APICOMPOSER. We are interested in extending the APICOMPOSER approach to consider semantic descriptions for improving the matching strategy and non-functional aspects (like QoS or price) in the generation of the global model when alternative APIs have a high degree of overlapping. To do so, we need to rely on the extensions of the OpenAPI metamodel and the extension of the APIDISCOVERER approach to support them presented before. We would also like to work on a DSL to configure the user's preferences for choosing resolution paths based on non-functional constraints (e.g., by using free APIs when possible).

We would like to extend our approach in order to support not only data retrieval but also data modification (i.e., support all CRUD operations). Such task is not trivial since it may require replicating changes in many APIs. We are also interested in improving the maintainability of our approach by allowing the update of the composed APIs as they evolve.

Finally, we envision to integrate an analysis framework to evaluate the composed API by analyzing the logs related to requests. User interaction with the system could be used to evaluate different parts of the composed API which will contribute to improve the system. For this purpose, several usage metrics (e.g., highly demanded parts of the global API, the average time of a request, or failure rate) and

improvement actions (e.g., substitute an API, fix an API) have to be defined.

ALL-IN-ONE. We plan to create a variety of industrial case studies to validate the contributions of this thesis. The goal of such studies is to demonstrate how our contributions could be integrated together in one scenario with proper empirical measures to sustain the underlying goal of this thesis.

9.2.2 *New Research Lines*

Other than the future work related to the different contributions of this thesis, we identified new research lines related to the topic of this thesis.

FLEXIBLE FRONT-END QUERIES. REST APIs typically offer a set of operations over HTTP protocol. However, most of these APIs do not respect all the required REST constraints [Rod+16] which makes using them confusing. OData[15] aims at solving this problem by defining a set of best practices for building and consuming "true" REST APIs. We presented an approach to model and generate REST APIs in OData style. This situation motivated also the creation of an alternative to REST called GraphQL[16]. GRAPHQL has been created by FACEBOOK in 2012 and released as Open Source in 2015. GRAPHQL is a query language, specification, and set of tools designed for building optimized and flexible Web APIs. It is designed to operate over a single endpoint via HTTP using a query language and proposes its own conventions.

We would like to extend the work presented in this thesis in order to support GRAPHQL as well. This extension will allow our work to enrich the range of supported Web APIs approaches and therefore reach a bigger impact on the field of Web APIs engineering. To integrate GRAPHQL into our approach we will need to create first a metamodel to represent GRAPHQL schemas. This metamodel will be then used by our model-driven approach to design, test and generate GRAPHQL APIs. In the following we describe the different extensions to our contributions to support GRAPHQL.

[15] `http://www.odata.org/`
[16] `https://graphql.org/`

APIGENERATOR. We would like to extend our APIGENERATOR approach to support the generation of GRAPHQL APIs from conceptual models. To achieve this goal, we will need to (i) create a transformation that derive GRAPHQL models from UML models, (ii) create a DSL to parametrize the generation process (e.g., authentication mechanism, targeted data base), and (iii) use the GRAPHQL model and the DSL to generate a ready-to-deploy GRAPHQL API and a database DDL targeting the chosen database.

APITESTER. We would like to extend our APITESTER approach to support the generation of test cases for GRAPHQL APIs. To do so, we will need to (i) derive a GRAPHQL model from a GRAPHQL schema, (ii) adapt our parameter values inference heuristics to GRAPHQL, (iii) extend our TestSuite metamodel, and (iv) create a set of rules to generate test cases based on the GRAPHQL query language with the goal to reach a high coverage level of the schema parts and the GRAPHQL query options (e.g, fields, arguments). Such task in not trivial since a GRAPHQL API shapes its response based on the parameters of the query (e.g., filter fields).

APICOMPOSER. We plan to explore a possible extension of our APICOMPOSER approach to support GRAPHQL as both a source API and query language for the global API. To do so, we will need to (i) derive UML models from GRAPHQL schemas, (ii) expose the global model as a GRAPHQL schema, and (ii) create the logic to transform GRAPHQL queries into requests to the source APIs then compose the responses and send them in GRAPHQL format. This will allow our approach to create a flexible hybrid API combining different data sources and supporting different query languages.

NON-FUNCTIONAL REQUIREMENTS. Non-functional requirements define the requirements that are not directly related to the functionality of a service. However, in the literature there is still a debate about what properties should be considered functional or non-functional. Chung et al. [CS09] addressed the question of non-functional requirements in Software engineering. The qualitative perspective, which is understood as a set of concerns related to the concept of quality (e.g., reliability,

interoperability), constitutes the heart of non-functional requirements. While designing or composing Web APIs, a particular attention should be paid to non-functional concerns, especially in vulnerable applications (e.g., flights booking, banks) where quality of service is required.

As an extension of our contributions, we would like to take into consideration non-functional requirements in our processes. This will have a deep impact on the artifacts developed in this thesis. We already highlighted some possible extensions to our contributions that go into this direction.

As mentioned before, we are interested in extending the OpenAPI metamodel and profile to add non-functional aspects (e.g., QoS, business plans). This will require an investigation to determine the most relevant non-functional requirements for REST APIs and a study to identify the different business plans adopted by API providers (e.g., freemium, credits, rate-limiting). Then we will need to model these concepts and integrate them into our metamodels. In the following we describe how plan to extend our contributions to support non-functional requirements.

APIDISCOVERER. We would like to extend our APIDISCOVERER approach to monitor APIs and calculate these non-functional aspects. We would also like to use web scraping and natural language processing techniques to extract business plans from Web API repositories (e.g., PROGRAMMABLEWEB[17]). The results of the latter will allow us to perform a statical analysis to determine the trends in Web API economy.

APIDISCOVERER. We are also interested in extending our APITESTER approach to generate non-functional test cases (e.g., load tests, speed tests) which check that non-functional aspects are respected (e.g., average response time). Such test cases are very important to ensure the quality of Web APIs specially for those generating a huge traffic and those having a Service Level Agreement (SLA) with their customers.

APICOMOPSER. We would like to extend our APICOMPOSER approach to take into consideration the non-functional aspects when composing REST APIs. This is specially useful to choose the most

[17] https://www.programmableweb.com/

suitable API for a query in the case of overlapping between APIs (e.g., free APIs, fastest APIs).

SMART CITIES. The Smart City concept defines an urban area which relies on collected data to efficiently manage resources in order to enhance the life quality of citizen. This data include data collected from citizens, devices, and assets which are processed in order to serve different application domains, namely: natural resources and energy, transport and mobility, buildings, living, government, and economy and people [Nei+14]. Smart Cities integrate Internet of things (IoT), which defines a network of devices connected and exchanging data, to achieve its goals. Smart Cities rely also on data exposed by Web APIs (e.g., Open Data APIs) to optimize the services they provide.

We see smart cities as a potential direction to apply the work presented in this thesis. Since Web APIs play an important role in the Smart Cities movement, we plan to extend our APIGENERATOR approach in order to generate REST APIs from already existing data sources (e.g., databases, XML, JSON) to facilitate the use of these data in smart cities tasks. To do so, we need to infer the data model from the data source and generate the REST API accordingly. We would also like to target the development of applications that consume IoT data. In fact, IoT technologies are evolving but mainly focus on the technological and infrastructure aspects to create such systems. Front-end development of IoT, on the other hand, still needs attention and research on this direction will contribute in the adoption of IoT solutions [BUA17]. Thus, as we did with REST APIs, we would like to define model-driven approaches to facilitate the design of applications that consume IoT applications. We are interested in extending the work of Brambilla and al. [BUA17], which proposes a model-driven approach to design user interfaces of IoT systems by extending IFML language to support IoT, to target also the generation of Web APIs that consume IoT data. We would also like to extend our APICOMPOSER approach in order to apply it to smart cities use cases by (i) integrating also IoT data to optimize city operations (e.g., searching for apartments taking into consideration not only the price and but also pollution and noise levels), and (ii) generating mobile applications for such use cases. By doing so, we aim to offer normal citizens the tools to contribute to the Smart City movement.

Acronyms

API	Application Programming Interface.
ATL	ATL Transformation Language.
CASE	Computer Aided Software Engineering.
CDI	Context Dependency Injection.
CORS	Cross-Origin Resource Sharing.
CSDL	Common Schema Definition Language.
DDL	Data Definition Language.
DSL	Domain Specific Language.
EDM	Entity Data Model.
EGL	Epsilon Generation Language.
EJB	Enterprise Java Bean.
EMF	Eclipse Modeling Framework.
ER	Entity Relationship.
EVL	Epsilon Validation Language.
GMF	Graphical Modeling Framework.
HATEOAS	Hypermedia As The Engine Of Application State.
hREST	HTML for RESTful Services.

HTTP	Hypertext Transfer Protocol.
HTTPS	Hypertext Transfer Protocol Secure.
IFML	Interaction Flow Modeling Language.
IoT	Internet of things.
JAX-RS	Java API for Representational State Transfer.
JAXB	Java Architecture for XML Binding.
JPA	Java Persistence API.
JSF	JavaServer Faces.
JSON	JavaScript Object Notation.
MDA	Model-Driven Architecture.
MDE	Model-Driven Engineering.
MDT	Model Development Tools.
MDWE	Model Driven Web Engineering.
MIME	Multipurpose Internet Mail Extensions.
MOF	Meta-Object Facility.
MTL	Model-To-Text Transformation Language.
OAI	OpenAPI Initiative.
OCL	Object Constraint Language.
OData	Open Data Protocol.
OMG	Object Management Group.
PIM	Platform-Independent Model.
POM	Project Object Model.
PSM	Platform-Specific Model.
QoS	Quality of Service.
QVT	Query/View/Transformation.
RDF	Resource Description Framework.
REST	Representational State Transfer.
RFC	Request for Comments.
SA-REST	Semantic Annotation of Web Resources.
SDK	Software Development Kit.
SLA	Service Level Agreement.
SOAP	Simple Object Access Protocol.

SQL	Structured Query Language.
SSL	Secure Sockets Layer.
TLS	Transport Layer Security.
UML	Unified Modeling Language.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
WADL	Web Application Description Language.
WSDL	Web Services Description Language.
WSMO	Web Service Modeling Ontology.
XMI	XML Metadata Interchange.
XML	Extensible Markup Language.
YAML	YAML Ain't Markup Language.

Bibliography

[AC13] Camilo Alvarez and Rubby Casallas. “MTC Flow: A Tool to Design, Develop and Deploy Model Transformation Chains”. In: *Workshop on ACadeMics Tooling with Eclipse*. 2013, pp. 1–9.

[Arc17] Andrea Arcuri. “RESTful API Automated Test Case Generation”. In: *International Conference on Software Quality, Reliability and Security*. 2017, pp. 9–20.

[Aué+18] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. “An Exploratory Study on Faults in Web API Integration in a Large-Scale Payment Company”. In: *International Conference on Software Engineering: Software Engineering in Practice*. 2018, pp. 13–22.

[Bai+05] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. “WSDL-based Automatic Test Case Generation for Web Services Testing”. In: *International Workshop on Service-Oriented System Engineering*. 2005, pp. 207–212.

[Bar+09] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. “WS-TAXI: A WSDL-based Testing Tool for Web Services”. In: *International Conference on Software Testing Verification and Validation*. 2009, pp. 326–335.

[Ben+14] Clara Benac Earle, Lars-Åke Fredlund, Ángel Herranz, and Julio Mariño. “Jsongen: A QuickCheck Based Library for Test-

ing JSON Web Services". In: *Workshop on Erlang*. 2014, pp. 33–41.

[BF+14] Marco Brambilla, Piero Fraternali, et al. *Interaction Flow Modeling Language*. Tech. rep. Object Management Group (OMG), 2014.

[BFM05] Tim Berners-Lee, Roy Fielding, and Larry Masinter. *Uniform Resource Identifier (URI): Generic Syntax, RFC 3986*. Tech. rep. 2005.

[BGD17] Luciano Baresi, Martin Garriga, and Alan De Renzis. "Microservices Identification Through Interface Analysis". In: *European Conference on Service-Oriented and Cloud Computing*. 2017, pp. 19–33.

[BHH13] Mustafa Bozkurt, Mark Harman, and Youssef Hassoun. "Testing and Verification in Service-Oriented Architecture: A Survey". In: *Software Testing, Verification and Reliability* 23.4 (2013), pp. 261–313.

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant B. Navathe. "A Comparative Analysis of Methodologies for Database Schema Integration". In: *ACM Computing Surveys (CSUR)* 18.4 (1986), pp. 323–364.

[Bor+07] Artur Boronat, José Á Carsí, Isidro Ramos, and Patricio Letelier. "Formal Model Merging Applied to Class Diagram Integration". In: *Electronic Notes in Theoretical Computer Science* 166 (2007), pp. 5–26.

[BUA17] Marco Brambilla, Eric Umuhoza, and Roberto Acerbis. "Model-Driven Development of User Interfaces for IoT Systems via Domain-Specific Components and Patterns". In: *J. Internet Services and Applications* 8.1 (2017), 14:1–14:21.

[BW14] Manfred Bortenschlanger and Steven Willmott. *The API Owner's Manual*. Tech. rep. 3Scale, 2014.

[CC14] Javier Luis Cánovas Izquierdo and Jordi Cabot. "Composing JSON-Based Web APIs". In: *International Conference on Web Engineering*. 2014, pp. 390–399.

[CC16] Javier Luis Cánovas Izquierdo and Jordi Cabot. "JSONDiscoverer: Visualizing the Schema Lurking Behind JSON Documents". In: *Knowledge-Based Systems* 103 (2016), pp. 52–55.

[CD17] Marco Cremaschi and Flavio De Paoli. “Toward Automatic Semantic API Descriptions to Support Services Composition”. In: *European Conference on Service-Oriented and Cloud Computing*. 2017, pp. 159–167.

[CFB00] Stefano Ceri, Piero Fraternali, and Aldo Bongio. “Web Modeling Language (WebML): A Modeling Language for Designing Web Sites”. In: *Computer Networks* 33 (2000), pp. 137–157.

[CFB17] Hanyang Cao, Jean-Rémy Falleri, and Xavier Blanc. “Automated Generation of REST API Specification from Plain HTML Documentation”. In: *Internationl Conference on Service-Oriented Computing*. Springer. 2017, pp. 453–461.

[CG12] Jordi Cabot and Martin Gogolla. “Object Constraint Language (OCL): A Definitive Guide”. In: *Formal Methods for Model-Driven Engineering*. Springer, 2012, pp. 58–90.

[CK09] Sujit Kumar Chakrabarti and Prashant Kumar. “Test-the-REST: An Approach to Testing RESTful Web-Services”. In: *International Conference on Advanced Service Computing*. 2009, pp. 302–308.

[CS09] Lawrence Chung and Julio Cesar Sampaio do Prado Leite. “On Non-Functional Requirements in Software Engineering”. In: *Conceptual modeling: Foundations and applications*. Vol. 5600. 2009, pp. 363–379.

[Dan+17] Gwendal Daniel, Gerson Sunyé, Amine Benelallam, Massimo Tisi, Yoann Vernageau, Abel Gómez, and Jordi Cabot. “NeoEMF: A Multi-Database Model Persistence Framework for Very Large Models”. In: *Science of Computer Programming* 149 (2017), pp. 9–14.

[DRZ10] Teodoro De Giorgio, Gianluca Ripa, and Maurilio Zuccalà. “An approach to enable replacement of SOAP services and REST services in lightweight processes”. In: *International Conference on Web Engineering*. 2010, pp. 338–346.

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2010.

[EZG14] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. “Web API Growing Pains: Stories from Client Developers and their Code”. In: *International Conference on Software Maintenance, Reengineering and Reverse Engineering*. 2014, pp. 84–93.

[EZG15] Tiago Espinha, Andy Zaidman, and Hans-Gerhard Gross. “Web API Growing Pains: Loosely Coupled Yet Strongly Tied”. In: *Journal of Systems and Software* 100 (2015), pp. 27–43.

[FB15] Tobias Fertig and Peter Braun. “Model-driven Testing of RESTful APIs”. In: *International Conference on World Wide Web*. 2015, pp. 1497–1502.

[Fid+12] Robson Do Nascimento Fidalgo, Elvis Maranhão De Souza, Sergio España, Jaelson Brelaz De Castro, and Oscar Pastor. “EERMM: a Metamodel for the Enhanced Entity-Relationship Model”. In: *International Conference on Conceptual Modeling*. 2012, pp. 515–524.

[Fie+99] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. *HTTP/1.1, RFC 2616*. Tech. rep. 1999.

[Fie00] Roy Thomas Fielding. “Architectural Styles and the Design of Network-based Software Architectures”. PhD thesis. 2000.

[Fra+16] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. “Example-Directed Synthesis: A Type-Theoretic Interpretation”. In: *ACM Symposium on Principles of Programming Languages*. 2016, pp. 802–815.

[Fra99] Piero Fraternali. “Tools and Approaches for Developing Data-intensive Web Applications: A Survey”. In: *ACM Computing Surveys* 31.3 (1999), pp. 227–263.

[Gar+16] Martin Garriga, Cristian Mateos, Andres Flores, Alejandra Cechich, and Alejandro Zunino. “RESTful Service Composition at a Alance: A Survey”. In: *Journal of Network and Computer Applications* 60 (2016), pp. 32–53.

[Gra96] Matthew Gray. *Growth and Usage of the Web and the Internet*. Tech. rep. 1996.

[Had06] Marc J. Hadley. *Web Application Description Language (WADL)*. Tech. rep. 2006.

[Hau+14] F. Haupt, D. Karastoyanova, F. Leymann, and B. Schroth. “A Model-Driven Approach for REST Compliant Services”. In: *International Conference on Web Services*. 2014, pp. 129–136.

[HM08] Samer Hanna and Malcolm Munro. “Fault-Based Web Services Testing”. In: *International Conference on Information Technology: New Generations*. 2008, pp. 471–476.

[HPB14] Ralf Handl, Michael Pizzo, and Mark Biamonte. *OData JSON Format Version 4.01*. Tech. rep. OASIS, 2014.

[HRW11] John Hutchinson, Mark Rouncefield, and Jon Whittle. "Model-Driven Engineering Practices in Industry". In: *International Conference on Software Engineering*. 2011, pp. 633–642.

[Hut+11] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. "Empirical Assessment of MDE in Industry". In: *International Conference on Software Engineering*. 2011, pp. 471–480.

[Jou+08] Frederic Jouault, Freddy Allilaire, Jean Bezivin, and Ivan Kurtev. "ATL: A Model Transformation Tool". In: *Science of Computer Programming* 72.1-2 (2008), pp. 31–39.

[KK12] Nora Koch and Sergej Kozuruba. "Requirements Models as First Class Entities in Model-Driven Web Engineering". In: *International Conference on Web Engineering, Workshops*. 2012, pp. 158–169.

[Kle+15] Meike Klettke, Uta Störl, Stefanie Scherzinger, and OTH Regensburg. "Schema Extraction and Structural Outlier Detection for JSON-based NoSQL Data Stores". In: *Conference on Database Systems for Business, Technology, and Web*. 2015, pp. 425–444.

[LG10] Markus Lanthaler and Christian Gütl. "Towards a RESTful service ecosystem". In: *International Conference on Digital Ecosystems and Technologies*. 2010, pp. 209–214.

[Lóp+15] Jesús J. López-Fernández, Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "Example-Driven Meta-Model Development". In: *Software & Systems Modeling* 14.4 (2015), pp. 1323–1347.

[Mas11] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media, Inc., 2011.

[Max+07] E.Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. "A Domain-Specific Language for Web APIs and Services Mashups". In: *International Conference on Service-Oriented Computing*. 2007, pp. 13–26.

[MCG10] Brambilla Marco, Jordi Cabot, and Michael Grossniklaus. “Tools for Modeling and Generating Safe Interface Interactions in Web Applications”. In: *International Conference on Web Engineering*. 2010, pp. 482–485.

[Mor90] Larry J. Morell. “A Theory of Fault-based Testing”. In: *IEEE Transactions on Software Engineering* 16.8 (1990), pp. 844–857.

[Mot+11] Hamid Reza Motahari-Nezhad, Regis Saint-Paul, Fabio Casati, and Boualem Benatallah. “Event Correlation for Process Discovery from Web Service Interaction Logs”. In: *International Journal on Very Large Data Bases* 20.3 (2011), pp. 417–444.

[Mus+16] Fathoni A Musyaffa, Lavdim Halilaj, Ronald Siebes, Fabrizio Orlandi, and Sören Auer. “Minimally Invasive Semantification of Light Weight Service Descriptions”. In: *International Conference on Web Services*. 2016, pp. 672–677.

[Nei+14] Paolo Neirotti, Alberto De Marco, Anna Corinna Cagliano, Giulio Mangano, and Francesco Scorrano. “Current Trends in Smart City Initiatives: Some Stylised Facts”. In: *Cities* 38 (2014), pp. 25–36.

[Nie+07] O. Nierstrasz, M. Kobel, T. Girba, and M. Lanza. “Example-Driven Reconstruction of Software Models”. In: *European Conference on Software Maintenance and Reengineering*. 2007, pp. 275–286.

[ODD17] Nathalie Oostvogels, Joeri De Koster, and Wolfgang De Meuter. “Inter-Parameter Constraints in Contemporary Web APIs”. In: *International Conference on Web Engineering*. 2017, pp. 323–335.

[OMG14a] OMG. *MDA Guide rev. 2.0*. Tech. rep. Object Management Group, 2014.

[OMG14b] OMG. *Object Constraint Language*. Tech. rep. Object Management Group, 2014. URL: https://www.omg.org/spec/OCL/2.4/PDF.

[OMG15] OMG. *XML Metadata Interchange (XMI) Specification*. Tech. rep. Object Management Group, 2015. URL: https://www.omg.org/spec/XMI/2.5.1/PDF.

[OMG16a] OMG. *Meta Object Facility (MOF) 2.0 Query / View / Transformation Specification*. Tech. rep. Object Management Group, 2016. URL: https://www.omg.org/spec/QVT/1.3/PDF.

[OMG16b] OMG. *OMG Meta Object Facility (MOF) Core Specification*. Tech. rep. Object Management Group, 2016. URL: https://www.omg.org/spec/MOF/2.5.1/PDF.

[OMG17] OMG. *OMG Unified Modeling Language (OMG UML)*. Tech. rep. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/2.5.1/PDF.

[ORG15] Lorena Otero-Cerdeira, Francisco J Rodríguez-Martínez, and Alma Gómez-Rodríguez. "Ontology Matching: A Literature Review". In: *Expert Systems with Applications* 42.2 (2015), pp. 949–971.

[OX04] Jeff Offutt and Wuzhi Xu. "Generating Test Cases for Web Services Using Data Perturbation". In: *Software Engineering Notes* 29.5 (2004), pp. 1–10.

[Pau09] Cesare Pautasso. "RESTful Web service composition with BPEL for REST". In: *Data & Knowledge Engineering* 68.9 (2009), pp. 851–866.

[PHZ14a] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. *OData Common Schema Definition Language (CSDL) JSON Representation Version 4.01*. Tech. rep. OASIS, 2014.

[PHZ14b] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. *OData Common Schema Definition Language (CSDL) XML Representation Version 4.01*. Tech. rep. OASIS, 2014.

[PHZ14c] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. *OData Version 4.0 Part 1: Protocol*. Tech. rep. OASIS, 2014.

[PHZ14d] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. *OData version 4.0 part 2: URL Conventions*. Tech. rep. OASIS, 2014.

[PHZ14e] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. *OData Version 4.0 Part 3: Common Schema Definition Language (CSDL)*. Tech. rep. OASIS, 2014.

[PR11] Ivan Porres and Irum Rauf. "Modeling Behavioral RESTful Web Service Interfaces in UML". In: *ACM Symposium on Applied Computing*. 2011, pp. 1598–1605.

[PZL08] Cesare Pautasso, Olaf Zimmermann, and Frank Leymann. “RESTful Web Services vs. “Big”’ Web Services”. In: *International Conference on World Wide Web*. 2008, pp. 805–814.

[QBC13] Silvia Quarteroni, Marco Brambilla, and Stefano Ceri. “A Bottom-up, Knowledge-Aware Approach to Integrating and Querying Web Data Services”. In: *ACM Transactions on the Web* 7.4 (2013), pp. 19–33.

[QN10] Xhevi Qafmolla and Viet Cuong Nguyen. “Automation of Web Services Development Using Model Driven Techniques”. In: *International Conference on Computer and Automation Engineering*. Vol. 3. 2010, pp. 190–194.

[RAR13] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O’Reilly Media, Inc., 2013.

[RB01] Erhard Rahm and Philip A Bernstein. “A Survey of Approaches to Automatic Schema Matching”. In: *the VLDB Journal* 10.4 (2001), pp. 334–350.

[Riv+13a] José Matías Rivero, Sebastian Heil, Julián Grigera, Martin Gaedke, and Gustavo Rossi. “MockAPI: An Agile Approach Supporting API-first Web Application Development”. In: *International Conference on Web Engineering*. 2013, pp. 7–21.

[Riv+13b] José Matías Rivero, Sebastian Heil, Julián Grigera, Martin Gaedke, and Gustavo Rossi. “MockAPI: an Agile Approach Supporting API-first Web Application Development”. In: *International Conference on Web Engineering*. 2013, pp. 7–21.

[Riv+14] José Matías Rivero, Sebastian Heil, Julián Grigera, Esteban Robles Luna, and Martin Gaedke. “An Extensible, Model-driven and End-User Centric Approach for API Building”. In: *International Conference on Web Engineering*. Ed. by Sven Casteleyn, Gustavo Rossi, and Marco Winckler. 2014, pp. 494–497.

[RMM15] Diego Sevilla Ruiz, Severino Feliciano Morales, and Jesús García Molina. “Inferring Versioned Schemas from NoSQL Databases and its Applications”. In: *International Conference on Conceptual Modeling*. 2015, pp. 467–480.

[Rod+13] Roberto Rodríguez-Echeverría, Fernando Macías, Víctor M. Pavón, José M. Conejero, and Fernando Sánchez-Figueroa. "Model-Driven Generation of a REST API from a Legacy Web Application". In: *International Conference on Web Engineering, Workshops*. 2013, pp. 133–147.

[Rod+15] Pablo Rodriguez Mier, Carlos Pedrinaci, Manuel Lama, and Manuel Mucientes. "An Integrated Semantic Web Service Discovery and Composition Framework". In: *IEEE Transactions on Services Computing* 9.4 (2015), pp. 537–550.

[Rod+16] Carlos Rodríguez, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali, and Gianraffaele Percannella. "REST APIs: a large-scale analysis of compliance with principles and best practices". In: *International Conference on Web Engineering*. Springer. 2016, pp. 21–39.

[SAM15] SM Sohan, Craig Anslow, and Frank Maurer. "SpyREST: Automated RESTful API Documentation Using an HTTP Proxy Server (N)". In: *International Conference on Automated Software Engineering*. 2015, pp. 271–276.

[Sch+08] Wieland Schwinger, Werner Retschitzegger, Andrea Schauerhuber, Gerti Kappel, Manuel Wimmer, Birgit Pröll, Cristina Cachero Castro, Sven Casteleyn, Olga De Troyer, Piero Fraternali, et al. "A Survey on Web Modeling Approaches for Ubiquitous Web Applications". In: *International Journal of Web Information Systems* 4.3 (2008), pp. 234–305.

[Sch06] Douglas C Schmidt. "Model-Driven Engineering". In: *IEEE Computer Society* 39.2 (2006), p. 25.

[SCL14] Angel Mora Segura, Jesús Sánchez Cuadrado, and Juan de Lara. "ODaaS: Towards the Model-driven Engineering of Open Data applications as Data Services". In: *International Conference on Enterprise Distributed Object Computing, Workshops and Demonstrations*. 2014, pp. 335–339.

[Sel03] Brian Selic. "The Pragmatics of Model-Driven Development". In: *IEEE Software* 20.5 (2003), pp. 19–25.

[Ser+08] Belkacem Serrour, Daniel P. Gasparotto, Hamamache Kheddouci, and Boualem Benatallah. "Message Correlation and Business Protocol Discovery in Service Interaction Logs". In:

International Conference on Advanced Information Systems Engineering. 2008, pp. 405–419.

[Ser+17] Diego Serrano, Eleni Stroulia, Diana Lau, and Tinny Ng. "Linked REST APIs: A Middleware for Semantic REST API Integration". In: *International Conference on Web Services*. 2017, pp. 138–145.

[She+14] Quan Z Sheng, Xiaoqiang Qiao, Athanasios V Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. "Web Services Composition : A Decade's Overview". In: *Information Sciences* 280 (2014), pp. 218–238.

[SP04] Cristina Schmidt and Manish Parashar. "A Peer-to-Peer Approach to Web Service Discovery". In: *International Conference on World Wide Web*. 2004, pp. 211–229.

[SWK06] Andrea Schauerhuber, Manuel Wimmer, and Elisabeth Kapsammer. "Bridging Existing Web Modeling Languages to Model-Driven Engineering: A Metamodel for WebML". In: *International Conference on Web Engineering*. 2006.

[Ter+17] Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. "MicroBuilder: A Model-driven Tool for the Specification of REST Microservice Architectures". In: *International Conference on Information Society and Technology*. 2017, pp. 179–184.

[TM15] Davide Tosi and Sandro Morasca. "Supporting the Semi-Automatic Semantic Annotation of Web Services: A Systematic Literature Review". In: *Information and Software Technology* 61 (2015), pp. 16–32.

[TV13a] Nírondes A. C. Tavares and Samyr Vale. "A Model Driven Approach for the Development of Semantic RESTful Web Services". In: *International Conference on Information Integration and Web-based Applications & Services*. 2013, p. 290.

[TV13b] Nírondes AC Tavares and Samyr Vale. "A Model Driven Approach for the Development of Semantic RESTful Web Services". In: *International Conference on Information Integration and Web-based Applications & Services*. 2013, p. 290.

[Val+07] Antonio Vallecillo, Nora Koch, Cristina Cachero, Sara Comai, Piero Fraternali, Irene Garrigós, Jaime Gómez, Gerti Kappel, Alexander Knapp, Maristella Matera, Santiago Meliá, Nathalie Moreno, Birgit Pröll, Thomas Reiter, Werner Retschitzegger, José Eduardo Rivera, Andrea Schauerhuber, Wieland Schwinger, Manuel Wimmer, and Gefei Zhang. "MDWEnet: A Practical Approach to Achieving Interoperability of Model-Driven Web Engineering Methods". In: *International Conference on Web Engineering, Workshops*. 2007.

[Ver+17] Ruben Verborgh, Dörthe Arndt, Sofie Van Hoecke, Jos De Roo, Giovanni Mels, Thomas Steiner, and Joaquim Gabarro. "The Pragmatic Proof: Hypermedia API Composition and Execution". In: *Theory and Practice of Logic Programming* 17.1 (2017), pp. 1–48.

[VP11] Pedro Valderas and Vicente Pelechano. "A Survey of Requirements Specification in Model-driven Development of Web Applications". In: *ACM Transactions on the Web* 5.2 (2011), p. 10.

[W3C14a] W3C. *Cross-Origin Resource Sharing*. Tech. rep. W3C, 2014. URL: `https://www.w3.org/TR/cors/`.

[W3C14b] W3C. *RDF Schema 1.1*. Tech. rep. W3C, 2014. URL: `https://www.w3.org/TR/cors/`.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.

[XOL05] Wuzhi Xu, Jeff Offutt, and Juan Luo. "Testing Web Services by XML Perturbation". In: *International Symposium on Software Reliability Engineering*. 2005, pp. 257–266.

[Yan+18] Jinqiu Yang, Erik Wittern, Annie T. T. Ying, Julian Dolby, and Lin Tan. "Towards Extracting Web API Specifications from Documentation". In: *International Conference on Mining Software Repositories*. 2018, pp. 454–464.

[Zol+17] Christoforos Zolotas, Themistoklis Diamantopoulos, Kyriakos C Chatzidimitriou, and Andreas L Symeonidis. "From Requirements to Source Code: A Model-Driven Engineering Approach for RESTful Web Services". In: *Automated Software Engineering* 24.4 (2017), pp. 791–838.

[ZPH14] Martin Zurmuehl, Michael Pizzo, and Ralf Handl. *OData Atom Format Version 4.0*. Tech. rep. OASIS, 2014.

[ZZ05] Jia Zhang and L-J Zhang. “Criteria Analysis and Validation of the Reliability of Web Services-Oriented Systems”. In: *International Conference on Web Services*. 2005.