INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

RESTful Web Services Development with a Model-Driven Engineering Approach

RAFAEL CORVEIRA DA CRUZ GONÇALVES

Julho de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

RESTful Web Services Development with a Model-Driven Engineering Approach

1130837, Rafael Gonçalves

Dissertation to obtain the Master Degree in Informatics Engineering, Area of Expertise in Software Engineering

Porto, july 2018

Instituto Superior de Engenharia do Porto

RESTful Web Services Development with a Model-Driven Engineering Approach

1130837, Rafael Gonçalves

Dissertation to obtain the Master Degree in Informatics Engineering, Area of Expertise in Software Engineering

Supervisor: Isabel Azevedo

Porto, july 2018

Abstract

A RESTful web service implementation requires following the constrains inherent to Representational State Transfer (REST) architectural style, which, being a non-trivial task, often leads to solutions that do not fulfill those requirements properly.

Model-driven techniques have been proposed to improve the development of complex applications. In model-driven software development, software is not implemented manually based on informal descriptions, but partial or completely generated from formal models derived from metamodels.

A model driven approach, materialized in a domain specific language that integrates the OpenAPI specification, an emerging standard for describing REST services, allows developers to use a design first approach in the web service development process, focusing in the definition of resources and their relationships, leaving the repetitive code production process to the automation provided by model-driven engineering techniques. This also allows to shift the creative coding process to the resolution of the complex business rules, instead of the tiresome and error-prone create, read, update, and delete operations.

The code generation process covers the web service flow, from the establishment and exposure of the endpoints to the definition of database tables.

Key-words: Web service, Model driven engineering, OpenAPI, Resource, Domain specific language, RESTful

Resumo

A implementação de serviços web RESTful requer que as restrições inerentes ao estilo arquitetónico “Representational State Transfer” (REST) sejam cumpridas, o que, sendo usualmente uma tarefa não trivial, geralmente leva a soluções que não atendem a esses requisitos adequadamente.

Técnicas orientadas a modelos têm sido propostas para melhorar o desenvolvimento de aplicações complexas. No desenvolvimento de software orientado a modelos, o software não é implementado manualmente com base em descrições informais, mas parcial ou completamente gerado a partir de modelos formais derivados de meta-modelos.

Uma abordagem orientada a modelos, materializada através de uma linguagem específica do domínio que integra a especificação OpenAPI, um padrão emergente para descrever serviços REST, permite aos desenvolvedores usar uma primeira abordagem de design no processo de desenvolvimento de serviços da Web, concentrando-se na definição dos recursos e das suas relações, deixando o processo de produção de código repetitivo para a automação fornecida por técnicas de engenharia orientadas a modelos. Isso também permite focar o processo de codificação criativo na resolução e implementação das regras de negócios mais complexas, em vez de nas operações mais repetitivas e propensas a erros: criação, leitura, atualização e remoção de dados.

O processo de geração de código abrange o fluxo do serviço web desde o estabelecimento e exposição dos caminhos para os serviços disponíveis até à definição de tabelas de base de dados.

Key-words: Serviço web, Engenharia orientada a modelos, OpenAPI, Recurso Linguagem específica do domínio, RESTful

Table of contents

List of figures

List of tables

List of code snippets

Notation and Glossary

Term	Description
AHP	Analytic Hierarchy Process
API	Application Programming Interface
CBO	Coupling Between Objects
CR	Consistency Ratio
CRUD	Create, Read, Update, and Delete
DBMS	Database Management System
DIT	Depth of Inheritance Tree
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
EMF	Eclipse Modeling Framework
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JAX-RS	Java API for RESTful Web Services
JDT	Java Development Tools
JSON	JavaScript Object Notation
LCOM	Lack of Cohesion in Methods
MDD	Model-Driven Development
MDE	Model-Driven Engineering
MDS	Mode-Driven Security
MDSD	Model-Driven Software Development
MOF	Meta-Object Facility
NCD	New Concept Development
NOC	Number of Children
NOM	Number of methods
OAI	OpenAPI Initiative

Term	**Description**
OAS	OpenAPI Specification
OWASP	Open Web Application Security Project
PaaS	Platform as a Service
POJO	Plain Old Java Object
QoS	Quality of Service
RAML	RESTful API Modeling Language
REST	REpresentational State Transfer
RFC	Response for Class
SDK	Software Development Kit
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
UI	User-Interface
URI	Uniform Resource Identifier
VNA	Value Network Analysis
WMC	Weighted Methods per Class
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

This chapter is intended for the presentation of the research scope of the work developed during Master Thesis of the Informatics Engineering Course program at Polytechnic of Porto - School of Engineering. First, the background and motivation associated with it are presented, followed by the main goals that this work proposes to achieve. A brief reference is made to the main technologies involved, with a more extensive description in Chapter 2. The main contributions and the development methodology of the work on display are mentioned. A case study is introduced, presenting a high-level summary of the respective main functionalities. Finally, the structure of the report itself is described, trying to establish a contextual order of the accomplished objectives.

1.1 Background and motivation

During the last few years, offering software in the form of web services has gained popularity due to the evolution of cloud architectures and the impact of the Internet of Things (IoT) on the number of connections that can take place.

Ever since its introduction by Fielding (Fielding, 2000), the REpresentational State Transfer (REST) architectural style has been increasingly preferred by developers for its simplicity and scalability (Haupt, Leymann, Scherer, & Vukojevic-Haupt, 2017). REST comprises a set of rules and practices offering simple comprehensible application programming interfaces (APIs), clear representations, and scalable services (Dimitrieski et al., 2017).

In an attempt at defining a set of standards to describe APIs, and to aid developers in the creation of the interfaces, the OpenAPI Initiative (OAI) was formed by a consortium of forward-looking industry experts who recognize the immense value of standardizing on how REST APIs are described. As an open governance structure under the Linux Foundation, the OAI is focused on creating, evolving and promoting a vendor neutral description format (The Linux Foundation, 2017a).

The OpenAPI Specification requires the description of the capabilities of the service without mandating a specific development process, such as design-first or code-first. It does facilitate either technique by establishing clear interactions with a REST API (The Linux Foundation, 2017b).

Meanwhile, from an automated software engineering perspective, Model Driven Engineering (MDE) is gaining popularity (Zolotas, Diamantopoulos, Chatzidimitriou, & Symeonidis, 2017), and it can be useful for the development of REST APIs. A great amount of time is required because developers must address several issues (software architecture, error handling, security, access rules, among others) to ensure a suitable design and the desired usability (Schreibmann & Braun, 2014). To implement this solution according to all the best practices is time consuming and it is an expensive part of every software project.

The specification of the structural and behavioral aspects of an application using model-driven principles can be used for code generation (Schreibmann & Braun, 2014).

With the standardization of REST APIs description, provided by the OAI, an MDE approach for the development of a Domain Specific Language (DSL) to generate full API applications can be a helpful and powerful resource for the developers (Scheidgen, Efftinge, & Marticke, 2016). Although there have been several initiatives devoted to the modeling and formal description of REST applications, most of them do not support or even force RESTful compliance (Ed-Douibi, Izquierdo, Gómez, Tisi, & Cabot, 2015; El-khoury, Gurdur, & Nyberg, 2016).

1.2 Objectives

The domain of REST APIs development conducts to solutions with repetitive or very similar code between the different interfaces. This is the realm where Model-Driven Development can be useful, abstracting this characteristic, using models as the primary artifact of the development process.

The objective of this research and overall work is to come up with a solid, feasible and efficient engineering solution, based on MDE techniques, specifically DSL(s), to take as inputs the structural and the behavioral aspects of the business domain, and then generate the associated RESTful web services, while being compliant with the most recent version of the Open API specification.

Therefore, some questions can be derived from the earlier paragraphs, summarizing the main objectives that this work intends to achieve:

1. What MDE approach can be adopted to ensure a more efficient and reliable process of web services development?
2. What are the compromises to specify a language agnostic metamodel to represent and define the OpenAPI Specification?
3. Can a code generation process be developed over the DSL referred previously, and consequently aiding developers in web services development?
4. How does an OAS-based DSL and code generation process compare with a Resource-based one, with a simpler language?

1.3 Case study

My Pocket Nutritionist (finalist of 2017 "Concurso Montepio Acredita Portugal"[1]) is a health and wellness Android and Web based platform, that enables users to approach a new life style by engaging in a system that enables full control of their eating and exercising activities.

[1] Source: http://www.acreditaportugal.pt/outras-edicoes/

Inserting itself in an increasingly relevant area in a society where physical well-being is a priority for most of active-age individuals, we can divide the field of application into three complementary themes: nutrition, food and fitness. Operating as a system for monitoring user behavior, it enables the recording of calories and nutrients ingested, the energy expended in performing certain physical exercises, and provides information on various subjects associated with each of the points mentioned, in the form of a small encyclopedia.

The application has features that distinguish it from other similar ones, bringing together a system of monitoring, counseling and consultation, highlighting: the possibility of interacting directly with experienced and qualified professionals, obtaining detailed feedback according to the user's profile and history; a suggestions system that, according to the user's objective, provides personalized nutrition and fitness plans; a real-time recommendations system that assists in daily meals planning. The monitoring provided by the application allows its use as a system to support the decision making by evaluating the performance transmitted by the user, identifying priorities for action or improvement. This system allows a professional to supervise the performance of a user according to their spheres of action, in nutritional aspects, in the definition of food plans and physical activities, selecting the indicators most appropriate to the goal defined by the user.

Supported by a classic client-server architecture, where the backend provides multiple web services implemented in Ruby on Rails, that are consumed by multiple clients, namely: two web sites developed in AngularJS and two Android applications, the platform is envisioned to undergo a major refactor of the backend implementation, with the intent of establishing a consistent and well documented interface, as well an improvement of general quality of service (QoS) since some degradation of services request is being observed.

Currently the most relevant services that the backend provides allows the management (CRUD operations) of users and professionals; foods and exercises information; and food and exercises plan generations based on the user profile

The envisioned refactor of the backend structure is also open to a stack technology change, establishing an ideal scenario for the application of MDE techniques, allowing the comparison in terms of implementation consistency, quality and overall performance between the different scenarios.

This case study intents to help answering the question where the generated code is ready for being integrated and deployed in a production, or at least staging environment, providing a high quality, maintainable and evolutive solution.

1.4 Adopted methodology

The approach used to reach the proposed objectives is described in the following paragraphs, from the problem contextualization to the solution evaluation.

The framing and contextualization. Some main concepts related to the research focus of this work will be presented and correlated in the first chapters, providing an overview of their relationships, and how they work together in an API development context.

The reasoning behind the Open API Specification necessity will be also referenced, while also providing an overview of the main rules and standards defined.

An analysis of the current state of the art in the thematic of model driven engineering applied to the RESTful API applications generation will be presented, considering the limitations as well as the potential of existing solutions, making a comparative study between them.

Exploring the existing solutions. The experiment described in this thesis is in the domain of health and wellness applications. Domain analysis is chosen as the method for understanding the domain and to identify the concepts within it. The programming language used for the implementation is the Java Programming Language 1.9 (Java). Java is currently an actively used production language which is object-oriented and with proper tools and IDE support.

Based on the information retrieved in the state of the art analysis, the existing solutions were tested using the referred domain, providing some additional context on the structure of the generated code, and how it complies with the existing industry quality standards.

Example domain. The domain example chosen to aid the development of the DSL, by setting up a comparison code-base structure, is the My Pocket Nutritionist backend. The complete domain analysis of the system is described in detail in the domain analysis, Chapter 4.

The DSL development. Taking into consideration the OAS, and the data from the comparison between the existing solutions, a DSL capable of representing the domain and the respective resources, and their interactions was developed.

The domain model, the OAS, and the industry standards on the architecture definition for a well-designed API, form the basis for the DSL development.

DSLs promise flexibility in problem-solving. It is a goal to produce a model unrestricted by existing language concepts, enjoying the full range of possibilities enabled by custom syntax and semantics.

Results analysis and comparison. The solution developed was compared to the experimental java manual implementation, establishing the first level of validation of the DSL.

A comparison between the achieved solution through the developed DSL and the previously studied approaches was made, reflecting the code quality achieved, performance of the implemented services and speed of the code generation.

The use of a case study provides the opportunity to intensively analyze many specific details and difficulties. With a backend platform supported in an API that provides multiple services to be consumed by different clients, the implemented infrastructure is representative of modern small to medium startups, allowing generalizations to be made in the conclusions achieved through the development of this work.

1.5 Contribution to Knowledge

This research and its developments place a contribution in the Computer Science domain, particularly, in the autonomous code generation techniques under a Domain-Specific Modeling (DSM) environment.

The proposed solution for this project, which is presented in Chapter 4, contributes with a tool that uses a given input and translates it into a different format to enable a complete web service to run right after the code generation, without any manual coding or any further adjustments.

Furthermore, the solution proves that it is possible to generate a complete web service from its conception (domain model) to its production-ready deployment using MDE techniques.

In the same chapter it is also demonstrated that this project also allows the generation of quality and consistent code from an input model, providing database mappings, RESTful services, among other infrastructure pieces of generated software, granting a better time-to-market to get new models ready to ship to their customers.

1.6 Thesis outline

This subsection outlines the structure and organization of this document, presenting a summary of the content found in each chapter, providing a general notion of all the developed work.

In each chapter will be presented a previous summary that exposes the main thematic ones treated there, making possible the exploration by the reader only of the subsections of his interest.

In Chapter 2 the context of the development of this project is presented, framing the problem that it proposes to solve, as well as the business area in which it can be applied. A description of the main concepts that integrate this work is presented.

The current state of the art is analyzed in Chapter 3, presenting other applications with similar objectives and making an assessment analysis between them. A requirements analysis is made, defining the main implementation components. The adopted technologies are described and a DSL solution outlining, focusing in the features it pretends to deliver, is defined. An outline of the solution evaluation parameters is also presented.

In Chapter 4 the design and implementation of the DSL solution are detailed, as well the domain model of a web services project that will support the solution development. An OAS specification is elaborated based on the mentioned domain model.

From the evaluation plan outlined in Chapter 3, Chapter 5 leverages on the defined evaluation parameters to assert the solution quality.

Chapter 6 closes the document, by synthesizing the work done and listing the successful objectives. Some considerations are presented in relation to the developed solution, and areas with a margin of progression in future interventions are identified, always aiming to improve the user's perceived experience, without neglecting the process of development and implementation of new functionalities. The chapter ends with an overall appreciation of the work developed.

Additional information, not crucial to the understanding of the document and project, is included in the Annexes, providing a more detailed understanding of the document and project.

Chapter 2

Context

This chapter intends to frame the project in question by describing the problem to solve, presenting the mains concepts involved and formulating a state-of-the-art review with a compilation of existing tools and approaches to design RESTful APIs, DSL frameworks used in code generation and related topics. Findings and current developments of related projects will also be analyzed.

2.1 Problem and opportunities

Service-Oriented Architecture (SOA) has established itself as the main architectural pattern for web applications development. Services constitute the basic constructs that support rapid, low-cost and easy composition of distributed applications, allowing the integration and interaction in/between heterogeneous environments (Huhns & Singh, 2005). In SOA, a service is "a logical representation of a repeatable business activity that has a specified outcome" (SOA Work Group, 2016). This definition follows the same direction as (W3C, 2004), that defines a service as "a web interface that supports interoperable operations between different software applications using a standard messaging protocol". In these definitions, one can perceive the focus on the abstraction of the service concept. Such abstraction is needed when thinking of a model representing a business process, since it usually encapsulates some complex logic, whose implementation shouldn't be addressed by the client.

Web services, as the most popular implementation of SOA, have some fundamental characteristic's: they offer robustness and agility to business enterprises, allowing them to perform their business processes efficiently by supporting software reuse, application-to-application interoperability, design flexibility, and a loosely coupled architecture.

When the web services implementation is based on the REST software architectural style they are denominated as "RESTful web services". Conforming with the concepts of REST allows the web services to avoid the performance degradation resulting from the use of SOAP and XML (Tihomirovs & Grabis, 2016), (Haupt, Karastoyanova, Leymann, & Schroth, 2014), (Pavan, Sanjay, & Zornitza, 2012).

Services designed through the RESTful approach expose their functionality as Web resources, where each resource is addressed with a unique URI: a user can directly access a specific resource by the associated URI or traverse the offered functionality through a hierarchical structure.

The modeling process of RESTful applications is divided in structural and behavioral modeling (Schreier, 2011). Structural modeling describes the resource types, their attributes, and relations as well as their interface and representations, while the latter offers the possibility to describe the behavior with state machines. Thereafter, these models are interpreted by a developer with the purpose of gathering the necessary information to build the application. The development of the solution, in this context, usually leads to de implementation of the classic CRUD operations over the defined resources and their relationships, always taking into consideration the underlying purpose of developing a REST compliant application.

Following the constrains inherent to REST architectural style is a non-trivial task, often not fulfilled properly (Haupt et al., 2014). This leads to the fact that these applications are often not exploiting the full potential of the REST architectural style. Also, the similarity between the written code for the CRUD operations in the different resources, makes the development process tedious and error prone.

Model-driven techniques have been proposed to improve the development of complex applications (Stahl, Völter, Bettin, Haase, & Helsen, 2006). In model-driven software development (MDSD), software is not implemented manually based on informal descriptions but automatically generated based on formal models that derive from a metamodel. This approach in general leads to better code quality, fewer errors, increased reuse of best practices, better maintainability through "standardized" code, and increased portability through the separation of platform independent models (PIM) and platform specific models (PSM). Given the several issues the developers have to address to ensure a suitable design and the desired usability, an MDE based approach would vastly improve the developers productivity, while also increasing the code quality (Schreibmann & Braun, 2014).

As said before the introduction of an MDE approach to develop RESTful web services needs the definition of a formal description/model of the application to be developed. While there is not an accepted standard for describing REST Services, an emerging specification standard, the OpenAPI Specification (OAS) within the Open API Initiative (OAI), backed up by some industry giants, like Google, Microsoft, IBM, PayPal, etc., is gaining relevance as the main "programming language-agnostic interface description for REST APIs, which allows both humans and computers to discover and understand the capabilities of a service without requiring access to source code."(The Linux Foundation, 2017b).

The conjunction between the OAS as a standard to describe the web services, with a model-driven approach based on a metamodel whose implementation takes into consideration the REST constrains, would lead to a development solution that would improve the quality and productivity of software development process, allowing the developers to focus on the business core rules. The metamodel gives a generic mechanism to create formal specifications based on the OAS. Afterward, the conceptual models created can be used for generating machine readable specifications or the integration code by the corresponding transformation rules. Additionally, the OAS standard allows the mitigation of another frequent problem in RESTful web services development sphere: the proper documentation of the web service. The origin of OAS, previously known as Swagger, allows an easy integration within a web platform where all the involved stakeholders can access the authorized information related to the web services, and even test them against either a real back-end infrastructure or a mocked one.

Writing RESTful web services is a time-consuming matter and there is no guarantee that the outcome will be REST-conform. Also, when a large number of web services is to be implemented the challenges raised usually lead to redundant program code that covers the same functionality over multiple and different layers of the software architecture. This often leads to mistakes as a developer introduces unintentional errors to the repetitive code constructs. Furthermore, the configuration of web services architecture is not a trivial task and it requires a complex and redundant work to be performed (Dimitrieski et al., 2017).

Thus, summarily the main problems that are addressed by the work described in this document are as followed:

1. What MDE approach can be adopted to ensure a more efficient and reliable process of web services development?
2. Can the OAS be used as a Domain Specific Language in an MDE approach for web services development?
3. Can a code generation process be developed over the DSL referred previously, and consequently aiding developers in web services development?
4. How does an OAS-based DSL and code generation process compare with a Resource-based one, with a simpler language?

The main purpose of this work is to present a model-driven approach that integrates the OAS in the development of RESTful web services. It intends to provide a tool that allows developers to use a design first approach in the web service development process, focusing in the definition of resources and their relationships, leaving the code production process to the automation provided by MDE techniques. The code generation process should cover the entire web service flow, from the definition and exposure of the endpoints to the definition of database tables.

2.2 Value analysis and proposition

The section 2.1 provided a global context of the thematic involving the current work. This section pretends to focus on the value that is envisioned to be provided, using some models as support to demonstrate it: New Concept Development (NCD) model, Canvas model, value network analysis (Verna Allee model (Allee, 2002)) and finalizing with an Analytic Hierarchy Process (AHP) analysis.

2.2.1 Value analysis

The “Front End of Innovation” is known as a first stage of the innovation process. Duration and dynamism of such innovation "journey" very frequently depend on many factors that have an impact particularly on this stage. Peter Koen (Koen et al., 2001) proposes a model, the NCD model, that provides a common language to optimize the “Front End of Innovation”, distinguishing five different front-end elements:

- Opportunity identification: this element relates to the identification of business and/or technological opportunities;
- Opportunity analysis: the previous identified opportunities are translated to implications for the business or technological company context. The perception of the new ideas impacts must be evaluated, trying to align them with the target objective;

- Idea genesis: this element transforms the identified opportunity into a tangible product;
- Idea selection: the purpose of this element is to analyze the respective potential business value, and decide if it is worth to pursue it;
- Concept and technology development: finally, this part the front-end, a concrete business case is developed.

The following sections frame the NCD model in the context of this work, focusing the referred front end elements.

2.2.1.1 Opportunity identification

The development of web services is a time-consuming, repetitive and error-prone task, since each new common web service has the same structure of the already implemented ones and needs to be integrated in the global platform using the exact same methods (Barukh & Benatallah, 2013). Furthermore, the enforcement of the RESTful constraints in the development process adds complexity to the code to be implemented, and architecture that will support the whole system.

MDE techniques allows the common service-related low-level logic to be abstracted, organized, incrementally developed and thereby re-used, through the development of a meta-model that allows the construction of instances, the models, representing the service to be implemented. Being an established approach for developing software systems, MDE been adopted successfully in many industries (Mussbacher et al., 2014).

Both concepts, web services and MDE, in this work context bond in the definition of the API specification, where each resource and associated endpoints are defined, as well the relationships between them.

Arising from the previous statements, the development of a domain-specific language, integrating the OAS and RESTful concepts, consists in a valuable solution, granting:

- Fast paced and cost-effective development;
- Increased quality and less error-prone solutions;
- Software being less sensitive to changes in personnel;
- Advanced programmers additional time to focus on the creative aspects of their work;
- The capture of domain knowledge;
- Up-to-date documentation bridging the gap between business and IT;
- Focus on business problems instead of technology.

2.2.1.2 Opportunity analysis

Although MDE is often considered to be synonymous with code generation (or at least model-driven development), other development techniques derive from the concept:

- Model Interpretation: model directly executes on an engine (or virtual machine). No code is generated, or transformations are defined, the model is placed in a runtime environment and it executes as defined by the semantics of the model;

- Model Transformation: transforming a model into another model means that a source model is transformed into a target model based on some transformation rules.

Figure 1 shows the diffusion of MD* (related model driven techniques) techniques among companies, by respective size.

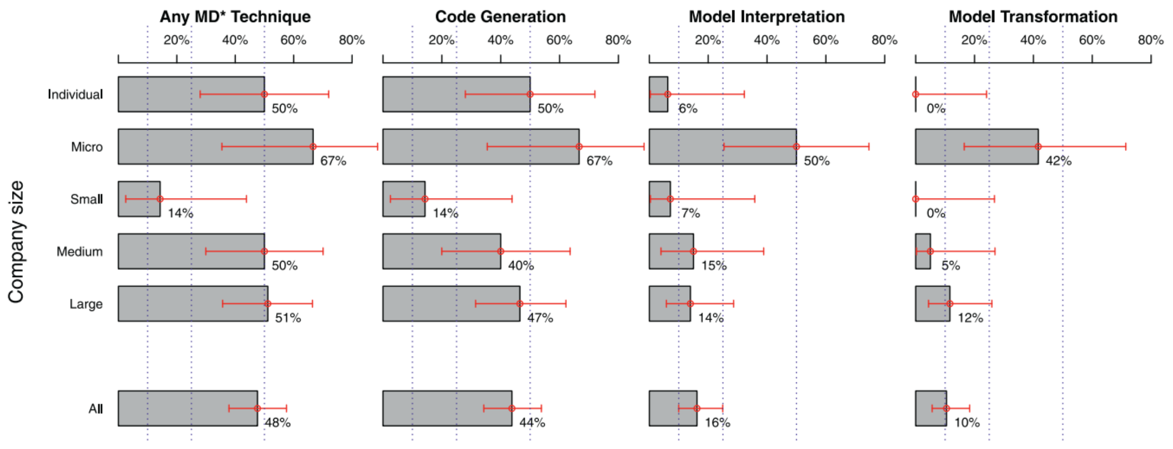


Figure 1 - MD* techniques diffusion by company size.[1]

Analyzing Figure 1, code generation reveals itself as the most used MDE approach in software development, regarding any company size. Code generation is perceived to bring benefits such as productivity, where reports show a high variability gains, from a 27% loss to an 800% gain (Mohagheghi & Dehlen, 2008). Most companies seem to experience productivity increases of between 20 to 30 percent.

Other advantages are in the support that MDE provides benefits in standardization of the implementation procedures - Figure 2 (circles indicated statistically significant difference).

[1] Source: (Torchiano et al., 2013)

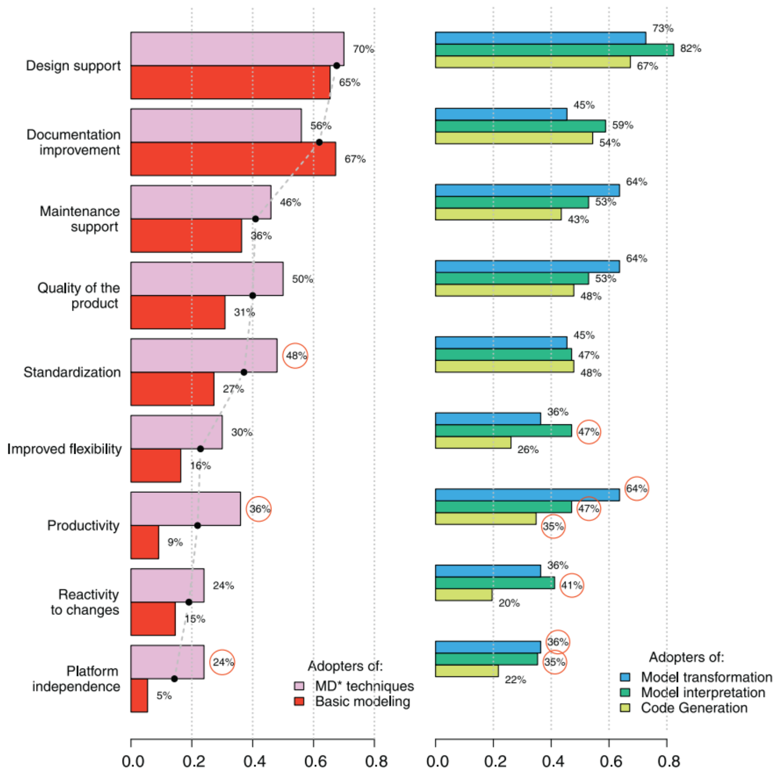


Figure 2 - Benefits achieved through MD techniques.[2]

Dealing with a developer perspective, additional reports state interesting conclusions related to productivity, problem solving capabilities, creativity and enjoyment from the use of MDE practices in applications development - Figure 3.

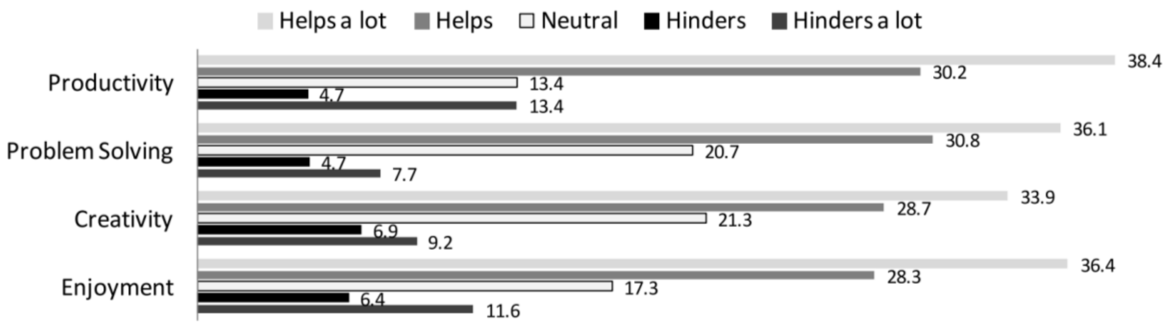


Figure 3 - How MDE affects developers experience.[1]

One of the main drawbacks in this dominion, is related to the increased training costs and substantial organizational changes when a MDE adoption is followed (Whittle, Hutchinson, & Rouncefield, 2014). The same study suggests MDE isn't appropriate for every type of organization. Companies that target a particular domain are more likely to use MDE than companies that develop generic software.

[1] Source: (Hutchinson et al., 2014)

Given the SOA proliferated use as a form of companies deliver their solutions in delivering solutions, the global API[1] management market is set of a massive growth - Figure 4.

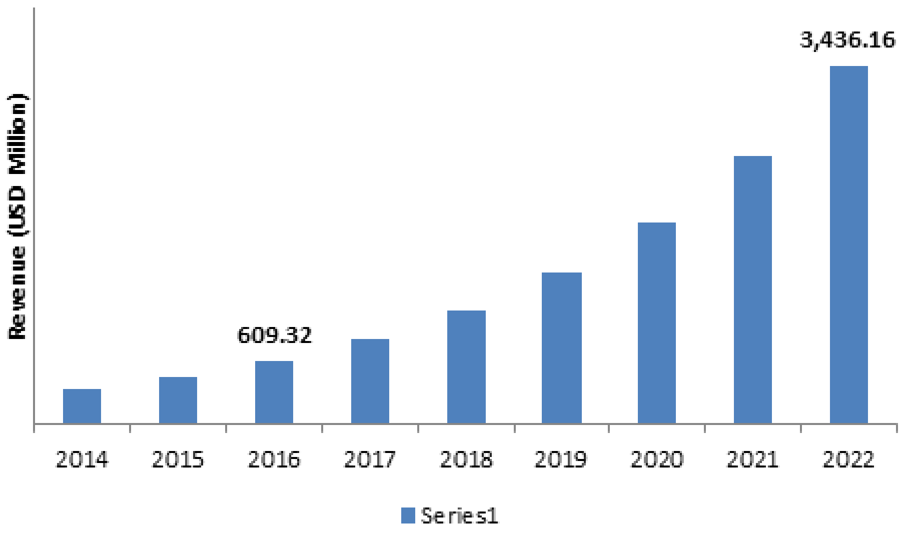


Figure 4 - Global API management market revenue, 2016-2022 (USD Millions).[2]

Advancements of the Internet of Things & Big Data, cost and feature benefits and increasing needs to manage API traffic are some of the key factors predicted to shape the API managements market. System integrators and SOA and PaaS (Platform as a Service) integrations are likely to open up new alluring opportunities for API management market in near future (Zion Market Research, 2017).

2.2.1.3 Idea generation, selection and concept definition

Considering Figure 1, where relative frequency of adoption of the MD* specific practices among modelers is depicted, from a universe of 105 modelers, 48% adopt at least one of the three key MD* techniques: code generation is in use by 44%, model interpretation by 16% and model transformation by 10%.

If the scope is narrowed down to MD* adopters only, 46 out of 50 MD* adopters (92%) use code generation, 34% use model interpretation, and 20% use model transformation.

This indicates a tendency from companies to prefer the adoption of code generation relatively to other MDE techniques.

The emerging of the OpenAPI specification, backed up by industry giants, as a standard for describing API overall behavior, that can be easily interpreted by different users, of different backgrounds, not necessarily related to the IT industry, constitutes an ideal scenario to develop a DSL based solution that interprets the specification and generates the related web services code.

[1] In this context a web service is a type of API, with a specification that almost always operates over HTTP
[2] Source: https://www.zionmarketresearch.com/news/api-management-market

Figure 5 presents the concept definition and the integration of the envisioned DSL in the development process.

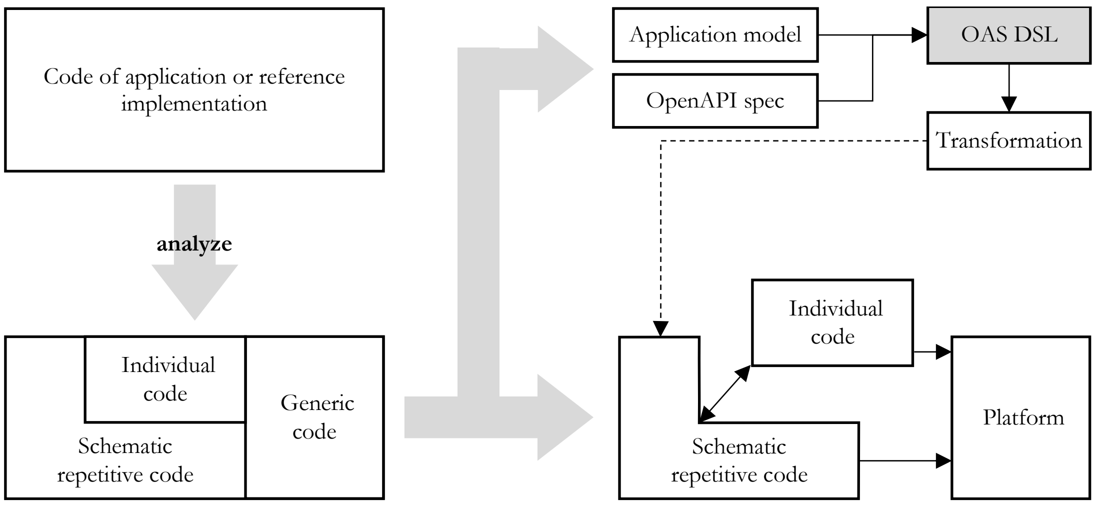


Figure 5 - Concept integration and definition, based on (Stahl et al., 2006).

Analyzing an existing application or a reference implementation (the upper left corner of the diagram), the code can be restructured in three parts (the lower left corner): a generic part that is identical for all future applications, a schematic part that is not identical for all applications, but possesses the same systematics (for example, based on the same design patterns), and finally an application-specific part that cannot be generalized.

MDSD aims to derive the schematic part from an application model. Intermediate stages can occur during transformation, but in any case, DSL, transformation (in this work context the code generation process), and platform will constitute the key elements.

2.2.2 Value proposition

Multiple definitions of value exist, but ultimately it resides in the benefit that someone or some company retrieves from using a specific product or service, making a transaction between the two parts. Following this setting, and in the context of the present work, the main value proposition that emerges is:

"MDE approach in web services implementation improves developer's productivity, companies gain, and ultimately the costumer experience, by focusing the implementation effort on the core business rules. This achieves a consistent approach and definition of the web services across the defined domain model, leading to an improvement of service quality, responsiveness, reliability and maintainability."

2.2.3 Perceived value

Perceived value, as the name suggests, relates to the benefits from a concrete individual perspective. In this case three different perspectives can be identified:

- Developers - productivity improvement from access to a fast paced, less error-prone development environment;

- Companies - costs reduction, less time to market and focus on business instead of technology concerns;
- End-users - access to a reliable and performant service.

The value to the client can be interpreted from a longitudinal perspective of value with the benefits and sacrifices encompassing four temporal values - Table 1.

Table 1 - Longitudinal perspective of value.

		Ex Ante	Transaction	Ex Post	Disposition
Developers	Benefits	-	innovation; focus on creative process	increased productivity; less errors; quality improvement	ease of maintainability and new features integration
Developers	Sacrifices	mistrust; unfamiliarity; training costs	effort; steep learning curve	-	-
Companies	Benefits	-	innovation, focus on business rules	less time to market; reliable solutions; easy evolution	service quality; increased profits
Companies	Sacrifices	mistrust; unfamiliarity; training costs	invested time	-	benefits evaluation
End-users	Benefits	-	-	better user experience; QoS and reliability	satisfaction
End-users	Sacrifices	-	-	-	-

2.2.4 Canvas model

The business model defines how to combine the means to deliver value to the interested parts and capture value to the organization. Table 2 portrays a Canvas model adapted to the context of the work developed in this thesis.

Table 2 - Canvas model.

Key partners	**Key activities**	**Value propositions**	**Costumer relationships**	**Costumer segments**
Open API Initiative	Programming	OAS DSL improves developer's productivity, companies gain, and ultimately the costumer experience, by focusing the implementation effort on the core business rules. This achieves a consistent approach and definition of the web services across the defined domain model, leading to an improvement of service quality, responsiveness, reliability and maintainability.	Web services reliability and quality	Developers
	Web services development			Developers will have a tool to optimize productivity and improve overall quality of developed software
	Evolution/Maintenance of web services solutions			Companies
	Migration of technological stacks			*Business area* Costs reduction (development and maintenance); Less time to market; Reliable solutions
	Key resources		**Channels**	*Domain knowledge area* Ease in conveying the business requirements
	Human Resources		Digital marketing	Final users
	Hardware		Technical conferences	Better service quality and reliability
	Software		Technical workshops	
	OpenAPI specification			

Cost structure	**Revenue streams**
Human Resources	Licensing the DSL
Hardware	Web services development
Software Licenses	Maintenance contracts

2.2.5 Value network analysis (Verna Allee model)

The Value Network Analysis exists on the premise that work itself can be model as a network (Allee, 2002). Value networks are sets of roles and values exchanges that generate a specific kind of value, allowing the discovery of:

- How the work gets done;
- The kind of value being created;
- How efficiently an organization converts resources (inputs) to value outputs (value conversion);
- Failures points in the networks.

Figure 6 portrays the value network associated with the current work, where the roles involved are identified, as well the tangible and intangible relationships/transactions.

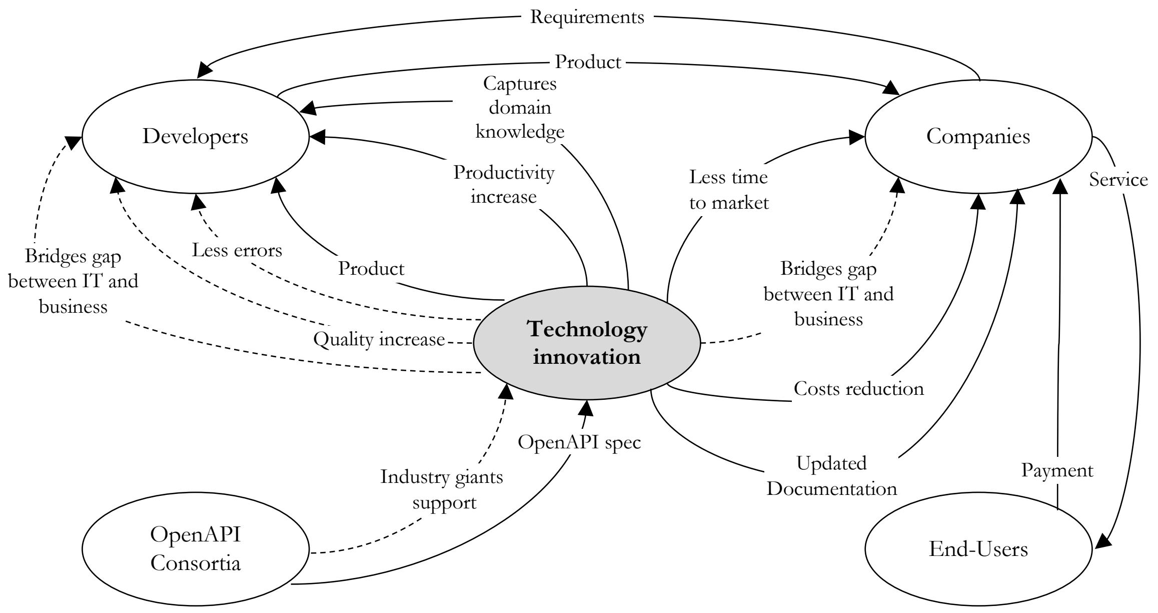


Figure 6 - Value network analysis.

In the network shown the relationships/transactions (solid line: tangible relation; dashed line: intangible relation) between the five roles identified:

- Technology innovation: the DSL developed in the context of this work;
- Developers: the individuals that will use and model the business concepts and relationships using the developed DSL;
- Companies: will provide the requirements and business model, from which developers implement the required web service, that later will be made available by the company, with the purpose of making profit;
- End-users: the final users of the web service;
- OpenAPI Consortia: provide the OpenAPI specification, supported by multiple companies.

2.2.6 AHP analysis

One of the main methods developed in the field of Multicriteria Discrete Decisions is the AHP, created by Professor Thoma L. Saaty in 1980.

This method allows the use of qualitative as well as quantitative criteria in the evaluation process. The main idea is to divide the decision problem into hierarchical levels, thus facilitating its comprehension and evaluation.

Figure 7 shows the hierarchy tree associated with the current work, where three hierarchical levels are portrayed:

1. Problem statement;
2. Criteria;
3. Alternatives.

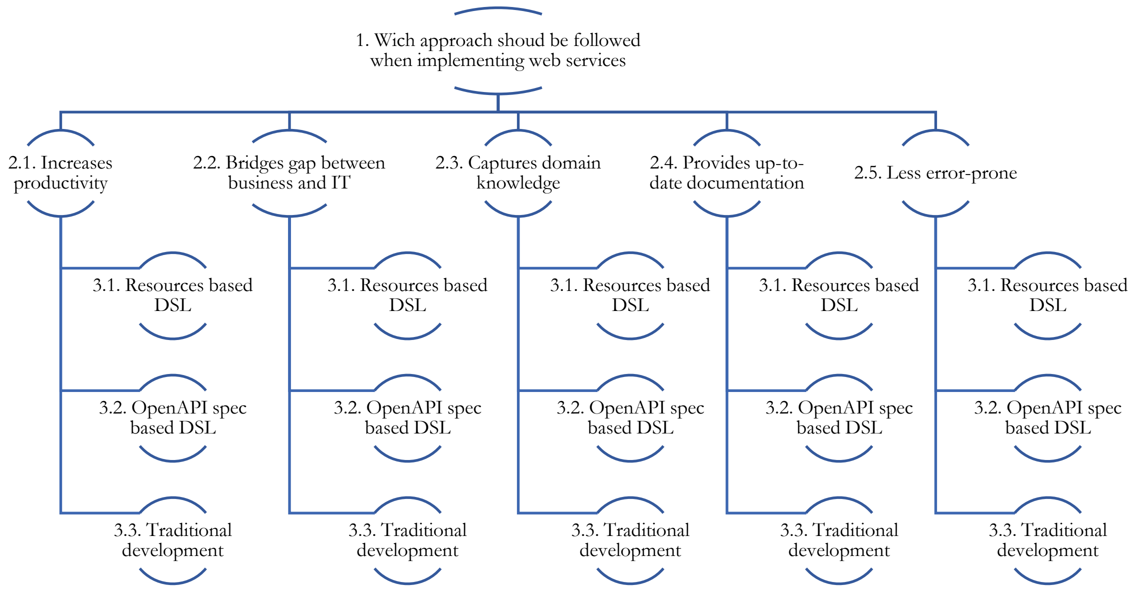

1. Wich approach shoud be followed when implementing web services
2.1. Increases productivity
2.2. Bridges gap between business and IT
2.3. Captures domain knowledge
2.4. Provides up-to-date documentation
2.5. Less error-prone
3.1. Resources based DSL
3.2. OpenAPI spec based DSL
3.3. Traditional development

Figure 7 - Hierarchy tree.

After building the hierarchical three, the second phase consists in establishing priorities among the elements for each level of the hierarchy, by means of a comparison matrix, following a comparison scale from values in the interval 1 through 9 (Saaty, 2008) - Table 3.

Table 3 - Criteria pairwise comparison.

	2.1.	**2.2.**	**2.3.**	**2.4.**	**2.5.**
2.1.	1	8	4	7	1
2.2.	1/8	1	1/5	1/5	1/9
2.3.	1/4	5	1	3	1/5
2.4.	1/7	5	1/3	1	1/8
2.5.	1	9	5	8	1

The next step consists in the normalization of comparison matrix, and determination of the priority vector - Table 4.

Table 4 - Normalized matrix and priority vector.

	2.1.	2.2.	2.3.	2.4.	2.5.	Priority vector
2.1.	0.3972	0.2857	0.3797	0.3646	0.4105	37%
2.2.	0.0496	0.0357	0.0190	0.0104	0.0456	3%
2.3.	0.0993	0.1786	0.0949	0.1563	0.0821	12%
2.4.	0.0567	0.1786	0.0316	0.0521	0.0513	7%
2.5.	0.3972	0.3214	0.4747	0.4167	0.4105	40%

The next step is to calculate the Consistency Ratio (CR) to measure how consistent the judgments were in relation to large random samples of judgments. Evaluations of the AHP method assume that the decision maker is rational, that is, if A is preferred to B and B is preferable to C, then A is preferred to C.

If the CR is greater than 0.1 the judgments are not reliable because they are too close to the comfort of randomness, in this case the obtained results do not present consistent values.

In this case a CR of 0.082 was reached, so it can be concluded that the relative priority values of the example used are consistent.

All the previous procedures for constructing the comparison matrix and determining the relative priority of each criterion must be made again, and now observe the relative importance of each of the alternatives that make up the hierarchical structure of the problem in question - Appendix Appendix A.

Table 5 - Alternative choice.

	2.1.	2.2.	2.3.	2.4.	2.5.	Alternative
3.1.	0.4814	0.3661	0.6594	0.4667	0.4737	**0.4953**
3.2.	0.4629	0.5753	0.2825	0.4667	0.4737	0.4491
3.3.	0.0557	0.0586	0.0580	0.0667	0.0526	0.0556

Based on the AHP analysis, the resources-based DSL presents itself as the best solution for the web service implementation.

The following sections describe the key concepts involved in the development of this work and its relationships.

2.3 RESTful web services

2.3.1 REST

REST is an acronym for REpresentational State Transfer and it is an "architectural style for distributed hypermedia systems" (Fielding, 2000). REST is not considered to be an architecture but it is described as a "set of constraints applied to elements within the architecture"(Giessler, Gebhart, Sarancin, Steinegger, & Abeck, 2015). Fielding describes the constraints in his dissertation as:

- **Client-Server**: this feature is most commonly found in Web applications. A server, with a set of services available, listens for requests to these services. A client, who wants to run a specific service in an available server, sends a request to the server. The server can then either reject or execute the requested service and return a response to the client;
- **Stateless**: another constraint imposed by the REST style concerns the interaction between client and server. Communication must be done without storing any type of state on the server, i.e. every client request to the server must hold all the information necessary for it to be understood. Therefore, session states, when needed, must be fully supported on the client;
- **Cache**: one way to lessen the impact of the downside brought about by performance reduction is by using cache. It also requires that data from a response, coming from a request to the server, to be marked as cacheable or noncacheable. If an answer is set as cacheable, then it will be reused in response to future equivalent requests;
- **Interface/Uniform contract**: the central feature that distinguishes REST architectural style from other network-based styles is its emphasis on a uniform interface between components (client, server). To obtain a uniform interface, REST defines four interface requirements: (i) identification of resources; (ii) manipulation of resources through representations; (iii) self-descriptive messages and; (iv) hypermedia as the application state mechanism;
- **Layered system**: to improve the scalability requirement of the Internet, to the REST style was added the layering feature. Multilayer systems use layers to separate different units by its responsibilities. The main disadvantage of this model is the addition of overhead and latency in the processed data, reducing performance. For a network-based system that supports caching, this drawback can be mitigated;
- **Code-on-demand**: the last item in the set proposed by the REST style is an optional feature. REST allows clients to have the ability to directly download and execute code on the client side. This way, it simplifies the client side and focuses on extensibility, in contrast, reduces visibility. A known practical example of Code-On-Demand is Adobe Flash: A user (client) requests a Web page which contains a link to a SWF using a web browser. After the request, the Web page is transported to the client machine together with a SWF and executed.

2.3.2 HTTP verbs

The HTTP protocol has 9 different methods. Only 6 of them are widely used: GET, POST, PUT, DELETE, HEAD and OPTIONS. HEAD and OPTIONS are special methods. HEAD is used to return only the headers of the response and OPTION is for getting allowed methods on a resource (Nguyen, Qafmolla, & Richta, 2014). The others are used for operating with resources.

Table 6 - HTTP common verbs.[1]

Verb	**Action**
GET	Used to retrieve a representation of a resource. It is a read-only, *idempotent*, and safe operation.
PUT	Used to update a reference to a resource on the server and it is *idempotent* as well.
POST	Used to create a resource on the server based on the data included in the body request. It is the only *nonidempotent* and *unsafe* operation of HTTP.

[1] From (Selic, 2003)

Table 6 - HTTP common verbs.[1]

Verb	Action
DELETE	Used to remove a resource on the server. It is *idempotent* as well.
HEAD	Like GET but returning only a response code and the header associated with the request.
OPTIONS	Used to request information about the communication options of the addressed resource (e.g., security capabilities such as CORS).

2.3.3 RESTful web services

Web services are currently the most common way to exchange data among information systems. Web services comprehend some fundamental characteristics: they are self-contained, modular and dynamic (Vasudevan, 2017). SOAP (Simple Object Access Protocol) and REST are the most usual implementation of web services, where each of these approaches has its own advantages and disadvantages. Its fundamental to choose the right type of web services, otherwise it can lead to certain problems in data exchange or impose some restrictions.

The constrains identified by Fielding (API Evangelist, 2015) and explained in section 2.3.1, if fulfilled by a web service implementation and behavior, compose the essential requirements to define the service as RESTful. The only exception is "Code on Demand", since it is an optional constraint and has not to be implemented by a web service.

Giessler et al (API Evangelist, 2015) identified, collected, and categorized best practices for a quality-oriented design of RESTful web services:

- **No versioning**: RESTful web services completely avoid the necessity of a versioning strategy due to the hypermedia constrain of REST architectures. Building on this, RESTful web services can be compared with traditional websites, where the content remains available across multiple web browsers even when changes are made;
- **Resources description**: the resources defined in the RESTful web services abstract the underlying domain model and associated entities. This makes the description extremely relevant, since there is a direct connection between its quality and the usability of the web service. In this perspective some best practices are recommended:
  1. Nouns should be used for resources names;
  2. The resource name must be domain specific and concise, allowing the semantics inference without additional information;
  3. The number of resources should be limited, to avoid an overly complex system. This recommendation is highly dependent of the abstraction level of the base domain model;
  4. Consistency between the use of plural or singular in the resources naming must be enforced;
  5. The JavaScript naming conventions should be used since JSON (JavaScript Object Notation) is the preferred media format for message communication.
- **Identification of resources**: an URI should be used for the resources identification, unique for each one of them:

1. The URI must be self-explanatory;
2. A resource URI should be composed of two parts: the first one represents a set of states specific to the resource and the other one a specific state of the previously mentioned state;
3. According to the Open Web Application Security Project (OWASP), the identifier of a resource specific state must be difficult to guess and a direct reference to the associated object should be avoided;
4. Verbs should not exist within the URI, since this kind of policy implies a method-oriented style, such as SOAP.

- **Error handling**: the level of abstraction introduced through the defined web service resources, leads to the requirement of an error messaging system that must provide clear and understandable data associated with the origin of the error:

1. HTTP status code should be reduced to the minimum viably possible that allows the fast identification of the problem;
2. HTTP specification must be used in the employment of application explicit error status codes;
3. The error message should comprise four components:
   a) A message to developers describing the cause of the error, and ideally providing some hints on how to solve it;
   b) A message to be shown to the user;
   c) An application specific code;
   d) A hyperlink to additional information about the error.

- Parameters usage: each resource URI functionality can be improved with parameters, to send optional information to the service:

1. Filtering: allowing the resources to be filtered by its attributes or through a special query language;
2. Sorting: to sort the information a comma divided list with the attributes followed by the "-" or "+" is recommended, defining the order and respective parameter from which the results should be fetched;
3. Selection: choose which information should be returned by the web service response through a comma separated list of attributes;
4. Pagination: it enables the information division through several virtual pages, referencing, as well, the existence of next and previous pages.

- Interaction with resources: the underlying REST architectural style of the web service dictates that the interaction between the client and server is made through a representation of a resource. The communication is established through the HTTP protocol, and should follow the following recommendations:

1. The used HTTP methods should conform to the method's semantics defined in the official HTTP specification. Table 7 summarizes the most used HTTP methods and their characteristics.

2. If a large amount of data has to be transmitted support of HTTP-OPTIONS is recommended since it allows a client to request the supported methods of the current representation before transmitting information over the shared medium.

Table 7 - Most used HTTP methods characteristics.

HTTP method	Safe	Idempotent
POST	No	No
GET	Yes	Yes
PUT	No	Yes
DELETE	No	Yes

- Support of MIME Types: Multipurpose Internet Mail Extensions (MIME) types are used for the identification of data formats:

1. At least two representation formats should be supported by the web service, such as JSON or Extensible Markup Language;
2. JSON should be the default representation format since its increasing distribution;
3. Hypermedia MIME types should be used;
4. The client should be able to choose the representational format through the HTTP header field “ACCEPT”.

2.4 Model-Driven Engineering

Using models to raise the level of abstraction and automate the development process of building software is the core process of Model-Driven Engineering paradigm. To cope with complexity, abstraction is a fundamental technique, whereas automation is an effective method to increase productivity and quality (Ed-douibi, Izquierdo, Gómez, Tisi, & Cabot, 2016).

In model-driven software development, the first-class elements are models, which are all structured by a metamodel. Concisely, models are defined according to the semantics of a metamodel, which is a model for specifying models.

Model-driven engineering methodologies have been applied as a solution for better reaction to business trends and aims to increase efficiency as well as bring more agility to the development life-cycle of cloud and distributed systems (“EMF-REST Documentation,” 2015).

2.4.1 Domain Specific Languages (DSLs)

Many computer languages are domain specific rather than general purpose languages(GPLs). DSLs trade generality for expressiveness in a limited domain. By providing notations and constructs tailored toward a particular application domain, they offer substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs.

The following is a grammar for arithmetic expressions using only addition - Code snippet 1.

Code snippet 1 - Simple example grammar.

```
Expression ::= number | number "+" expression
number ::= [1-9][0-9]*
```

For simple arithmetic, 1+2+3 is according to grammar, while 1+2+ is not. The vocabulary of this language is: expression, number and addition. The semantics of this simple language is: the first and second operands of an expression are added.

The vocabulary of a DSL is taken from its domain. The syntax should be created to fit the domain, and preferably also the conventions within the domain.

2.5 OpenAPI specification

Before detailing the OpenAPI specification it is important to clarify what differs between an API specification and an API documentation.

API documentation as its name implies is simply that - documentation of an API, with examples of how developers can use each function (or, in a web API context, each endpoint), and the constrains that the API allows (Kristopher Sandoval, 2016).

API specification is much more concerned with the overall behavior of the API, and how it links to other APIs. Taking as example the OAS, a variety of functions is showed, how they are called and what they do. Additionally, a general overview of how they relate to one another, and how they can be used to more fully leverage the API is presented (Kristopher Sandoval, 2016).

Documentation is essentially how to do something, whereas specification is essentially how something should function, and what the user should expect.

2.5.1 API standardized design

API design is the creation of an effective interface that allows better maintaining and implementing an API, while enabling consumers to easily use this API (Hutchinson, Whittle, & Rouncefield, 2014). The use of a specification in the design process leads to predictable, industry-consistent experiences for users of company/enterprise APIs. But most of all, the specification defines a template to fill out the API, making clear what information is needed and how it is organized and structured.

Most organizations standardize design using Style Guidelines[1] [2] [3], assuring a consistency in the way APIs are designed and implemented to:

- provide a better developer experience;
- save time and money in the development process;
- improve the API sustainability.

Specifications provide a shareable definition that can establish an understanding across team members and project stakeholders, while also providing a machine-readable definition that can be used in documentation, and other systems, and client tooling. API specifications are central to the

[1] Source: https://github.com/Microsoft/api-guidelines
[2] Source: https://cloud.google.com/apis/design/
[3] Source: https://github.com/paypal/api-standards/blob/master/api-style-guide.md

API design process. The resulting definition acts as a contract throughout the technical, business, and legal side of API operations.

2.5.2 The OpenAPI specification

RESTful APIs being described in multiple and heterogeneous ways, complicated their understanding by potential consumers and incremented the amount of implementation logic needed to interact with different services. In order to solve these problems and standardize the process of defining RESTful APIs, some proposals emerged.

The OpenAPI Specification, originally known as the Swagger Specification, was born when SmartBear, the company that maintained the Swagger specification and associated tools, announced that it was helping create a new organization, under the sponsorship of the Linux Foundation, called the Open API Initiative. A variety of companies, including Google, IBM and Microsoft are founding members. SmartBear donated the Swagger specification to the new group. RAML and API Blueprint were also under consideration by the group. On 1 January 2016, the Swagger specification was renamed the OpenAPI Specification.

Basically, an OpenAPI Specification file describes an API, including (among others):

1. General information about the API;
2. Available paths (/resources);
3. Available operations on each path (get/resources);
4. Input/output for each operation.

The OpenAPI Specification is a formal specification for RESTful APIs, providing a way of describing them using JSON or YAML documents. Building on these formats makes OpenAPI equally accessible to humans and machines - Code snippet 2.

Code snippet 2 - OpenAPI 3.0 YAML standard example[1].

```yaml
openapi: "3.0.0"
info:
  version: 1.0.0
  title: Swagger Petstore
  license:
    name: MIT
servers:
  - url: http://petstore.swagger.io/v1
paths:
  /pets:
    get:
      summary: List all pets
      operationId: listPets
      tags:
        - pets
      parameters:
        - name: limit
          in: query
          description: How many items to return at one time (max 100)
          required: false
          schema:
            type: integer
            format: int32
      responses:
        '200':
          description: An paged array of pets
```

[1] Source: https://github.com/OAI/OpenAPI-Specification/blob/master/examples/v3.0/petstore.yaml

Code snippet 2 - OpenAPI 3.0 YAML standard example[1].

```
          headers:
            x-next:
              description: A link to the next page of responses
              schema:
                type: string
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Pets"
        default:
          description: unexpected error
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/Error"
```

2.5.3 OpenAPI 3.0.1 specification

At the time of writing the version 3.0.1 is the most recent one. Figure 8 shows the evolution from OpenAPI 2.0 to 3.0.

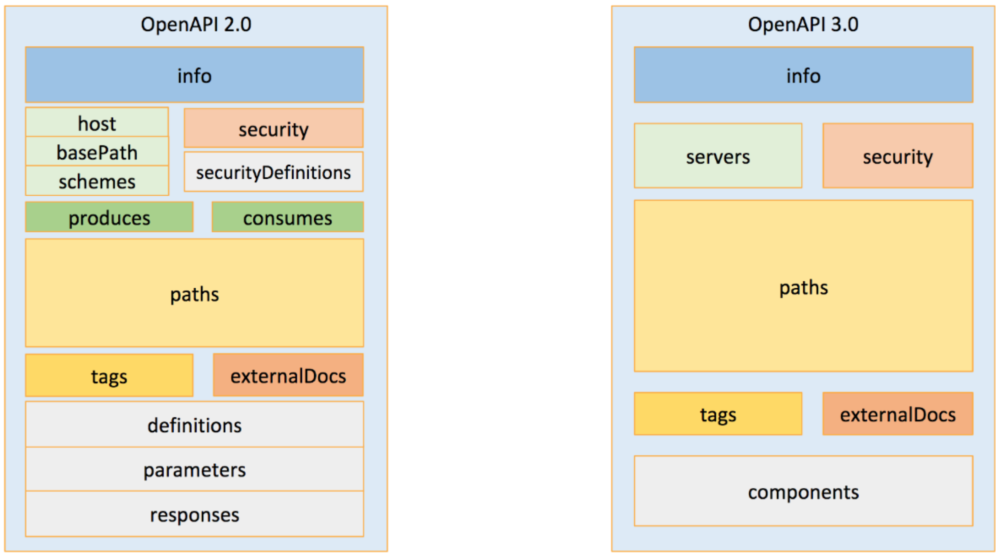


Figure 8 - OpenAPI evolution.[1]

The new version of the OpenAPI Specification differs from its predecessor by a clearer structure. At the top level, the structure has been cleaned up, with the result that the new version is incompatible with the existing one (but migration is possible automatically).

Based on Figure 8, the following code snippets, show the most relevant objects that integrate an OAS file:

- Info Object: contains basic information about the API, including the title, a description, version, link to the license, link to the terms of service, and contact information.

Code snippet 3 - OpenAPI 3.0.1: Info object example.

```
title: Sample Pet Store App
description: This is a sample server for a pet store.
termsOfService: http://example.com/terms/
```

[1] From: https://apievangelist.com/2017/03/16/what-will-it-take-to-evolve-openapi-tooling-to-version-30/

```
contact:
  name: API Support
  url: http://www.example.com/support
  email: support@example.com
license:
  name: Apache 2.0
  url: https://www.apache.org/licenses/LICENSE-2.0.html
version: 1.0.1
```

- Server object: specification of the basepath used in the API requests. The basepath is the part of the URL that appears before the endpoint.

Code snippet 4 - OpenAPI 3.0.1: Server object example.

```
servers:
- url: https://development.gigantic-server.com/v1
  description: Development server
- url: https://staging.gigantic-server.com/v1
  description: Staging server
- url: https://api.gigantic-server.com/v1
  description: Production server
```

- Security scheme object: defines a security scheme that can be used by the operations.

Code snippet 5 - OpenAPI 3.0.1: Security scheme object example.

```
type: http
scheme: bearer
bearerFormat: JWT
```

- Paths object: holds the relative paths to the individual endpoints and their operations.

Code snippet 6 - OpenAPI 3.0.1: Paths object example.

```
/pets:
  get:
    description: Returns all pets from the system that the user has access to
    responses:
      '200':
        description: A list of pets.
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/pet'
```

- Tag object: adds metadata to a single tag that is used by the Operation Object.

Code snippet 7 - OpenAPI 3.0.1 Tag object example.

```
name: pet
description: Pets operations
```

- Components object: holds a set of reusable objects for different aspects of the OAS.

Code snippet 8 - OpenAPI 3.0.1: Components object example.

```
components:
  schemas:
    Category:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
    Tag:
      type: object
      properties:
        id:
          type: integer
          format: int64
        name:
          type: string
  parameters:
    skipParam:
      name: skip
      in: query
      description: number of items to skip
      required: true
      schema:
        type: integer
        format: int32
    limitParam:
      name: limit
      in: query
      description: max records to return
      required: true
      schema:
        type: integer
        format: int32
  responses:
    NotFound:
      description: Entity not found.
    IllegalInput:
      description: Illegal input for operation.
    GeneralError:
      description: General Error
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/GeneralError'
  securitySchemes:
    api_key:
      type: apiKey
      name: api_key
      in: header
    petstore_auth:
      type: oauth2
      flows:
        implicit:
          authorizationUrl: http://example.org/api/oauth/dialog
          scopes:
            write:pets: modify pets in your account
            read:pets: read your pets
```

- External documentation object: allows referencing an external resource for extended documentation.

Code snippet 9 - OpenAPI 3.0.1: External documentation object example.

```
description: Find more info here
url: https://example.com
```

OpenAPI is backed up by a large ecosystem of tools that helps design, build, document, and consume RESTful APIs.

2.5.4 Other REST API Specifications

2.5.4.1 RESTful API Modeling Language Specification

RAML stands for REST API Modelling Language, and it is an YAML-based language used to describe REST API parameters and endpoints. This format makes it easier to represent the hierarchy of the operations and to reuse some parts of the code. It also has some interesting tooling: automatic documentation generator, online editor and code generator.

The Code snippet 10 shows a partial example of a RAML API specification file.

Code snippet 10 - RAML 1.0 standard example.[1]

```
#%RAML 1.0
title: Hello world # required title

/greeting: # optional resource
get: # HTTP method declaration
  responses: # declare a response
    200: # HTTP status code
      body: # declare content of response
        application/json: # media type
          # structural definition of a response (schema or type)
          type: object
          properties:
            message: string
          example: # example how a response looks like
            message: "Hello world"
```

2.5.4.2 API Blueprint

API Blueprint is a high-level language for describe web APIs. The syntax is a combination of Markdown and MSON syntax - Code snippet 11.

The syntax constitutes a great advantage of the API Blueprint: using Markdown to describe the API, greatly facilitates the editing of documents, even by those who are not familiar with coding languages.

Code snippet 11 - API Blueprint standard example.[2]

```
FORMAT: 1A

# Polls

Polls is a simple API allowing consumers to view polls and vote in them.

## Questions Collection [/questions]

### List All Questions [GET]

+ Response 200 (application/json)

        {
            "question": "Favourite programming language?",
            "choices": [
                {
                    "choice": "Swift",
                    "votes": 2048
                }, {
                    "choice": "Python",
```

[1] source: https://github.com/raml-org/raml-spec
[2] source: https://help.apiary.io/api_101/api_blueprint_tutorial/

```
                    "votes": 1024
                }
            ]
        }
```

2.5.4.3 Web Application Description Language (WADL)

The Web Application Description Language (WADL) is a machine-readable XML description of HTTP-based web services. WADL models the resources provided by a service and the relationships between them. WADL is intended to simplify the reuse of web services that are based on the existing HTTP architecture of the Web. It is platform and language independent and aims to promote reuse of applications beyond the basic use in a web browser.

2.5.4.4 RESTful API Description Language (RADL)

RESTful API Description Language (RADL) is an XML vocabulary for describing Hypermedia-driven RESTful APIs. Unlike most HTTP API description languages, RADL focuses on defining a truly hypermedia-driven REST API from the client's point of view, moving the conversation forward.

2.5.4.5 RESTful Service Description Language (RSDL)

The RESTful Service Description Language (RSDL) is a machine-and human-readable XML description of HTTP-based web applications (typically REST web services), adding another way to describe APIs in a machine-readable format.

Chapter 3

MDE in REST web services development

Aiding REST web services development through MDE techniques is not a new idea and some existing solutions have already addressed this kind of approach, which are described in this chapter. Their main features are also compared to support a solution perspective, focusing on the appropriate technologies and the DSL design. Finally, an evaluation procedure to assess the solution suitability is described.

3.1 Related work - State of the Art

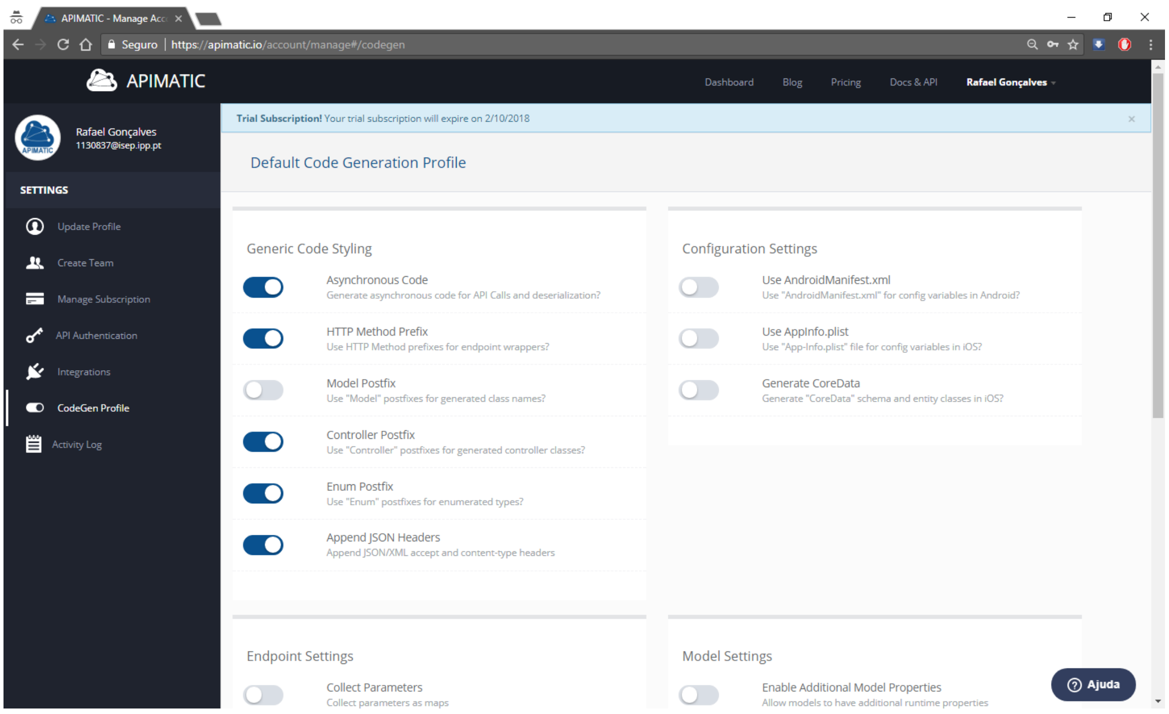
This section briefly summarizes currently available tools which are relevant to REST web services implementation and code generation.

3.1.1 Apimatic

Apimatic[1] describes itself has a "code-generation-as-a-service" platform. It is an automatic SDK generator for REST APIs that tries to cover all aspects concerning REST, from the API definition on the backend, to generation of SDKs to the frontend. With Apimatic the developer must interactively define resources and their attributes using a web-interface. Apimatic then generates an API providing CRUD operations on these resources and SDKs to be used in different clients.

Apimatic allows developers to import Swagger or WADL descriptions of an API to their platform, streamlining the creation of web services and easing the migration from existing projects. The code generation can also be configured through a simple interface (Figure 9), where the developer can decide some generic behaviors and implementation details.

[1] https://apimatic.io/



APIMATIC - Manage Acc

Seguro | https://apimatic.io/account/manage#/codegen

APIMATIC

Dashboard Blog Pricing Docs & API Rafael Gonçalves

Rafael Gonçalves
1130837@isep.ipp.pt

SETTINGS
- Update Profile
- Create Team
- Manage Subscription
- API Authentication
- Integrations
- CodeGen Profile
- Activity Log

Trial Subscription! Your trial subscription will expire on 2/10/2018

Default Code Generation Profile

Generic Code Styling

- ● Asynchronous Code — Generate asynchronous code for API Calls and deserialization?
- ● HTTP Method Prefix — Use HTTP Method prefixes for endpoint wrappers?
- ○ Model Postfix — Use "Model" postfixes for generated class names?
- ● Controller Postfix — Use "Controller" postfixes for generated controller classes?
- ● Enum Postfix — Use "Enum" postfixes for enumerated types?
- ● Append JSON Headers — Append JSON/XML accept and content-type headers

Configuration Settings

- ○ Use AndroidManifest.xml — Use "AndroidManifest.xml" for config variables in Android?
- ○ Use AppInfo.plist — Use "App-Info.plist" file for config variables in iOS?
- ○ Generate CoreData — Generate "CoreData" schema and entity classes in iOS?

Endpoint Settings

- ○ Collect Parameters — Collect parameters as maps

Model Settings

- ○ Enable Additional Model Properties — Allow models to have additional runtime properties

Figure 9 - Apimatic code generation configuration page.

Currently Apimatic does not support the concept of hypermedia, which is essential to increase the flexibility of an API.

3.1.2 RAML

This tool was already referenced in the previous chapter, as a REST specification, but it comprises a pack of tools to define, create, test, and publish RESTful APIs. RAML uses YAML as markup language and is based on the idea of defining resources and their representations as JSON schemas. The created schema is used as an input in the code generator tool, RAML[1] for JAX-RS[2], scaffolding a JAVA + JAX-RS application based on the RAML API definition.

At the time of this work, the code generator only supports JAX-RS (Java API for RESTful Web Services), a Java programming language API spec. RAML offers three possibilities to use this tool:

- Using the command line, raml-to-jaxrs-cli:

Code snippet 12 - RAML command line sintax.

```
usage: ramltojaxrs -d <arg> [-g <arg>] [-m <arg>] [-r <arg>] [-s <arg>]
-d,--directory <arg>              generation directory
-j,--json-mapper <arg>            sonschema2pojo annotation types
-g,--generate-types-with <arg>    generate types with plugins
-m,--model-package <arg>          model package
-r,--resource-package <arg>       resource package
-s,--support-package <arg>        support package
```

[1] https://raml.org/

[2] https://github.com/mulesoft-labs/raml-for-jax-rs

- Using the Gradle plugin, where multiple configuration options must be defined:

Table 8 - RAML gradle configuration options.

Property	Description	Required
sourceDirectory	The path to the directory containing source *.raml files	Yes
outputDirectory	The output directory for the generated JAX-RS resource source files.	Yes
supportPackageName	The package used for support classes.	Yes
resourcePackageName	The package used for resource classes.	Yes
modelPackageName	The package used for type classes.	Yes
jsonMapper	The annotation style used for jsonschema objects	No
jsonMapperConfiguration	Options for jsonschema objects (jsonschema2pojo)	No
generateTypesWith	options for annotating RAML types	No

- Using Maven plugin:

Code snippet 13 - Maven configuration example.

```
<build>
    <plugins>
        <plugin>
            <groupId>org.raml.jaxrs</groupId>
            <artifactId>raml-to-jaxrs-maven-plugin</artifactId>
            <version>$VERSION</version>
            <dependencies>
                <dependency>
                    <groupId>org.raml.jaxrs</groupId>
                    <artifactId>jaxrs-code-generator</artifactId>
                    <version>$VERSION</version>
                </dependency>
            </dependencies>
            <configuration>
                <ramlFile>${path}/types_user_defined.raml</ramlFile>
                <resourcePackage>example.resources</resourcePackage>
                <modelPackage>example.model</modelPackage>
                <supportPackage>example.support</supportPackage>
                <generateTypesWith>
                    <value>jackson</value>
                </generateTypesWith>
            </configuration>
        </plugin>
    </plugins>
</build>
```

RAML code generator only accepts as input RAML specification files and, like Apimatic, doesn't support hypermedia implementation.

3.1.3 Swagger Codegen

Inserted in the Swagger ecosystem, the Swagger Codegen[1] generates API client libraries (SDK generation), server stubs and documentation automatically given an OpenAPI Specification.

Swagger Codegen comes with 25+ server stub generators for different server-side frameworks such as PHP Symfony, C# Nancy, Java Spring, Python Flask, etc. The auto-generated server-side code allows back-end developers to easily implement a RESTful backend given an OpenAPI/Swagger 2.0 specification file.

[1] https://swagger.io/swagger-codegen/

The Codegen project provides a command-line interface (CLI), which is a framework for plugins supporting output to various technologies - Code snippet 14.

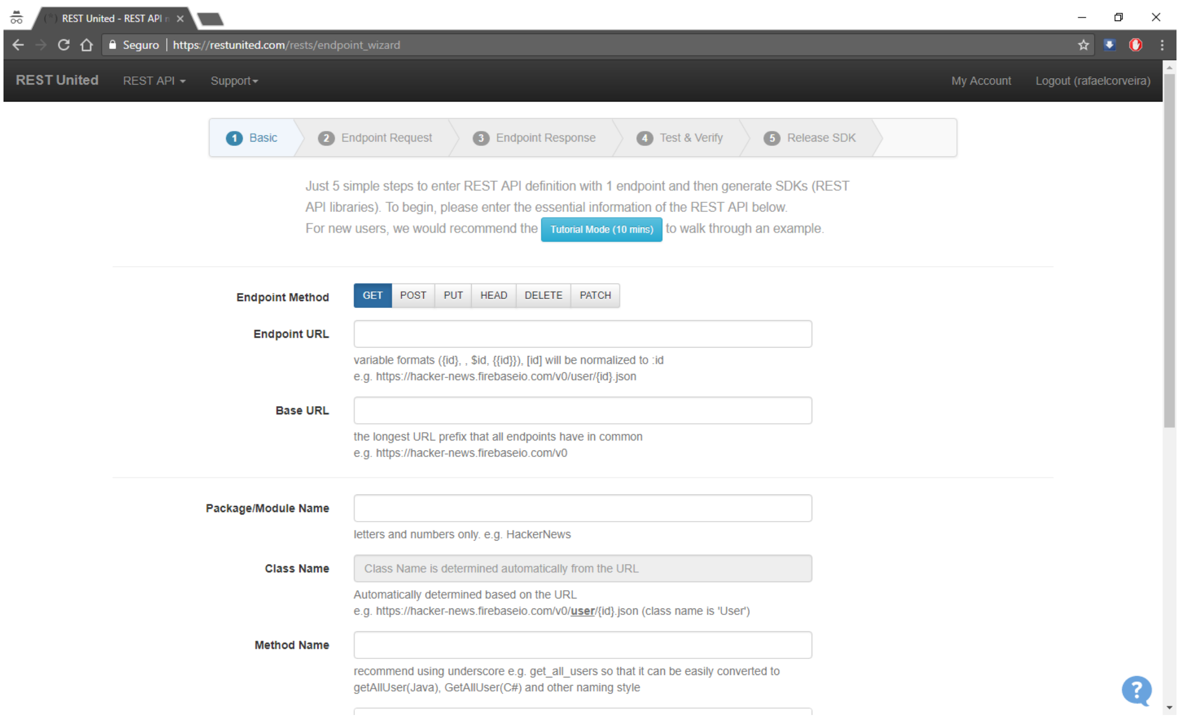
Code snippet 14 - Swagger Codegen command line syntax.

```
java -jar swagger-codegen-cli.jar generate \
     -i https://apis.voicebase.com/v3/defs/v3-api.yaml \
     -l java \
     -c java-config.json \
     -o v3client
```

Swagger Code gen only accepts Swagger/OpenAPI spec files as input to code generation, and. like the previous frameworks, it doesn't support hypermedia.

3.1.4 REST United

REST United[1] features an easy-to-use interface that allows users to build automatically generated API client libraries (SDK generation) with customizable documentation and code samples. It uses a customized version of Swagger Codegen project (Torchiano, Tomassetti, Ricca, Tiso, & Reggio, 2013). REST United offers an easy-to-use wizard to generate SDKs for a REST API in 5 steps - Figure 10.



Just 5 simple steps to enter REST API definition with 1 endpoint and then generate SDKs (REST API libraries). To begin, please enter the essential information of the REST API below.
For new users, we would recommend the Tutorial Mode (10 mins) to walk through an example.

Endpoint Method: GET | POST | PUT | HEAD | DELETE | PATCH

Endpoint URL
variable formats ({id}, , $id, {{id}}), [id] will be normalized to :id
e.g. https://hacker-news.firebaseio.com/v0/user/{id}.json

Base URL
the longest URL prefix that all endpoints have in common
e.g. https://hacker-news.firebaseio.com/v0

Package/Module Name
letters and numbers only. e.g. HackerNews

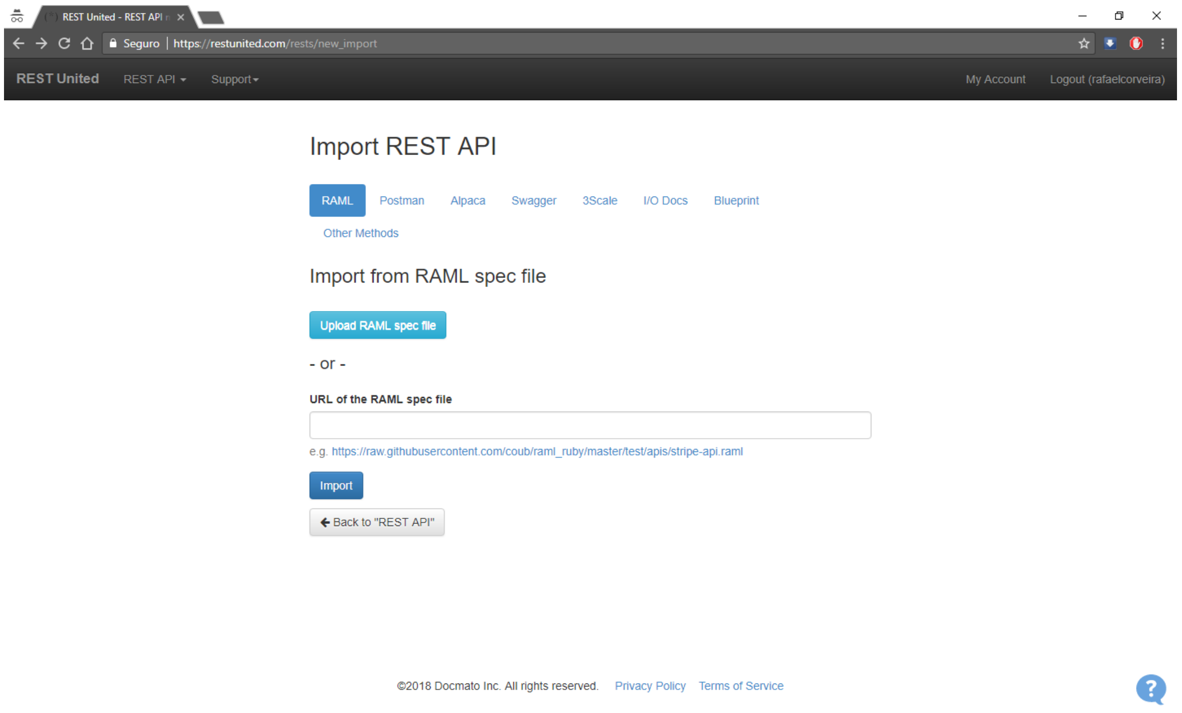
Class Name: Class Name is determined automatically from the URL
Automatically determined based on the URL
e.g. https://hacker-news.firebaseio.com/v0/user/{id}.json (class name is 'User')

Method Name
recommend using underscore e.g. get_all_users so that it can be easily converted to getAllUser(Java), GetAllUser(C#) and other naming style

Figure 10 - REST United code generation configuration page.

[1] https://restunited.com/

It is also possible to import an existing REST API definition from a range of formats - Figure 11.



Import REST API

Import from RAML spec file

- or -

URL of the RAML spec file

e.g. https://raw.githubusercontent.com/coub/raml_ruby/master/test/apis/stripe-api.raml

©2018 Docmato Inc. All rights reserved.

Figure 11 - REST United import specification wizard.

3.1.5 Restlet Framework

Restlet Framework[1] is a Java based framework to develop REST APIs in the same programming language. It complies with REST API specifications, supports standard security and authentication methods, and, with the built-web server, provides an environment suitable for both server and client Web applications.

Restlet Studio uses Swagger CodeGen for Objective-C, but has its own CodeGen engine for Android and Java (Sharma & Chug, 2015).

3.1.6 AutoRest

The AutoRest[2] tool generates client libraries for accessing RESTful web services. Input to AutoRest is a spec that describes the REST API using the OpenAPI Specification format.

It uses a configuration file to control the code generation process - Code snippet 15.

[1] https://restlet.com/open-source/
[2] https://github.com/Azure/autorest

Code snippet 15 - AutoRest configuration file example and command line syntax.

```
input-file: petstore.json # full Unicode support

csharp:
  namespace: Petstore
  output-folder: Client
  enable-xml: true     # enable experimental XML serialization support
  # azure-arm: true    # uncomment this line to enable code generation in the Azure flavor

autorest [config-file.md] [additional options]
```

AutoRest can generate client-side code from the Swagger specification files. The generator supports C#, Java, Node, Python and Ruby programming languages.

3.1.7 EMF REST

EMF REST[1] is a framework build on the top of the Eclipse/Java/EMF development stack and it transforms an ecore model into a functional REST API. This is a solution for developers familiar with EMF and ecore models. It also provides a JavaScript library for the generated API, so the developer can include this library and use it as a middle-man in the communication between the server and the client. It is meant to be a solution useful for prototyping and validation purposes (Hutchinson et al., 2014).

EMF-REST automatically creates a RESTful API conforming to the JAX-RS specification that can be automatically deployed in an application server.

This solution has some drawbacks:

- No support for custom endpoints, only CRUD operations are supported;
- It is not obvious if the solution is using any database or the returned data is just static;
- The supported HTTP methods are POST, PUT, DELETE and GET (Sharma & Chug, 2015).This means that it does not support requests that first ask the server for available methods with an OPTIONS call.

3.2 Comparative analysis

From a high-level analysis of the previous identified frameworks, a comparative assessment of some features presents itself in Table 9.

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

Table 9 - Code generation tools comparison.

	Apimatic	RAML	Swagger-codegen	REST United	Restlet Framework	AutoRest	EMF REST
Authentication code	Yes	No	Yes	Yes	Yes	Yes	Yes
Hypermedia support	No	No	No	No	Yes	No	No
Most common supported specifications	API Blueprint, WADL, WSDL, RAML, OAS	RAML	Swagger 2.0	RAML, Swagger 2.0, 3Scale, I/O Docs Blueprint	Swagger 2.0 RAML	Swagger 2.0	-
Code quality	Code comments, Coding standards for some languages	-	-	-	Code comments	Code comments	-
Language support	Java, C#, iOS, Android, PHP, Ruby, Python, Golang, Angularjs, Nodejs	Java	30 languages, including: Ada, C#, C++, Clojure, Erlang, Java, Kotlin, PHP, etc.	Android, C#, ActionScript, Java, Objective-C, PHP, Python, Ruby, Scala	Android, Java, Objective-C, AngularJS, Node.js	C#, Go, Java, Node.js, TypeScript, Python, Ruby, PHP	Java

It can be concluded that, overall, the available solutions present the same drawbacks:

- Poor hypermedia support;
- Poor code quality measures;
- Poor support of OAS 3.0;
- No database related scripts generation;
- No integration and unit tests support.

This expose the major areas where the solution developed should focus, trying to overcome these limitations, and revealing itself as a strong alternative to the existing code generation tools based on OAS.

3.3 Solution perspective

The main goal of this work in to bridge the gap between high-level concepts of REST and the low-level of implementation of a web interface in a specific programming language, through an MDE approach, using the OAS as a design guide. To validate the viability of the proposed approach a simpler language should be implemented first, focused on the REST concepts, namely the resource definition, from which a code generation process must then be developed. The knowledge acquired and the implementation itself from this simpler methodology will aid the construction of the OAS-oriented DSL processes, allowing to narrow the scope inside the multiple objects that the specification includes, choosing the most relevant ones for the code generation process.

This approach pretends to leverage on MDE techniques, DSL and code generation specifically, to produce RESTful web services out of plain data models alleviating and speeding up the development process from a developer perspective by enabling a language with concise set of concepts which are specific to the domain of REST web services development.

Such language should allow developers to have a single specification of a web service without writing any boilerplate or redundant code.

API development usually follows one of two schools of thoughts: the design-first and the code-first approaches. The code-first approach follows a traditional process to build APIs, with the code development being made after the business requirement are laid out, and eventually generating the documentation from the code. A design-first approach advocates for designing the API's contract before writing any code.

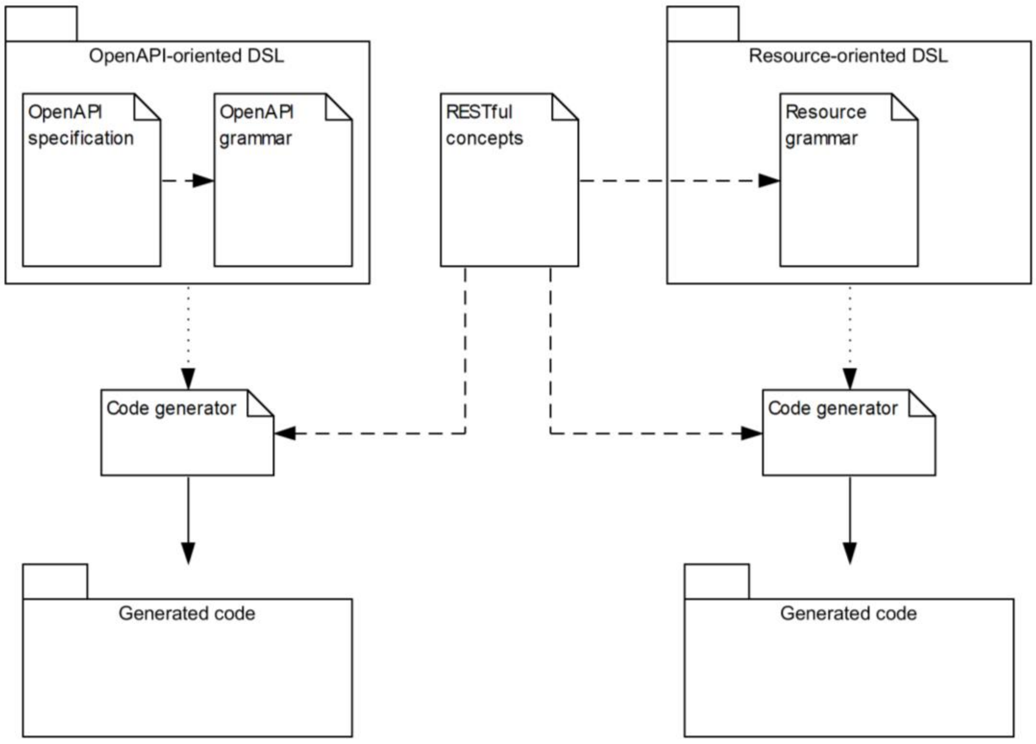


Figure 12 - DSLs' design alternatives.

Following this line of thought two high level solution alternatives were identified for the DSL - Figure 12:

1. Develop a DSL whose base concepts have a one-to-one direct relation to the OpenAPI specification objects, using the same syntax and semantic inherent to the specification, while following the REST architecture constrains (contract-first driven API based approach):

   Foreseen advantages:

   - This approach narrows the gap between business and technical components, by using a formal language easily understandable by the non-technical side;
   - Allows developers to focus on the API design, decreasing the learning curve in the integration procedures, increasing reuse, value and engagement;
   - Loose coupling between contract and implementation is possible in this approach.

   A DSL implementation that follows this approach would need developers to invest time learning the OpenAPI specification and used it like a modeling language. Figure 13 illustrates the main concepts that intervene in the DSL specification.

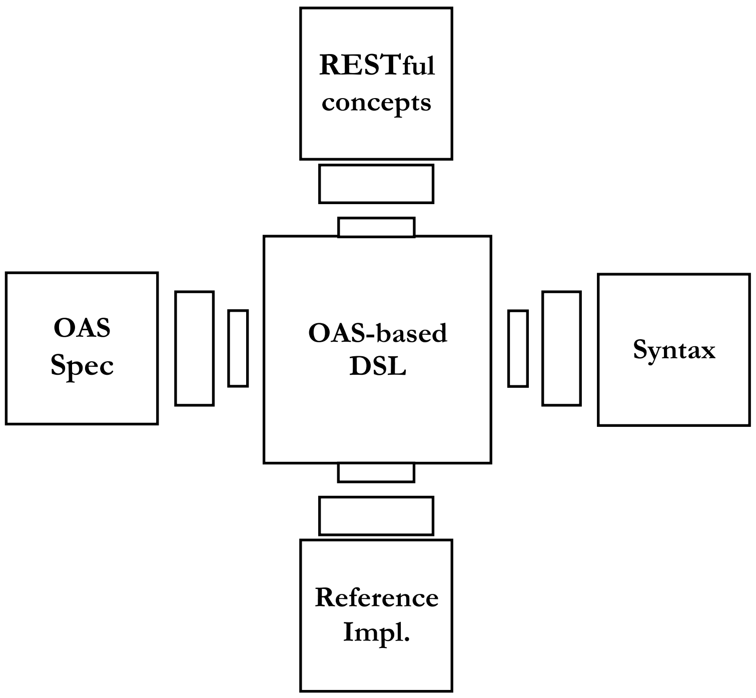


Figure 13 - OAS-based DSL context.

2. Develop a DSL that takes on the concepts of RESTful architectures and resources-based syntax, whose web interface is granted to be OpenAPI compliant (code-first driven API based approach):

   Foreseen advantages:

   - No need of depth knowledge of OpenAPI specification;
   - More intuitive DSL: the language follows a syntax more readable from the developer perspective, being based on concepts derived from coding procedures, leading to a faster implementation.

   While this approach might lead to a faster development, it might be difficult to establish it has a central draft that keeps all the involved parts updated with the API's objectives.

Table 10 summarizes the high-level advantages and advantages from each approach.

Table 10 - Comparative analysis between DSLs approaches.

	OAS based approach	**Resources based approach**
Communication and understandability between involved parts	X	
OAS knowledge required	X	
Loose coupling between contract and implementation	X	
Short learning curve		X
Convention over configuration approach		X

Both solutions should ensure that the REST constrains are respected, generating true RESTful web services.

As an assurance of code quality, industry standards should integrate the code generation process, observing common principles like SOLID and GRASP, and applying design patterns when suitable. Tests should be inferred, allowing testable code to be generated, improving the overall delivered code quality.

Chapter 4 details the presented alternatives design and implementations.

The following section meets the requirements acknowledged from the previous analysis, retrieved either from the comparative study of the commercial solutions, and the two approaches identified.

3.4 Requirements analysis

In this section, the requirements of the DSLs are to be identified and gathered, as well the intended user profiles for the DSLs usage and its context of use. The requirements at this step mean identifying what main functionalities the DSLs should provide, and how they should do it.

3.4.1 Functional requirements

The main requirement related to the DSLs functionality is the OAS support, either as the input language or by ensuring the compliance of the generated web service. Moreover, the code generation process should cover all the architectural layers and constrains defined for the web service, highlighting the ones associated with the RESTful architectural style.

This initial analysis divides the main requirements in three subsets: the web service reference implementation; the OpenAPI specification support; and the code generation process. Table 11 summarizes the functional requirements identified in the context of the current work.

Table 11 - DSL functional requirements.

Nº	Description		
Req_01	Create web service reference implementation that will guide the DSLs development		
Req_02	OpenAPI specification support	Req_02.1	Define DSL grammar - OAS-based DSL grammar - Resource-based DSL grammar
		Req_02.2	Implement Xpect tests
Req_03	Code generation	Req_03.1	Generate database tables creation script
		Req_03.2	Generate project structure
		Req_03.3	Implement RESTful compliant architectural style
		Req_03.4	Generate gateways layer
		Req_03.5	Generate presentation layer
		Req_03.6	Generate business layer
		Req_03.7	Generate database access layer
		Req_03.8	Generate unit and integration tests
		Req_03.9	Generate code comments
		Req_03.10	Generate the OAS spec from the Resource-based DSL

From the exposed requirements the main derivate Use Cases are presented in Figure 14.

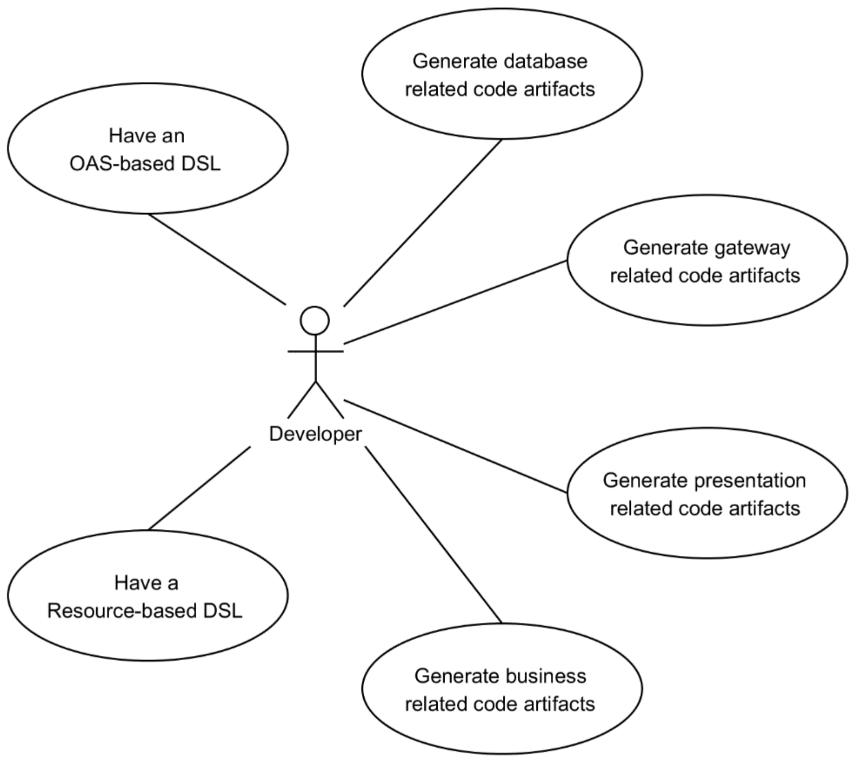

Have an OAS-based DSL
Generate database related code artifacts
Generate gateway related code artifacts
Developer
Generate presentation related code artifacts
Have a Resource-based DSL
Generate business related code artifacts

Figure 14 - OAS-based DSL context.

3.4.2 Non-functional requirements

The non-functional requirements can be divided in three areas, the first focused on the web service reference implementation; the second on the DSLs and the last one in the generated web services.

The reference implementation should be constructed following adequate software principles, namely:

- SOLID (Single responsibility principle; Open/closed principle; Liskov substitution principle; Interface segregation principle; Dependency inversion principle);
- DRY (Don’t Repeat Yourself).

Regarding the DSLs the following non-functional requirements were identified:

- Usability: the extent to which the DSL can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a stated context of use;
- DSL overall performance when generating the related code, namely the time taken to have the code artifacts available to deploy;
- Readability: how easy is to read and understand the language;
- Writeability: how easy is to model the web service through the language.

Concerning the web service generated, the main non-functional requirements to observe are:

- Generated code quality;

- Web service overall performance;
- QoS;
- Testability.

It is also relevant characterize the user's profiles expected to take the more benefits from the two different concepts-based DSLs and their context of use:

- Given that the OAS-oriented DSL it is based on the specification itself, it is crucial that the developer has a thorough knowledge of the specification. Only this way the productivity gains from the DSL usage will be visible in the development process;
- Regarding the Resource-oriented DSL, it only requires knowledge over concepts concerning web services development, being adequate for new entry developers, even those with less experience in MDE approaches. It can also provide a first contact with the OAS specification and how it relates itself with the Resource concept and related code artifacts.

3.5 Adopted technologies

This section emphasis on describing the technologies involved in the development of this work, focusing on the web service reference implementation and DSL development process. It is important to state that the focus of this work is not to present or address the available technologies related to this development context, but to give the reader a brief presentation of the used technologies, chosen based on the authors professional experience.

3.5.1 Reference web service implementation

Java was chosen as the coding language used to build the reference implementation of the web service that supports the DSL development. The following sub-sections detail the technological components that aided the web service construction:

- JAX-RS: an API that eases the development of applications that integrate REST architectures;
- EclipseLink: Java Persistence API (JPA) 2.1 specification reference implementation. JPA describes the management of relational data in applications using Java.

In addition, an open source database server, MySQL server. MySQL is a database management system (DBMS), which uses Structured Query Language (SQL) as the interface. It is currently one of the most popular databases, ranking number two in popularity[1].

3.5.1.1 JAX-RS

In Java, support for the implementation of RESTful Web Services was added in 2008 by the JSR-311 specification, which was named JAX-RS (Oracle, 2018b). This specification was created to simplify the development of REST applications and quickly became of utmost importance as it

[1] Source: https://db-engines.com/en/ranking

was one of the first frameworks based on POJO classes and annotations capable of publishing RESTful services.

The Jersey (Oracle, 2018a) RESTful Web Services framework is the reference implementation of the JAX-RS specification, available as an open-source framework. Code snippet 16 illustrates an example of a web service annotated with Jersey components.

Code snippet 16 - JAX-RS web service example.[1]

```
/**
 * Retrieves representation of an instance of helloWorld.HelloWorld
 * @return an instance of java.lang.String
 */
@GET
@Produces("text/html")
public String getHtml() {
  return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";
}
```

3.5.1.2 EclipseLink

EclipseLink[2] is an Open Source Eclipse Fundation Project that allows Java developers to interact with various types of information services such as Database, Web Services, XML Objects, EIS, etc. Therefore, EclipseLink implements not only the Java Persistence API (JPA) standard, but also other standards such as JAXB, JCA, and SDO.

JPA is a specification that regulates very powerful tools to automate and save time in development processes. This specification helps in all processes related to database interactions, in a way that it can be used to execute queries, inserts, updates and deletes. It permits the developer to work directly with objects rather than with SQL statements.

"Entity" is the base concept of JPA. JPA uses a database table for every entity. All entity classes must define a primary key, must have a non-arg constructor and or not allowed to be final. Keys can be a single field or a combination of fields.

Code snippet 17 - JPA entity example.[3]

```
@Entity
public class Todo {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String summary;
    private String description;

    public String getSummary() {…}

    public String getDescription() {…}
}
```

Code snippet 18 shows the find example, that allows retrieve a specific entity from the associated id.

[1] Source: https://docs.oracle.com/javaee/6/tutorial/doc/gipzz.html

[2] http://www.eclipse.org/eclipselink/

[3] Source: http://www.vogella.com/tutorials/JavaPersistenceAPI/article.html

Code snippet 18 - EclipseLink find example.[1]

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
    ...
    public Employee findEmployee(Integer employeeId) {
        return (Employee) em.find(Employee.class, employeeId);
    }
    ...
```

The persist example is presented in Code snippet 19.

Code snippet 19 - EclipseLink persist example.[2]

```
@Stateless
public class EmployeeDemoSessionEJB implements EmployeeDemoSession {
    ...
    public void createEmployee(String fName, String lName) {
        Employee employee  = new Employee();
        employee.setFirstName(fName);
        employee.setLastName(lName);
        em.persist(employee);
    }
    ...
```

3.5.2 DSL implementation

Xtext and Xtend, which are the main technologies and frameworks used for the DSL implementation are briefly described in the next sections.

Additionally, Xpect was used. It consists in a unit and integration testing framework, based on Junit, that stores test date in any kind of texts files. The core focus of Xpect is on testing Xtext languages and supporting the process of designing Xtext languages.

3.5.2.1 Xtext

Xtext was the elected framework to implement the DSL. It covers all aspects of a complete language infrastructure, starting from the parser, code generator, or interpreter, up to a complete Eclipse IDE integration with all the typical IDE features. A standalone specific version, called Eclipse DSL, that natively supports this framework is available, currently at version "Oxygen 2". Xtext generates a default grammar example, that can be seen in the following code snippet.

Code snippet 20 - Xtext default generated grammar.

```
grammar org.example.entities.Entities with org.eclipse.xtext.common.Terminals

generate entities "http://www.example.org/entities/Entities"

Model:
       greetings+=Greeting*;
Greeting:
       'Hello' name = ID '!';
```

3.5.2.2 Xtend

[1] Source: http://wiki.eclipse.org/EclipseLink/Examples/JPA/Configure

[2] Source: http://wiki.eclipse.org/EclipseLink/Examples/JPA/Configure

The code generation process was achieved through Xtend which is a fully featured general purpose Java-like language that is completely interoperable with Java. Xtend has a more concise syntax than Java and provides powerful features such as type inference, extension methods, dispatch methods, and lambda expressions, not to mention multiline template expressions, which are useful when writing code generators.

Since Xtend is completely interoperable with Java, all the Java libraries can be reused. Moreover, all the Eclipse JDT (Java Development Tools) will work with Xtend seamlessly.

Code snippet 21 - Hello word print statement in Xtend.

```
package org.example.xtend.examples

class XtendHelloWorld {
        def static void main(String[] args) {
                println("Hello World")
        }
}
```

The similarities with Java are evident, though the removal of syntactic noise is already obvious by the fact that terminating semicolons (;) are optional in Xtend. All method declarations start with either def or override. Methods are public by default.

3.6 Solution evaluation methodology

The assessment of the developed solution overall suitability, giving the proposed objectives, requires the definition of evaluation plan, where the relevant metrics to be measured and compared are identified, as well the hypothesis and related test methods.

First, an appraisal of the two proposed DSLs design is needed, trying to understand which grants a better user experience when defining/developing a web service.

A second evaluation component relates to two strands of quality measurement:

1. The quality of the generated code;
2. The quality of service (QoS) of the related web services.

This section intents to identify the project evaluation metrics, hypothesis and tests methods, whose detailed implementation is the subject of **Erro! A origem da referência não foi encontrada.**.

3.6.1 DSLs usability

With two DSLs design alternatives, an evaluation of the languages usability has a major relevance in the context of this work. The end user of the DSL can be a domain expert, a programmer that works on specific domain or a regular domain user. Each of these users can have different background profiles and its own role in problem solution. A comparable validation procedure, that will assess user experience with the DSLs is envisioned in this section, based on user-interface (UI) evaluation:

"DSLs define a way for human to communicate with machines. Therefore, DSL evaluation should not be much different from evaluating a regular UI" (Barišić, Amaral, & Goulão, 2012).

3.6.1.1 General approach to usability evaluation

Usability evaluation can be assessed through four different ways (Barišić, Amaral, Goulão, & Barroca, 2012):

- **Formally**: using models and simulations to predict measures such as time to complete a task or the difficulty of learning to use a product. Some models have the potential advantage that they can be used without the need for any prototype to be developed;
- **Automatically**: this can be done by automated checking of conformance to guidelines and standards or by evaluation of data collected during system usage. This kind of evaluation is possible when initial prototypes or initial versions of full implementation are available.
- **Empirically**: evaluation with users is recommended at all stages of development if possible, or at least in final stage of development. Formative methods that focus on understanding the user's behavior, intentions and expectations to understand any problems encountered can be used to establish and test user requirements. Iterative testing with small numbers of participants is preferable, starting early in design and development process.
- **Heuristically**: by simply looking at the product and passing judgment according to an own opinion. It is usually considered as evaluation conducted by expert and it can be used when initial prototypes are available.

Moreover, the same authors propose in (Barišić, Amaral, & Goulão, 2018) multiple patterns for DSL evaluation, to be used during DSLs' life-cycle. These patterns can by grouped in three high-level guidelines:

- **Agile development process**: by including patterns devoted to management and engineering scopes of a domain specific language, an iterative approach allows the continuous tracking of the usability requirements and evaluation;
- **Iterative User-Centered Design**: provides patterns to include an Evaluator in the development process, that will collect information concerning the DSL developer, mainly focused on the Domain Users interpretations;
- **Experimental evaluation design**: supports the experiment execution (e.g. hypothesis, tests, samples, etc.);

The methodology proposed by the referred authors involves a continuous assessment and evaluation of the DSL, both during and after their development. Further evaluation analysis considerations are exposed in Section 5.1.

3.6.1.2 Evaluation method

Empirical method - Evaluation with users

Expert assessment techniques are more geared to filter problems and refine issues related to the visual communication of systems, and as such, cannot be considered as substitutes for a user-conducted assessment (Mendes, 2014).

The unpredictability of users is one of the main factors that this type of evaluation can consider, involving the measurement of the performance of users accomplishing typical tasks of those who use the system under evaluation. Usually combines observational techniques, questionnaires and interviews, ideally in controlled environments (rooms equipped with video and audio recording equipment, for example), and can be performed in the usual place of use of the systems being tested. Figure 15 shows that under normal conditions, 5 people undergoing the same usability assessment test can detect 85% of the problems and 3 are able to detect more than half of these same problems.

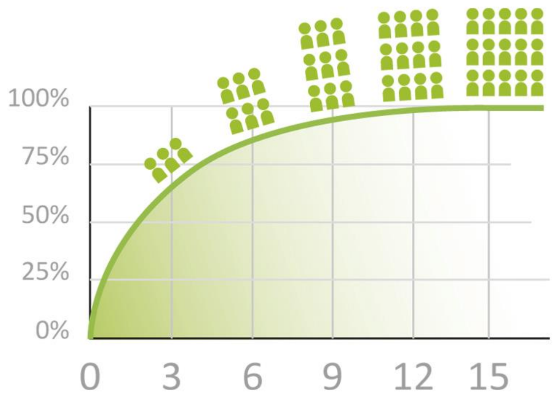


Figure 15 - Percentage of problems encountered and number of users.[1]

For logistical reasons, the main means of obtaining data for the analysis was to carry out a survey, with monitoring the response time, testing the participants and their ability to perform some tasks.

The SUS questionnaire (Brooke, 1996) was selected as a data collection instrument because it is simple and fast, showing the overall view of the user in relation to the system.

3.6.1.3 Statistical analysis

For she statistical analysis of the data collected in the questionnaires, the study process was divided into three main points:

- Sample characterization, identifying some features of the responders;
- Performance of the participants in the execution of the proposed tasks, by measuring the time taken in each one;
- The usability analysis through the answers to the SUS questionnaire.

3.6.2 Code metrics

Given the nature of the current work where the web services code is being generated through a DSL, the analysis of the generated code quality is of major significance. This section synthetizes the relevant metrics in code analysis, procedures to measure them and the hypothesis to be validated.

[1] From: (Manuel, 2011)

3.6.2.1 Metrics

A software metric is a quantitative measure of the degree to which a software system possesses properties like coupling, cohesion, inheritance, abstraction, among others. The objective is to have reproducible and quantifiable measurements with numerous valuable applications such as schedule and budget planning, cost estimation, quality assurance testing, software debugging, software performance optimization, and optimal personnel task assignments (Hutchinson et al., 2014).

Table 12 - List of static metrics.[1]

Direct	**Indirect**
WMC (Methods per Class)	The sum of McCabes's cyclomatic complexities of all local methods in a class.
DIT (Depth of Inheritance Tree)	The metric measures class level in the inheritance tree, root class is considered as zero.
NOC (Number of Children)	It counts number of immediate sub classes of a class in a hierarchy.
CBO (Coupling between Objects)	It represents the number of classes to which the given class is coupled.
RFC (Response for a Class)	The number of local methods plus the number of non-local methods called by local methods
LCOM (Lack of Cohesion of Methods)	The number of disjoint sets of local methods. Each method in a disjoint set shares at least one instance variable with at least one member of the same set.
LOC (Lines of code)	The number of lines of code excluding comments.
AMC (Average Method Complexity)	This metric measures the average method size for each class.
Ca (Afferent couplings)	A class's afferent couplings is a measure of how many other classes use the specific class
Ce (Efferent couplings)	A class's efferent coupling is a measure of how many other classes is used by the specific class.
NPM (Number of Public Methods)	All methods which have public access specifiers are counted by NPM.
DAM (Data Access Metric)	The metric computes the ratio of attributes declared as private or protected to the total number of attributes declared in the class.
MOA (Measure of Aggregation)	It measures the HAS-A relationship between attributes at run time.
MFA (Measure of Functional Abstraction)	The metrics computes the count of the number of inherited methods of a class divide by the total number of methods which are accessible by member methods of the class.
CAM (Cohesion Among Methods of Class)	The relevance between the class methods based on the list of specifications of the methods is computed by this metric.
IC (Inheritance Coupling)	This metric produces the total number of super classes to which a given class is coupled.
CBM (Coupling Between Methods)	The metric measures the total count of new or redefined methods to which all the inherited methods are coupled.
AMC (Average Method complexity)	The average method size for each class is measured by AMC, where the size of a method is equivalent to the number of java binary codes in the method.

3.6.2.2 Evaluation methods

Giving the extended number of static metrics to evaluate, multiple automatic solutions to collect this kind of data are available. In the context of this work, and since the target language that will

[1] From: (Sharma & Chug, 2015)

be generated is Java, the elected evaluation tool to gather code metrics was CKJM[1], focused on the analysis of OO code.

After obtaining static metrics values for the generated code, a comparison analysis against the code obtained from the available commercial solutions that accept OpenAPI specification files as input, and the case study was made.

3.6.2.3 Testing hypotheses

As testing hypotheses, to analyze the significance of the static metrics performance differences between the developed solution, the case study, and the available commercial solutions, it can be defined:

- H01: No significant difference exists between the performance of static metrics of the generated code and the case study project.
- H11: The generated code metrics performance is relatively better that the ones calculated from the case study project.
- H02: No significant difference exists between the performance of static metrics of the generated code and the commercial available solutions.
- H12: The generated code metrics performance is relatively better that the ones calculated from the commercial available solutions.

Also, a descriptive statistical analysis will be conducted, gathering common attributes like, mean, median, standards deviation, etc., drawing some conclusions over the obtained values.

3.6.2.4 Statistical analysis

The scenarios can be validated using hypothesis tests. The hypothesis test allows to decide (accept or reject the null hypothesis) between two or more hypotheses using the data obtained from a simulation scenario. From the established acceptance criteria, the scenario can be accepted or rejected against the actual values of a series.

The current analysis will provide a small sample for each metric, stricter assumptions must be imposed to give statistical validity to the test procedure:

<u>Small Sample Tests for a Population Mean</u>

To begin with, one common assumption is that the population from which the sample is taken has a normal probability distribution. Under such circumstances, if the population standard deviation is known, then the test statistic $(\bar{x} - \mu_0)/(\sigma/\sqrt{n})$ still has the standard normal distribution, as in the previous two sections. If σ is unknown and is approximated by the sample standard deviation s, then the resulting test statistic $(\bar{x} - \mu_0)/(s/\sqrt{n})$ follows Student's t-distribution with *n-1* degrees of freedom.

[1] Source: https://github.com/dspinellis/ckjm

The distribution of the second standardized test statistic (the one containing s) and the corresponding rejection region for each form of the alternative hypothesis (left-tailed, right-tailed, or two-tailed), is shown in Figure 16.

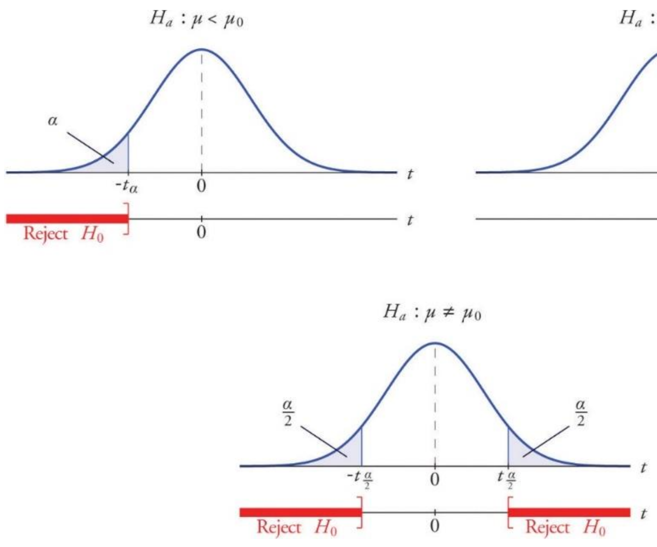


Figure 16 - Distribution of the standardized test statistic and the rejection region.[1]

The p-value of a test of hypotheses for which the test statistic has Student's t-distribution (Weisstein, n.d.) can be computed using statistical software. R[2] will be used for the statistical computing and graphics.

As the different web services technological stacks implementations are never executed together, data for each sample was gathered independently. Thus, the assumption of independence was not violated. The assumption of normal distribution will be tested using observation of normal probability plots, histograms with normal curve, and the combination of skewness and kurtosis coefficients.

Levene's test (Gastwirth, Gel, & Miao, 2009) for assumption that the variances of the groups are equal should be conducted to verify that assumption is not violated (i.e., $p > 0.05$) for all experimental groups.

Given the previous assertions, independent samples t-tests will be conducted for each web services implementations groups.

3.6.3 Web services QoS

In addition to analyzing the code quality, the assessment of some QoS parameters of the web services, given the context of this work, constitutes another crucial component in the evaluation of the general quality of the implemented solution. The following points summarize the metrics considered, the respective evaluation metrics and the hypothesis to judge.

[1] Source: https://saylordotorg.github.io/text_introductory-statistics/s12-04-small-sample-tests-for-a-popul.html
[2] Source: https://www.r-project.org/

3.6.3.1 Evaluation metrics

This section describe the QoS model used in this work, based on the work of (Aljazzaf, 2015) and (Kumari & Rath, 2015), selecting some attributes to evaluate in the context of this thesis. Table 13 summarize the relevant QoS, describing what each one represents.

Table 13 - Web services QoS evaluation metrics.[1]

Quality factors	Description
Latency ($QoS_{Latency}$)	Latency is the round-trip delay (RTD) between sending a request and receiving a response.
Execution time ($QoS_{Execution}$)	The execution time of a service is the time taken by the service to execute and process its sequence of activities.
Response time ($QoS_{Response}$)	The following is the formula to evaluate the response time: $QoS_{Response} = QoS_{Latency} + QoS_{Execution}$
Throughput ($QoS_{Throughput}$)	Throughput is defined either in terms of number of the requests or in terms of average data bytes per second: $QoS_{Throughput} = n^{o}\ of\ requests / time\ period$
Accessibility ($QoS_{Accessibility}$)	Accessibility refers to the service capability to serve the client's request: $QoS_{Accessibility} = n^{o}\ of\ successful\ user's\ requests / total\ user's\ requests$.

Additional metrics exists, related to the service availability: the probability that a service is up, present, and accessible to use: $QoS_{Availability} = Uptime / Total\ time$; and reliability, related to the previous one, being the the ability of a service to perform its function under the stated conditions correctly with either "no fail" or "response failure to the user" for a specific interval of time (Aljazzaf, 2015): $QoS_{reliability} = 1 - n / N + t$ where t denotes the total time a service is monitored for recording the number of failures, n, and N is the total number of events(number of successful events plus number of failures).

The previous metrics are out of scope of this work, since they required a monitorization process during a certain period, during which the services requests are executed, and the responses registered. This is highly influenced by the server machines where the web services are deployed, which in the context of the developed study is a variable out of the scope analysis.

3.6.3.2 Evaluation methods

There are a few open-source web service testing tools available in the software market. Although the core functions of these tools are similar, they differ in functionality, features, usability and interoperability. Keeping in sight the above-mentioned aspects, three representative web service testing tools were identified (Hussain, Wang, Toure, & Diop, 2013):

- JMeter[2] - is a 100% open source desktop application in Java designed to run functional tests and measure application performance. It was originally designed to test Web applications, but its use expanded to other test functions. The goal of the JMeter strategy is to provide more realistic testing scenarios. Therefore, load tests should simulate as close to reality as

[1] Source: (Aljazzaf, 2015; Kumari & Rath, 2015)
[2] http://jmeter.apache.org/index.html

possible, since realistic scenarios help minimize the effects of underestimation or overestimation of application response times;

- SoapUI[1] - is an open source Java tool whose main function is to consume and test Web Services. With this tool it is possible to perform functional, performance, load and safety tests. SoapUi uses web services for interaction and communication between different applications through an XML schema type called Web Service Description Language (WSDL);
- Storm[2] - is a free and open-source tool for testing web services. Storm is developed in F# and allows to test web services written in any technology.

Comparison of different testing tools is a complex task since testing tools may not comply with the same test criteria i.e. one tool may have the ability to test throughput (JMeter and soapUI), while another tool i.e. Storm, does not have this criterion. Furthermore, one tool may have better performance in one test case, while poorer in other test criteria: soapUI has better response time but throughput is not as good as JMeter's throughput. (Hussain et al., 2013)

3.6.3.3 Testing hypotheses

This comparative study was supported by the execution of a hypothesis test, which allows, with a certain degree of confidence, to see if there is a significant difference between the web services performance under analysis:

- H03: No significant difference exists between the web services performance of the generated code and the case study project.
- H13: The generated web services performance is relatively better that the one provided by the case study project.

3.6.3.4 Statistical analysis

In the comparison process, the first step consisted in determining which type of hypothesis test to perform, whether a parametric or non-parametric test. This decision is based primarily on the analysis of samples, namely: whether the variable is quantitative or qualitative; the number of samples under study; whether they are independent or matched; if its distribution can be assumed as normal and finally if there is homogeneity between the respective variances.

As both web services implementations are never executed together, data for each sample was gathered independently, moreover, both solutions would never be implemented together (populations are also independent). Thus, the assumption of independence was not violated. The assumption of normal distribution will be tested using observation of normal probability plots, histograms with normal curve, and the combination of skewness and kurtosis coefficients.

Levene's test for assumption that the variances of the two groups are equal should be conducted to verify that assumption is not violated (i.e., $p > 0.05$) for all experimental groups. Given the previous assertions, independent samples t-tests will be conducted for each web services groups.

[1] https://www.soapui.org/

[2] https://archive.codeplex.com/?p=storm

Chapter 4

DSL Design and implementation

The previous chapters touch the definition of two DSLs that will allow the development of web services models, presented either through the OAS or based in the resources definition. This chapter focus on the design and implementation of this domain languages, providing models examples, and describing the code generation process. An empirical comparison between the two DSLs definition is made, providing some insights related to their respective usability.

4.1 Common approach

The idea behind this approach is to specify an abstract model, which envelops the problem space and allows the developer to solve it. The model is defined in a domain specific language, which defines a syntax and a semantic for the domain. In case of the presented framework, the DSL is tailored to the OpenAPI specification and is constructed from a metamodel.

In the classic development process of a REST API, the developer must implement multiple classes - usually one per resource - each with similar source code. Every minor API design change could lead to multiple hours of work. The model-driven approach allows to automate this task by changing only the lightweight model. The new model supplies the developer with a new version of his API. In addition to the higher code quality the rate of reuse is extremely high as, for a new customer, the developers only specify a new model and adapt the generated outcome to the specific requirements.

Figure 17, based on (Cosentino, Tisi, & Izquierdo, 2015), gives an overview of the architectural approach followed in the web service MDE development process.

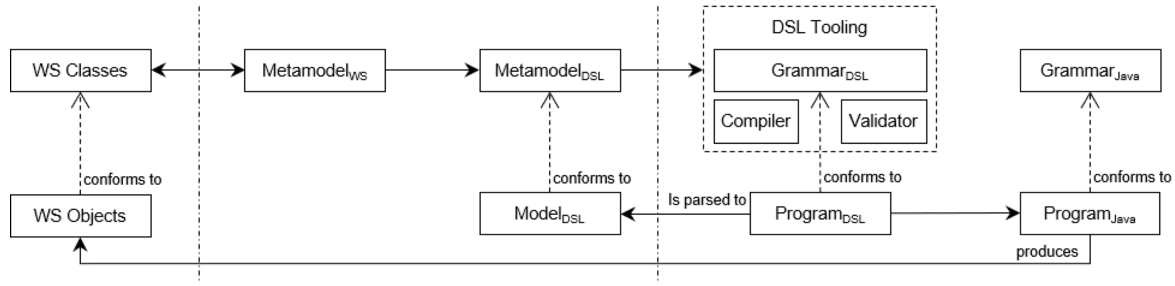


Figure 17 - Architectural overview of the approach followed.

Figure 17 spans through three technical spaces: the first one in which web service objects conforms to the set of defined web service classes and establishes itself as the foundation to develop the code-generation output; the second one where the metamodels associated with the web service and the developed DSL (OpenAPI based) are defined; and the last one where the grammar for the previous defined DSL, and the related code generation process is established, in conjunction with the base Java grammar foundations.

A modeling language raising the level of abstraction allows the reuse of models and keeps platform-specific artifacts at a separated tier in the development workflow. A modeling language with the ability to set aside technical concerns and still be able to tackle a problem in a specific platform is the main purpose of this work.

(Nguyen, Qafmolla, & Richta, 2014) defines a set of features that are essential to the DSL design in model-driven development of web services:

- **Effectiveness** the language must be able to provide a usable output without having the need of redefinition to adapt to a specific use case, while being easy to read and understand. This means that the language can bring good solutions to the domain for which it was designed and focus on solving the range of associated problems;
- **Automation and Agility**: as the modeling language can move the level of abstraction away from programming code through domain concepts, an important aspect is the ability to produce artifacts from these high-level specifications. This automatic transformation must match the requirements of the specific domain. Agility ensures that models can adapt to changes efficiently;
- **Support Integration**: the DSL needs to be able to integrate with other parts of the development process. This means that the language is used to edit, debug, compile and transform. The integration with other languages and platforms should also be effortless.

Given the formerly described approach, two DSLs were developed allowing the definition of a web service, following the OpenAPI specification on one side, and on the other following a custom resource definition, while building the foundations to a code-generation process.

4.2 Resource-oriented DSL

“A REST resource can be any accessible information, which includes values from algorithmic computations, virtual objects, binary files, texts and many more” (Schreibmann & Braun, 2014).

A resource is nothing more than an abstraction about a certain type of information that an application manages (it cannot be used *per se*), that can be accessed through a uniform interface, and may have different representations, based on known specifications. Every resource must have a unique identification, for the application to be able to differentiate which of the resources must be handled in a given request.

In the scope of this work, to reach the resource, the server specifies an URI following a template[1], which should contain only information necessary to access the resource while omitting any meta information. Additionally, the resource representation can be based on different restrictions: XML

[1] A minimal URI template would contain the base and the resource identification: http://baseuri.com/resourcename

and JSON are the most common ways to represent resources to different users; whichever representation is chosen, to follow a RESTful paradigm it must support links usage.

An efficient RESTful API, cohesively designed, has well-defined resources and structured relationships, that ease its usage, shortening the necessary time for getting accustomed with the API.

Succeeding the previous paragraphs, it can be stated that the resource is the most relevant concept in a web service development, and their correct identification is the most pertinent exercise in the design process. A model-driven process based on the resource identification can be then considered a natural approach when implementing such paradigm in web services development, providing to the web service designer full flexibility to model resources according to the application domain and use cases.

The following sections detail the methodology used in this work, from the metamodel definition, to the grammar implementation, to an example of a model build over the metamodel using the developed grammar.

4.2.1 Resource-oriented metamodel

While the resource representation establishes itself as the main component in the approach depicted in this section, the model must also be demonstrative of the additional and relevant information regarding the web service structure and behavior.

Figure 18 depicts a resource-based web service metamodel, referencing the main components and how they correlate to each other's. It was built based on the main concepts/constraints of the RESTful architectural style stated in section 2.3.3, revolving around the resource representation. It serves as abstract syntax of the DSL developed in Xtext and underpins the code generator written in Xtend. The metamodel conceptualization takes advantage of UML features such as inheritance, composition and aggregation, to add further meaning to the relationships between the portrayed components.

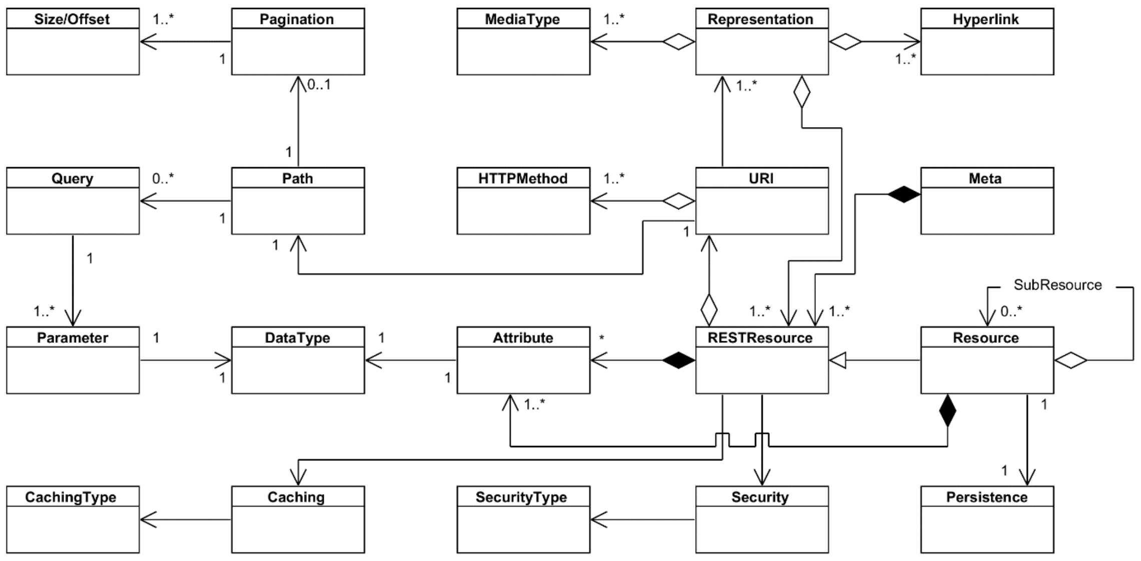


Figure 18 - Resource-oriented grammar metamodel.

To achieve a feasible representation of a RESTful web service, multiple concepts and their relationships were defined, and can be divided in three complementary areas:

1. Base information:
   - **Meta** - this component establishes some base information common to all resources, like the base path of the web service, the version of the API and the default media type.
2. Resources:
   - **REST resource** - the core concept, where all the others converge. As the name states it represents the REST resource;
   - **Resource** - while the previous concept integrates all the constrains inherent to the RESTful architecture, this resource comprehends the attributes that define the business object and is deeply coupled to the REST resource;
   - **Sub-resource** - the resource can have an attribute of the type sub-resource;
   - **Attribute** - it represents a specific property of the resource;
   - **Data type** - the attribute data type;
   - **Persistence** - contains generic information regarding the persistence of the resource, like the database table name associated, mapping between attributes and columns names, etc.
3. RESTful constrains:
   - **URI** - URI associated with the resource;
   - **HTTP method** - the supported HTTP methods over the resource;
   - **Representation** - for a specific URI the representation of the resource, associated with supported media-types;
   - **Media type** - supported media type by the resource, being the most commons JSON and/or XML;
   - **Path** - the resource path, complementing the URI information;
   - **Query** - associated with the URI and Path, a query over the resource or collection of resources can complement the request to filter over the available resources;
   - **Pagination** - allows the definition of a paging mechanism to allow clients to fetch only a limited number of resources in a single request;
   - **Caching** - the caching mechanism that the web service must support;
   - **Security** - the security mechanism that the web service must support;

   Additionally, the application states can be inferred from a combination of the resource and the HTTP verb, and represents one valid REST request, which uses a URI to access one resource

The author understanding of the REST architectural style and constrains was included into the presented meta-model, which covers all relevant aspects necessary to model a RESTful API. The following section renders the grammar development steps, supported on the previous defined model.

4.2.2 Resource-oriented grammar

The metamodel presented in Figure 18 intents to define with a high level of abstraction the main components of a web service. In the scope of this work, to focus on the functional side of web services, some concepts were left out of the grammar development, namely the security mechanism implementation, since it is one of the most complex undertakings in the REST field, and it is possible, with relative ease, to delegate this responsibility to external systems (Keycloak[1], i.e.) focused in authentication and authorization matters; and the queries custom definition.

Subsequent sections follow the same structure division of the metamodel, identifying the grammar part associated with the base information, resource definition and RESTful constraints.

4.2.2.1 Grammar structure and Meta information

Code snippet 22 shows the grammar main components, given a perspective on how the main concepts of the metamodel where implemented. Most of the components identified in the metamodel have a direct correspondence with parser rules from the grammar: Meta information; Resource; Attribute and Data Type.

Code snippet 22 - Grammar structure.

```
Model: (…)
Meta: (…)
ServerInfo: (…)
Resource: (…)
Attribute: (…)
Composite: (…)
Relation: (…)
Reference: (…)
Action: (…)
Trigger: (…)
DataType: (…)
MediaType: (…)
// auxiliary rules
```

The Meta parser rules identify the meta information of the web service, allowing the identification of common information: the project name, version, URI base path, server information; media type supported by default and the base package, used in multiple programming languages to build the project structure - Code snippet 23. Given its simplicity, when require, it can be easily upgraded with additional rules.

Code snippet 23 - Grammar meta information rules.

```
Meta:
  'project:' project=STRING
  'version:' version=STRING
  'basePath:' basePath=Url
  'server:' server=ServerInfo
  ('mediaType:' mediaType+=MediaType+)?
  ('basePackage:' package=QualifiedNameWithWildcard)?;

ServerInfo:
  '{'
  'description:' description=STRING
  'url:' url=Url
  '}';

MediaType:
```

[1] https://www.keycloak.org/

```
  JSON | XML;

Type:
  RsrcString | Integer | Long | BigDecimal | Calendar | Boolean | EnumType;

JSON:
  {JSON} 'Json';

XML:
  {XML} 'XML';

QualifiedName:
  ID ('.' ID)*;

QualifiedNameWithWildcard:
  QualifiedName '.*'?;
```

To aid in the grammar usage, validation rules over the Url rule where implemented through regular expressions in the DSL validator Xtend class, only allowing valid URLs to be introduced in this fields.

4.2.2.2 Resources and RESTful constraints

The Resource and RESTful constraints parser rules are presented in Code snippet 24. Resource parser rule, in its definition, unites multiple sub-rules that:

- Establish the respective inner properties - Attribute rule;
- Defines inner objects as a property of the main Resource - Composite rule;
- Outlines the relationships with other defined resources - Relation rule;
- References the parent Resource in a relationship - Reference rule;
- Allow the definition of custom actions over a Resource - Action rule;
- References the actions that actuate over the Resource - Trigger rule;
- State the supported HTTP methods over the Resource - HTTPMethod rule.

Code snippet 24 - Grammar resource rules.

```
Resource:
  'resource' (abstract?='abstract')? name=ID ('table' table=ID)? (cache?='cache')? (extends?=Parent)?
  '{'
  (attributes+=Attribute)*
  (contains+=Composite)*
  (relations+=Relation)*
  (references+=Reference)*
  (customActions+=Action)*
  (actuatedBy+=Trigger)*
  (httpMethods+=HTTPMethod)*
  '}';

Parent:
  'extends' parent=[Resource];

Composite:
  'contains' name=ValidID type=[Resource] (multiple?='*')?;

Reference:
  'reference' name=[Relation|QualifiedName] ('column' column=ID)?;

Relation:
  'relation' name=ValidID type=[Resource|QualifiedName] (multiple?='*')?
  ('method' '[' actions+=HTTPMethod+ ']')?;

Attribute:
```

Code snippet 24 - Grammar resource rules.

```
  'attribute' name=ValidID type=DataType (multiple?='*')? ('column' column=ID)?
  (mandatory?='mandatory')?;

Action:
  'action' name=ID 'on' resource=[Resource] 'over' attribute=[Attribute|QualifiedName]
  ('method' '[' actions+=HTTPMethod+ ']');

Trigger:
  'actuated' 'by' name=[Action|QualifiedName];
```

Analyzing each of the rules individually:

- **Resource** - rule that defines a Resource, its name; database name if necessary, and if it is supposed to be a cached resource;
- **Attribute** - rule to identify the Resource's properties. It allows the definition of the attribute name, data type, database column and if it is mandatory;
- **Composite** - defines if the main Resource contains a reference to other one, but this last resource isn't accessible as a relationship of the first (through a specific URI);
- **Relation** - if a Resource has a relationship with other one, that can be expressed through an URI, this rule defines the relation name, type, cardinality and the supported HTTP methods through the relationship;
- **Reference** - this rule is directly related to the Relation one, setting if a Resource has a parent resource, defining a specific name for it and a database column name;
- **Action** - defines custom actions that will impact an attribute of a specific Resource, being the current Resource the starting point;
- **Trigger** - references the Resource and the Action that can change some attribute of the current Resource;
- **HTTP Method** - simple parser rule that enumerates the supported HTTP verbs.

The main purposes of the Reference and Trigger rules is to enable access, in the child resource, to the parent information, that may be necessary in the code generation process.

4.2.2.3 Grammar outline

The presented grammar was built with focus on the web service resources definition and their relationships, establishing a supporting base to a code generation process. Given the high-level of abstraction provided by the grammar, code artifacts can be easily inferred, and adapted, through the exposed concepts and implemented using any preferred programming language (assuming its suitability to web development environments).

Cohesive and comprehensive rules were implemented in the grammar, to enable its easy comprehension and usage, decreasing the learning curve required to use it efficiently in the web service definition processes. Also, some validations were integrated in the grammar, improving its usability while a domain specific language ensuring a solid development environment, compliant with REST architectural style concepts.

As a DSL, one the main purposes of this grammar is to establish a comparison base with the OAS-based one, highlighting its advantages or disadvantages in relation to the other, providing some insights on how it can benefits the web service development process. The full grammar can be seen in the Appendix Appendix B.

4.2.3 Model example

This section provides an example of a model build over the developed grammar, providing an understating on how it can be used as a web service design base, and subsequently achieve the concrete implementation through an automated code generation process.

Figure 19 illustrates a simplified domain model of a nutrition clinic, showing the main business concepts and their relationships.

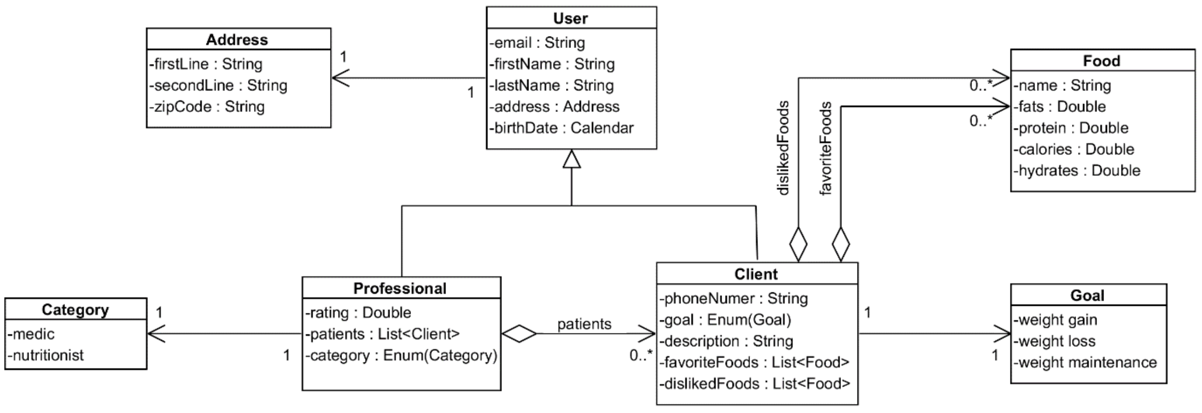


Figure 19 - Domain model example.

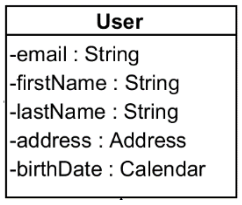
The focus of this model are the relations between the Clients, Professionals and the nutritional info of Foods, establishing an acting base over the preferences of the Client to achieve the intended Goal.

An inheritance relation is evident, where the Professional and Client concepts, inherit some attributes from the User artifact.

Additional business components are identified, namely in the form of enumerates: Category enumerates the available professional categories and Goal the existing possible objectives.

Leveraging on the presented domain model, the following figures reveal a possible model built over the grammar previously exposed, focused on the identified business concepts, and their relationships. The model builds on the assumption that, in the design process, the resources and sub-resources were identified based on the expected behavior of the platform that will use the web services.

Figure 20 illustrates how the business concepts can be represented through the developed grammar, including their attributes, relationships, allowed methods and persistence meta information (table and column names).

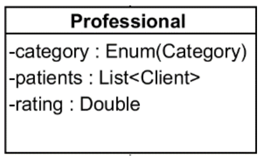


```
resource abstract User table USER {
  attribute firstName String column First_Name
  attribute lastName String column Last_Name
  attribute email String column Email
  attribute birthDate Calendar column Birth_Date

  contains address Address

  NONE
}
```

a) User resource definition.



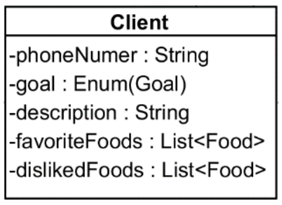
```
resource Professional table PROFESSIONAL extends User {
  attribute category Enum(Category) column Category
  attribute rating Integer column Rating

  relation patients Client* actions [GET POST DELETE]

  actuated by Client.rate

  POST PUT
}
```

b) Professional resource definition.



```
resource Client table CLIENT extends User {
  attribute phoneNumber String column Phone_Number
  attribute goal Enum(Goal) column Goal

  relation favoriteFoods Food* actions [GET POST DELETE]
  relation dislikedFoods Food* actions [GET POST DELETE]

  reference Professional.patients

  action rate on Professional over Professional.rating method [POST PUT]
}
```

c) Client resource definition.

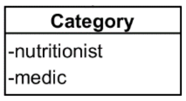
Figure 20 - Resource-oriented model: User related resources definition.

Some of the resources relations can be derived directly from the domain model, while others may need additional specifications. In this concrete model, the action "rate", available to the resource Client, that acts over the Professional attribute "rating", isn't explicitly defined in the domain model, but in a real-world situation may have been defined in the requirements analysis process - create or update the rating given from one Client to a specific Professional.

Presented in Figure 21, as an example, are the definitions of the additional relevant components that cover the remaining available rules provided by the developed grammar: the meta information and enumerates rule.

```
project: "Nutrition Clinic"
version: "1.0"
basePath: "nutrition.clinic"
server: {
  description: "Nutrition clinic development server"
  url: "http://nutrition.clinic"
}
mediaType: Json XML
basePackage: com.nutrition.clinic
```

a) Meta information definition.



```
enum Category {
  name Medic
  value 1

  name Nutritionist
  value 2
}
```

b) Enums definition.

Figure 21 - Resource-oriented model: Additional components.

The grammar usage has some intrinsic predefined behavior that need to be considered when using it as starting point to a code generation process:

- the URI paths definition isn't explicit, it must be derived from the resources definition and their relationships;
- the default functional behavior of the HTTP verbs cannot be influenced, which means that a POST will always try to create a new entity, i.e.;
- the relation rule establishes a relationship through an URI template similar to:

  `{resource}/{id}/{relatedResource}`

- the custom action establishes an action through an URI template like:

  `{resource}/{resourceId}/{targetResource}/{targetResourceId}/{action}/{value}`

- if no HTTP method is provided the most common ones are made available by default: POST, GET, PUT and DELETE.

The full model example is available in Appendix Appendix C.

4.3 OAS-oriented DSL

The main challenge in this approach is how to capture the behavior, and implementation details of a typical web service, and how these can be derived from the respective specification in the OpenAPI format. This last one requiring by itself a specific parser to interpret the multiple components that structure the specification.

This section outlines the OAS gammar development, taking in consideration that that process is heavily conditioned by the requirement of maintaining the OAS structure and content as the language in which the associated models are built.

4.3.1 OAS-oriented metamodel

Figure 22 depicted meta-model uses UML as meta meta-model, providing a better understanding of the multiple OpenAPI components and allows the inclusion of specific features into the modeling phase, giving additional meaning to the relationships between components.

The meta-model aids in finding an appropriate representation by exploring the boundaries and the core of the domain. It is derived from the concepts and properties described in the OpenAPI specification document and the use of UML and its artifacts allows a better understanding of the relationships between the multiple OpenAPI objects, providing an unambiguous and broad range of the assumed specification interpretation.

(Nguyen et al., 2014) suggests a division of the showed concepts in three main parts:

- **Behavioral elements**: in this category, the “Paths”, “OperationObject” and “ResponsesObject” objects assume the most relevant role in defining and interpreting the overall API behavior, mainly the exposed services and how it is supposed to answer to external calls;
- **Structural elements**: the main component that define how each OpenAPI object is built, describing the data types and available data structures, is the “Schema” object, which is used in multiple objects, namely in the “Components” one, allowing to define structures based on the JSON Schema Specification, with some OpenAPI specific features;
- **Serialization/deserialization elements**: in this group are included the elements that support the serialization and deserialization of OpenAPI models in JSON or YAML formats, namely the “Paths”, “OperationObject” and “SchemaObject” concepts.

The highlighted OpenAPI objects will be the core of the code generation process and were the focus of the grammar development. This decision, of bordering the supported objects by the grammar, resulted from the vast number of objects that the specification contains, bringing the need of focusing in the most relevant concepts in the web service definition, and adjust the implementation process to the time frame available for this work resolution. This decision is also supported by the previous defined Resource-based grammar, where the concepts that integrate it find an equivalent on the elected OpenAPI objects - Table 14.

Table 14 - Relation between the Resource and OAS oriented concepts.

Resource-oriented DSL concepts	**OAS-oriented DSL objects**
Resource	Component; Schema
Attributes	Schema properties
Relation	Paths; Operation
Contains	Schema; Schema properties
Action	Paths; Operation
HTTP method	Operation

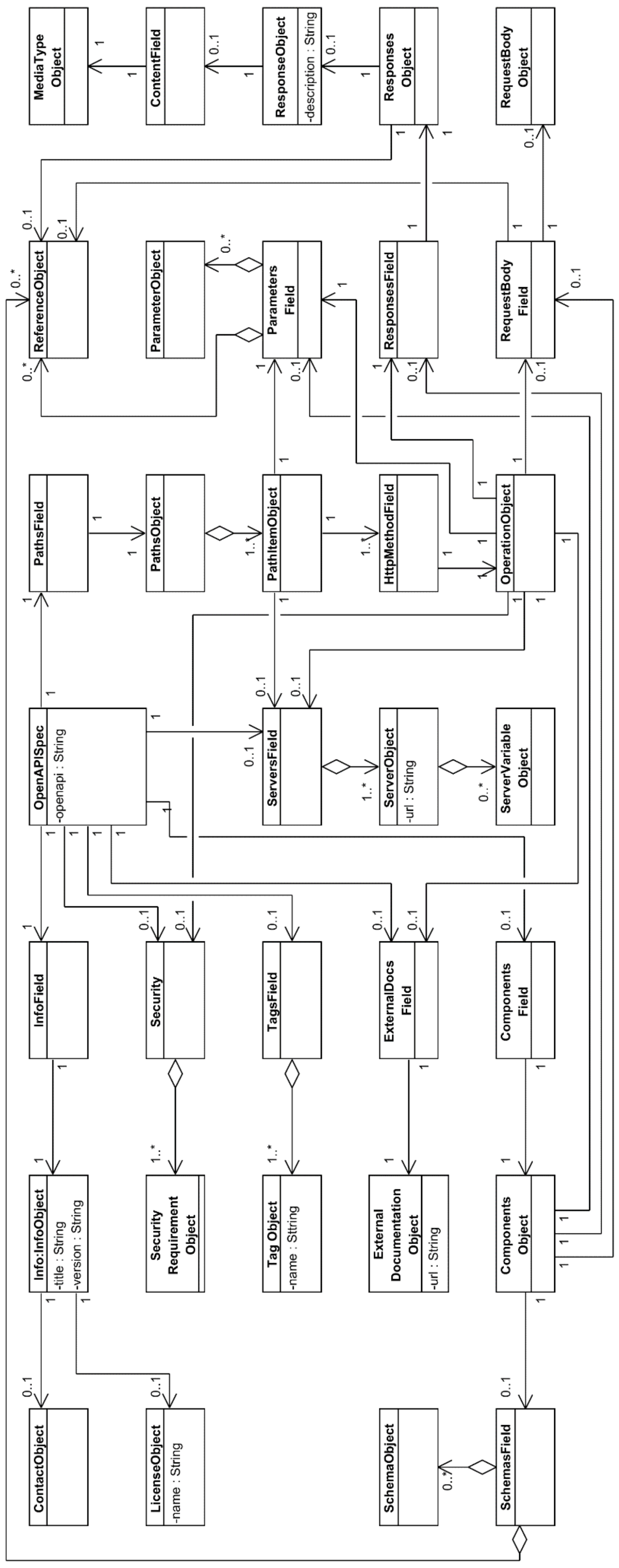


Figure 22 - OAS-oriented grammar metamodel.

Given the formerly described metamodel and associated categories, a DSL was developed allowing the definition of a web service, following the OpenAPI specification, while building the foundation to a code-generation process.

4.3.2 OAS-oriented grammar

Another challenge risen while developing the OAS based DSL was maintaining the structure of the original specification document, while avoiding the introduction of ambiguities in the grammar.

A document conforming to the OpenAPI specification is itself a JSON object, that can be represented either in JSON or YAML format: while the YAML format has a friendlier structure for a human reader by resorting to indentation to delimit the different components of the specification, the JSON version eases the interpretation of this limits programmatically, with use of specific characters - curly brackets. This factor, and the existence of multiple tools that allow the conversion between the two formats, defined that the developed DSL would interpret the JSON format only.

The Code snippet 25 provides a glimpse of how the grammar is implemented, and how it interprets an OpenAPI specification document: the main structure of the specification is parsed by the "OpenAPIObject", where the seven major components (see Figure 8) that constitute the OpenAPI specification document are clearly demarcated; each one of the components delegate the parsing of their content in inner objects with the required logic.

Code snippet 25 - Grammar excerpt defining the OAS structure.

```
grammar com.tmdei.xtext.dsl.OasDsl with org.eclipse.xtext.common.Terminals

generate oasDsl "http://www.tmdei.com/xtext/dsl/OasDsl"

OpenAPIObject:
  '{'
  documentOpenAPIVersion=OpenAPIVersionField
  infoField=InfoField
  (serversField=ServersField)?
  pathsField=PathsField
  (componentsField=ComponentsField)?
  (securityField=SecurityField)?
  (tagsField=TagsField)?
  '}';

OpenAPIVersionField:
  '"openapi":' openApi=STRING ',';

InfoField:
  '"info":' info=InfoObject;

InfoObject:
  '{'
  title=TitleField
  (description=DescriptionField)?
  (termsOfService=TermsOfServiceField)?
  (contact=ContactField)?
  (license=LicenseField)?
  (version=VersionField)?
  ('}' | '},');
```

The following code-snippet is responsible for parsing the "PathsObject" and "PathItemObject" and provides one additional example of how the objects are defined in the Xtext grammar.

Code snippet 26 - Grammar excerpt to parse the PathsObject.

```
PathsObject:
  url=Url ':' '{'
  paths+=PathItemObject+
  ('}' | '},');.

PathItemObject:
  (=> httpMethod+=HttpMethod ':' '{' operation+=OperationObject ('}' | '},'))+
  ('"$ref":' '{' ref=PathItemObject ('}' | '},'))?
  (summary=SummaryField)?
  (description=DescriptionField)?
  (serversField=ServersField)?
  (parameters=ParametersField)?;

Url:
  url=STRING;

HttpMethod:
  httpMethod=STRING;
```

In the context of this objects, two parser rules, “HttpMethod” and “Url”, can be identified and at first sight it seems that any string value can be assigned to them. However, semantically, the “HttpMethod” can only be assigned with a limited number of values (get, put, post, delete, options, head, patch and trace), and the “Url” value must agree with a specific format (e.g., `/user/{id}/address`).

The grammar should then validate the values that can be assigned to this parser rules. This was achieved through the implementation of validators, like is is displayed in the Code snippet 27.

Code snippet 27 - Custom grammar validators.

```
class OasDslValidator extends AbstractOasDslValidator {
  static val INVALID_HTTP_METHOD = "invalidHttpMethod";
  static val INVALID_PARAMETER_LOCATION = "invalidParameterLocation";

  val validHttpMethods = "get|put|post|delete|options|head|patch|trace";
  val validParameterLocations = "query|header|path|cookie|body";
  @C
  @Check
  def checkValidHttpMethod(HttpMethod httpMethod) {
    if (!httpMethod.getHttpMethod().matches(validHttpMethods)) {
      error('Invalid HTTP method!', OasDslPackage.Literals.HTTP_METHOD__HTTP_METHOD,
        OasDslValidator.INVALID_HTTP_METHOD);
    };
  }

  @Check
  def checkValidParameterLocation(ParameterObject parameterObject) {
    if (!parameterObject.in.in.matches(validParameterLocations)) {
      error('Invalid parameter location! \n' + 'Valid locations: ' +
             validParameterLocations.toString.formatValuesList,
             OasDslPackage.Literals.PARAMETER_OBJECT__IN, OasDslValidator.INVALID_PARAMETER_LOCATION);
    };
  }
}
```

This approach allows to establish a first layer of validations when using the DSL to define an OpenAPI specification document, aiding the code generation process, by releasing it from this additional processing requirement. Writing wrong resource specification without following the rules mentioned above, and others implemented to conform with the OpenAPI, will result in a project run-time error which will not generate the code in the target project.

4.3.3 Model example

Based on the domain model presented previously - Figure 19 - an OAS model was built with the developed grammar.

Code snippet 28 presents the model overall structure, and it can be seen that it perfectly mimics the OpenAPI specification document.

Code snippet 28 - Model excerpt defining the OAS structure.

```
{
  "openapi": "3.0.1",
  "info": {},
  "servers": [],
  "paths": {
    "/clients": {},
    "/clients/{clientId}": {},
    "/clients/{clientId}/favoriteFoods": {},
    "/clients/{clientId}/dislikedFoods": {},
    "/clients/{clientId}/professionals/{professionalId}/rate/{rating}": {},
    "/professionals": {},
    "/professionals/{professionalId}/clients/{clientId}": {},
    "/foods": {},
    "/foods/{foodId}": {}
  },
  "components": {
    "schemas": {
      "User": {},
      "Client": {},
      "Professional": {},
      "Food": {},
      "Address": {},
      "Clients": {},
      "Foods": {}
    }
  }
}
```

It shows the services that should be exposed in the web service for each of the resources that integrate the business domain and the HTTP verbs that they should support.

Code snippet 29 - Model excerpt defining a service over Client sub resource.

```
{
  "/clients/{clientId}": {
    "get": {
      "tags": [
        "users"
      ],
      "summary": "Info for a specific Client",
      "operationId": "showClientById",
      "parameters": [{…}],
      "responses": {
        "default": {
          "description": "Unexpected error",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/Error"
              }
            }
          }
        },
        "200": {…}
      }
    }
  }
}
```

```
"parameters": [{
    "name": "clientId",
    "in": "path",
    "description": "Client id to retrieve",
    "required": true,
    "schema": {
      "type": "string"
    }
  }
]

"responses": {
  "default": {},
  "200": {
    "description": "Response to a valid request",
    "content": {
      "application/json": {
        "schema": {
          "$ref": "#/components/schemas/Client"
        }
      }
    }
  }
}
```

The previous code snippet illustrates the detail of a Path object, where additional components are used do define: the service description, required parameters and the expected and possible responses. It can be stated, that a successful resource leads to a response with a reference to the component Client.

Code snippet 30 shows a model excerpt where three components are defined: User, Address and Client.

Code snippet 30 - Model excerpt defining the components schema.

```
"User": {
  "required": ["id", "email”],
  "properties": {
    "id": {
      "type": "integer",
      "format": "int64"
    },
    "firstName": {
      "type": "string"
    },
    "lastName": {
      "type": "string"
    },
    "email": {
      "type": "string"
    },
    "birthDate": {
      "type": "date"
    }
    "address": {
      "type": "object",
      "items": {
        "$ref": "#/components/schemas/Address"
      }
    },
  }
},
"Address": {
  "required": ["zipCode","firstLine"],
  "properties": {
    "firstLine": {
      "type": "string"
    },
    "secondLine": {
      "type": "string"
    },
    "zipCode": {
      "type": "string"
    }
  }
}
```

```
"Client": {
  "properties": {
    "allOf": [
      {
        "$ref": "#/components/schemas/User"
      }, {
        "type": "object",
        "properties": {
          "phoneNumber": {
            "type": "string"
          },
          "description": {
            "type": "string"
          },
          "favoriteFoods": {
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/Food"
            }
          },
          "dislikedFoods": {
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/Food"
            }
          },
          "goal": {
            "type": "string",
            "enum": [
              "gain weight",
              "lose weight",
              "maintain weight"
            ]
          }
        }
      }
    ]
  }
}
```

As it is stated in the domain model, an inheritance relationship exists between the User and Client entities, where the last one inherits some parameters from the first. In terms of OAS this relation translates to a component field named “allOf”, that states a reference to the parent component, and then integrates the specific properties of the child object.

While in this section the most relevant OAS components to the web service definition were presented, the full model specification can be seen in Appendix Appendix E.

4.4 Code generation

The code generation process is supported by a working reference implementation (detailed in the next sub section), developed around the domain model presented in Figure 19, that contemplates the application of adequate design patterns and principles. In the end of this process the application should be, structural and content wide, like the reference implementation.

Given the substantial differences between the two developed grammar the code generation process related to each is structured accordingly, but with the same underlying philosophy, focused on generating the code artifacts associated with the resources in a first phase, and then generate the gateways, exposing the identified web services.

The software generator gets a model as input and creates all necessary artifacts for a complete backend that contains the API, the business logic, and the source code of the persistence layer. This last component, in the concrete case of this work, is tightly coupled to a JPA implementation making use of the associated API in the persistence layer definition and behavior

4.4.1 Supporting implementation

Figure 23 gives an overview of most of the components that are generated in this process, while providing additional information regarding the web service architecture adopted for the reference implementation.

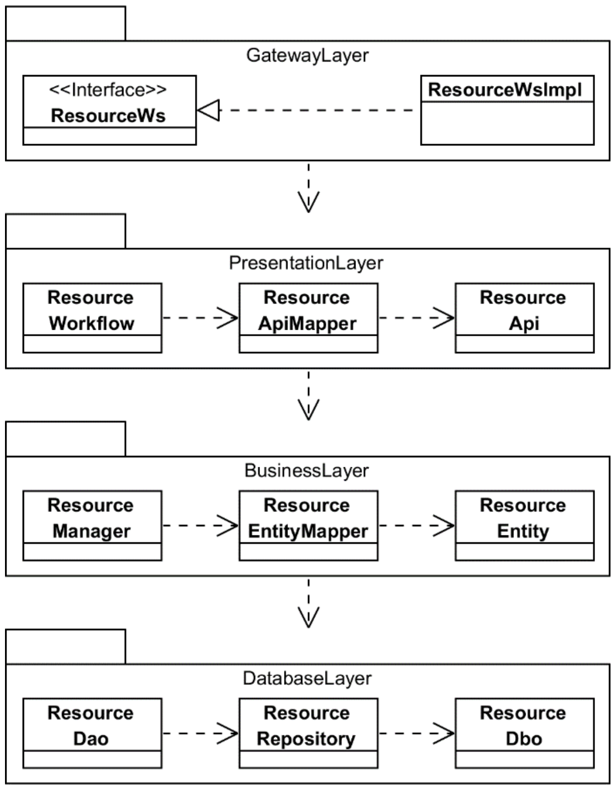


Figure 23 - Web service layered architecture.

A four-layer architecture was considered:

- **Database layer**: It is responsible for managing the database:

  - ResourceDao: data access object that encapsulates the database operations (save, update delete and retrieve) over the database objects;
  - ResourceRepository: encapsulates and manages the access to the database;
  - ResourceDbo: entity representative of the database table.
- **Business layer**: In this layer the business models are defined, with all associated operations:
  - The ResourceManager manages the operations in the business context;
  - The ResourceEntityMaper maps ResourceEntity to ResourceDbo (database layer) and vice-versa;
  - The ResourceEntity is a business entity, defining the main attributes relevant to the business rules implementation.
- **Presentation layer**: This intermediate layer between the presentation layer and the layer with the business logic that delegates the requests from the first to the second:
  - ResourceWorkflow: It is a class responsible for controlling the flow of the service, calling the managers of the lower layer, after mapping the ResourseApi entities to ResourceEntity entities;
  - ResourceApiMapper: It maps ResourceApi to ResourceEntity entities and vice-versa (business layer);
  - ResourceApi: This entity represents the JSON or XML entities accepted in the services.
- **Gateway layer**: It exposes the web services endpoints and delegates the received requests to the subsequent layer:
  - ResourceWs: It is an interface with the exposed web service endpoint;
  - ResourceWsImpl: It is an implementation of the resource web service interface.

Furthermore, and to provide a better understanding of the application flow, the sequence diagrams associated to each of the CRUD operations are presented in the next figures.

Figure 24 shows the create operation flow, where the communication between each layer is clear, and the intervention of each artifact exposed previously is evident.

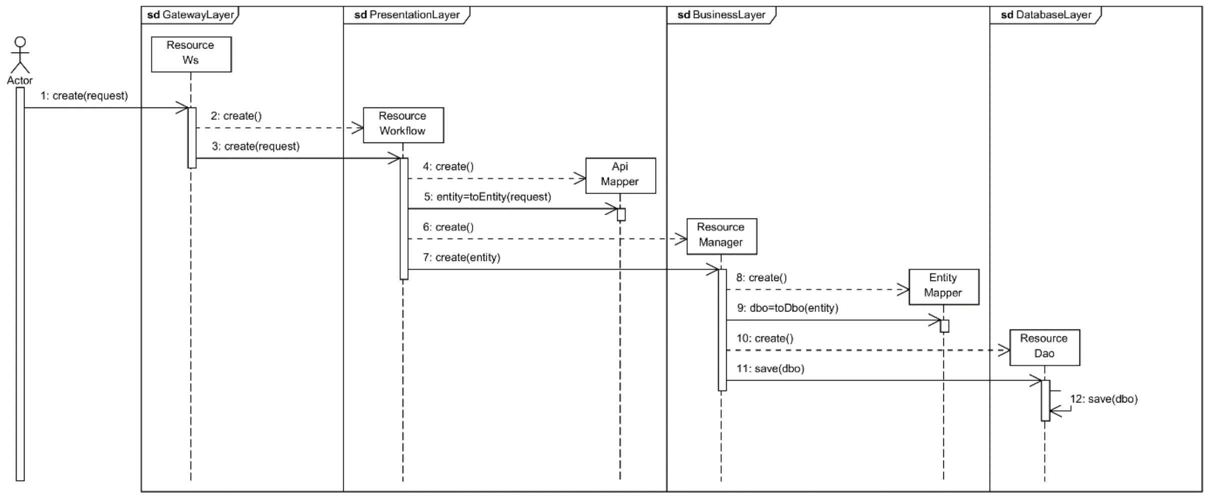


Figure 24 - CRUD: Create operation.

The flow to the search operation is patent in Figure 25.

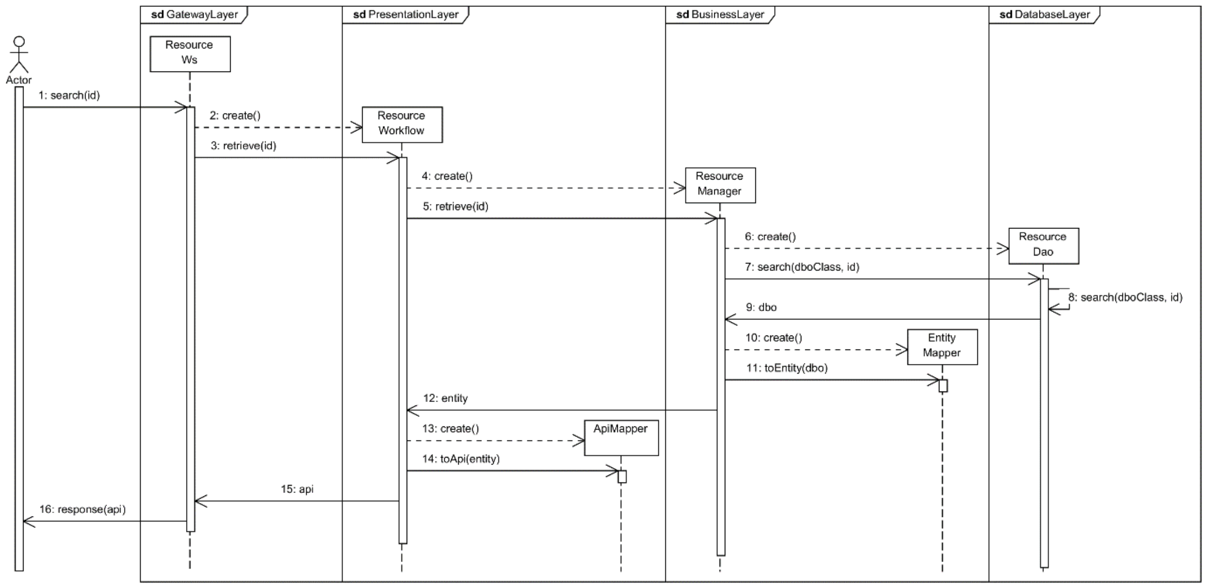


Figure 25 - CRUD: Search operation.

Each one of the previous described components is generated associated it the respective resource in the code generated process, detailed in the next sections.

4.4.2 Common implementation considerations

As it was already stated the implementation of web services leads to code repetition across the multiple resources involved, since the basic CRUD operations always follow the same flow. Taking this in consideration and to avoid the generation of repetitive code for each identified resource, some base implementations were introduced in the generation process, from which the new code artifacts can extend and access the common methods. Following this train of thought, Figure 26 illustrates how the base implementation integrates with the generated code.

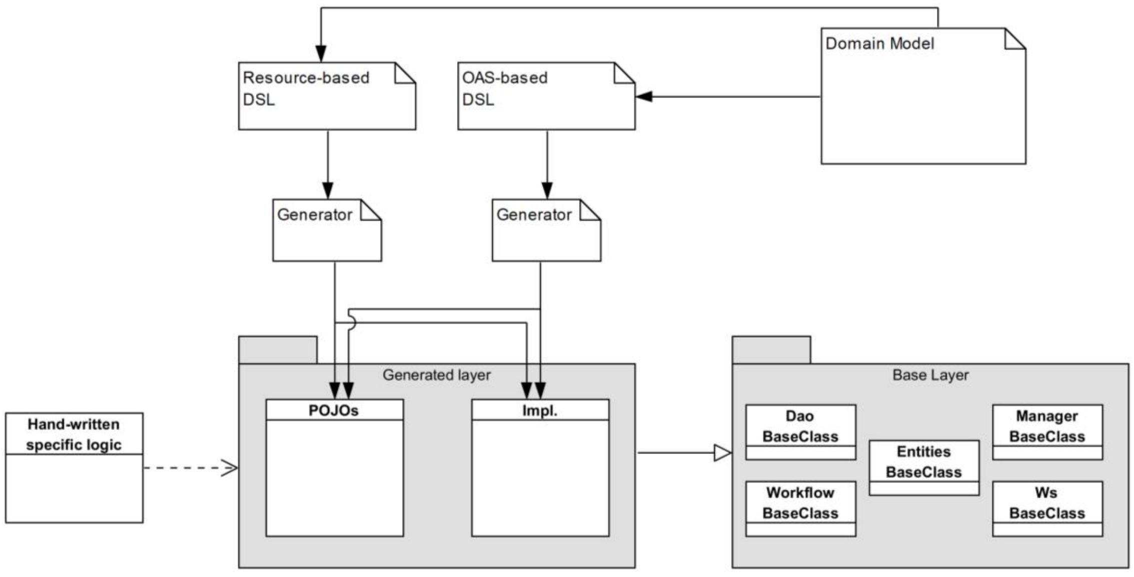


Figure 26 - Integration between the generated code and the base implementation.

Two layers were considered, one with the base artifacts, and other with the generated code. From the domain model, the developer chose which DSL to use to model the domain, and the generator will create the code artifacts associated: plain old Java objects (POJOs) and the implementations that will extend the base classes with the required methods overridden.

Another component common to both code generation processes are the Xpect tests, that work like the commonly used unit tests, that will ensure the correctness of each generated code artifact.

These common implementation details are subject of the subsequent points.

4.4.2.1 Base project

Following the architecture displayed in Figure 23, for each layer the most relevant base implementations are:

- **Database layer**
  - BaseDao, provides the base implementation for the persistence methods
    - `public D save(D dbo) {…}`
    - `public D retrieve(Class<D> c, Long id) {…}`
    - `public List<D> retrieve(QueryParameters queryString) {…}`
    - `public void delete(D dbo) {…}`
- **Business layer**
  - BaseManager
    - `public E create(E entity) {…}`
    - `public E retrieve(long id) {…}`
    - `public List<E> retrieve(QueryParameters query) {…}`
    - `public E update(E entity, long id) {…}`
    - `public E delete(long id) {…}`
- **Presentation layer**
  - BaseWorkflow
    - `public A create(A api) {…}`
    - `public A retrieve(long id) {…}`
    - `public List<A> retrieve(QueryParameters query) {…}`
    - `public A update(A api, long id) {…}`
    - `public A delete(long id) {…}`

The presence of this base implementations promotes code reusability, while improving the overall maintenance ease of the application. Each one of these methods have its own implementation, but, when needed, can be manually overridden to accommodate additional logic depending on the defined business requirements.

To ensure RESTful compliance with the constraints related to filtering, sorting and pagination of the resources, an external library[1] was used, removing additional complexity from the code

[1] https://github.com/kumuluz/kumuluzee-rest

generation process by delegating these complex operations in this dependency. This library enables each one of the referred features through query parameters usage:

- **Pagination**
  - `GET /api/foods?offset=10`
  - `GET /api/foods?limit=5`
  - `GET /api/clients?offset=10&limit=5`
- **Sorting**
  - `GET /api/foods?order=id DESC`
  - `GET /api/foods?order=calories ASC`
  - `GET /api/foods?order=calories ASC,proteins DESC`
- **Filtering**
  - `GET /api/foods?filter=id:EQ:1`
  - `GET /api/foods?filter=name:NEQIC:'doe'`
  - `GET /api/foods?filter=name:LIKE:H%`
  - `GET /api/clients?filter=address.zipCode:LIKE:4425%`

HATEOAS, specifically the links component, was also implemented in the base application through some protected methods (Code snippet 31), that must be overridden by the concrete classes if the default implementation does not suffice or if the associated relation (rel) does not exist.

Code snippet 31 - Default links methods available in the base implementation.

```
protected ResourceLink getLinkSelf(…);
protected ResourceLink getLinkUpdate(…);
protected ResourceLink getLinkDelete(…);
protected ResourceLink getLinkRelationships(…);
```

Code snippet 32, illustrates the response associated with the default implementation, for a POST request.

Code snippet 32 - Response example from a request to create a Client.

```
{
  "id": 1,
  "links": [{
      "href": "http://localhost:8080/nutrition.clinic/api/clients/1",
      "rel": "self"
    }, {
      "href": "http://localhost:8080/nutrition.clinic/api/clients /1",
      "rel": "update"
    }, {
      "href": "http://localhost:8080/nutrition.clinic/api/clients/1/favoritefoods",
      "rel": "foods"
    }, {
      "href": "http://localhost:8080/nutrition.clinic/api/clients/1/dislikedfoods",
      "rel": "foods"
    }
  ],
  "address": {…},
  "email": "test_01@mail.com",
  …
}
```

4.4.2.2 Unit testing

To improve the code generation process reliability, it is fundamental to create tests that will confront the generated code with the expected one. Following this train of thought a framework focused on this matter, Xpect[1], was integrated in each one of the Xtext's projects.

The test creation process is simple, and involves creating a file with *.xt extension where two areas are demarcated: one where the expected response is defined, and other where the model that supports the code generation is written - Code snippet 33.

Code snippet 33 - Xpect test example.

```
/* XPECT_SETUP com.tmdei.xtext.dsl.tests.generator.GeneratorRsrcTest END_SETUP */

/*
test generating FoodEntity.java
XPECT generated file nutrition.clinic/src/main/java/nutrition.clinic/business/entity/FoodEntity.java
---
package nutrition.clinic.business.entity;

import nutrition.clinic.business.entity.BaseEntity;

public class FoodEntity extends BaseEntity {
  private String name;
  private Double fats;
  private Double proteins;

  // setters and getters
}
--
*/

project: "Nutrition Clinic"
version: "1.0"
basePath: "nutrition.clinic"
server: {
  description: "Nutrition clinic development server"
  url: "http://nutrition.clinic.dev.com"
}

resource Food table food {
  attribute name String;
  attribute fats Double;
  attribute proteins Double;
}
```

The test will assert the equivalence between the code generated from the model defined, and the expected result. To accomplish a reliable test suite, ideally all of the scenarios that can be built from the grammars should be tested.

4.4.3 Generation process

The code generation process can be divided in three different major components: one focused on the entities, defining the code artifacts that represent the business entities (ResourceEntity.java) and the associated data transfer objects (ResourceApi.java) and database objects (ResourceDbo.java), as well, the mappers that ensure their transformations across the multilayered architecture; other in the definition of behavioral artifacts that control the flow of the application,

[1] http://www.xpect-tests.org/

namely the "ResourceManager" and "ResourceWorkflow" classes; and finally, the services entry points, the "ResourceWs" implementations.

A formal structure was established and integrated in all the individual generation processes - Code snippet 34. This approach was followed in both of the code generations processes developed.

Code snippet 34 - Structure followed in the generation process.

```
class Generator {
  // inject necessary helpers
  @Inject extension GeneratorHelper

  // local variables declarations
  Meta meta = null
  Resource resource = null
  Attribute attributes = null

  def generate(Meta meta, Resource resource, IFileSystemAccess2 fsa) {
    // local variables attributions
    this.meta = meta
    this.resource = resource
    this.attributes = resource.attributes

    // path and file name definitions
    fsa.generateFile(
      FileName + ".java",
      generateClass()
    )
  }

  // function responsible of generating the software class structure and content
  def generateClass() '''
    «generatePackage()»
    «generateImports()»
    «generateEntity()»
  '''

  // function responsible to generate the package declaration
  def generatePackage() '''
  '''

  // function responsible to generate the imports declaration
  def generateImports() '''
  '''

  // function responsible to generate the entity content, usually through a defined template
  def generateEntity() '''
  '''
}
```

Each one of the previous mentioned components generation is detailed in the following sections, and, given the relevant structural differences between the processes associated with the developed grammars, the explanation is extended to each one independently.

4.4.3.1 Syntactic considerations

Regarding the naming conventions adopted for the code generation process, it is important to make some considerations:

- The code itself follows the usual Java conventions[1];
- As for the specificities of the web services, node names were considered plural by default:

[1] https://www.oracle.com/technetwork/java/codeconvtoc-136057.html

`GET nutrition.clinit/clients/33245 and http://nutrition.clinit/clients`

- For the database table and column names it was assumed:
    - Tables: use the resource name in Upper Camel Case, ex: FavoriteFoods;
    - Columns: use the attribute name in Lower Camel Case, ex: clientId.

4.4.3.2 Resource-oriented code generation process

The resource-oriented grammar was built around the resource concept itself, which vastly decreases the processing required to identify the resources that will give origin to a code artifact. A simple iteration through the model resources will lead to a software class associated with the identified entity. Given some specific attributes characteristics presented in the model the resulting class may result in a slightly different variant with specifications depending on those characteristics significance.

Taking the resource presented in Figure 20 a), the code generation process will apply a template through an Xtend class, like the one shown in Code snippet 35, and generate the respective business artifact *UserEntity.java*.

Code snippet 35 - Business entities generation process.

```
class BusinessEntityGenerator {
  @Inject extension RsrcDslGeneratorHelper
  @Inject extension BusinessLayerGeneratorHelper

  def generateBusinessEntity(Meta meta, Resource resource, IFileSystemAccess2 fsa) {
    this.meta = meta
    this.resource = resource
    this.composites = resource.contains
    this.attributes = resource.attributes
    this.entity = resource.generateBusinessEntityClassName

    if (resource.extends !== null) {
      this.parentResource = resource.extends.name
    }

    fsa.generateFile(
      meta.package.retrieveBusinessLayerBaseFolder.retrieveSourceJavaFolder +
      meta.package.retrieveEntityFolder + entity + ".java", generateClass())
  }

  def generateClass() '''
    «generatePackage()»
    «generateImports()»
    «generateEntity()»
  '''

  def generateEntity() '''
    public class «entity» extends BaseEntity {

      «generateAttributes(attributes)»
      «generateComposites(composites, "Entity")»

      «generateConstructorDefault(entity)»
      «generateConstructorWithParameters(resource, entity)»

      «generateAttributesGettersAndSetters(attributes)»
      «generateCompositesGettersAndSetters(composites, "Entity")»
    }
  '''
}
```

The previous code snippet illustrates how a business entity class is generated. It can be seen that is was intended to promote code reuse as much as possible, by delegating small tasks in smaller functions, accessible through helpers, and made available to all of the code generation components by injection.

Another code generation process example is shown in Code snippet 36, in this case the Manager class generation. Since the most relevant difference is at the entity template definition, that is the focus of the example, demonstrating that the code generation structure achieves high levels of code reuse, increasing the productivity when new artifacts are required to be generated.

Code snippet 36 - Business entities generation process.

```
class BusinessEntityGenerator {

  // …

  def generateEntity() '''
    /**
     * «businessEntity» manager
     */
    public class «businessEntityManager» extends BaseManager<«businessEntity», «databaseEntity»> {

      // DAOs
      BaseDao<«databaseEntity»> «databaseDaoVariable»;

      // Mapper
      «businessEntityMapper» «businessEntityMapperVariable»;

      @Override
      protected BaseDao<«databaseEntity»> getDao() {
        if («databaseDaoVariable» == null) {
          this.«databaseDaoVariable» = new «databaseDao»();
        }
        return this.«databaseDaoVariable»;
      }

      @Override
      protected BaseEntityMapper<«businessEntity», «databaseEntity»> getEntityMapper() {
        if («businessEntityMapperVariable» == null) {
          «businessEntityMapperVariable» = new «businessEntityMapper»();
        }
        return this.«businessEntityMapperVariable»;
      }

      @Override
      protected Class<«databaseEntity»> getDboClass() {
        return «databaseEntity».class;
      }

    }
  '''

  // …

}
```

The gateways generation process follows the same lines of the previous components. The major difference is in the overall behavior of the final application. Given that the each one of the ResourceWs extends a BaseWs, where all the CRUD are made available by default, when a specific action is not supposed to be available it must be generated with an overridden annotation, raising an exception - Code snippet 37.

Code snippet 37 - Example of an overridden service.

```
@Path("clients")
public class ClientWsImpl extends BaseWs<ClientApi, ClientEntity, ClientDbo> {
  // …

  @Override
  @DELETE
  @Path("{id}")
  @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
  public Response delete(@PathParam("id") long id) {
    return Response.status(Status.NOT_IMPLEMENTED)
        .entity("The requested operation is not supported for this resource.").build();
  }
```

Lastly, the generation of the OAS from the model is achieved through a mapping process between the most relevant objects from the specification and the grammar rules, as specified in Table 15.

Table 15 - Mapping between classes and OpenAPI components artifact

Grammar rule	OpenAPI artifact
Meta	Info and server object
Resource	Component object
Resource Attribute	Component properties field
Resource Relation and Action	Paths object

Additional OAS objects, like the Request Body object and Response object, are inferred from the Resource-oriented grammar rules:

- The Response object is expected to be a representation of the resource, i.e., for a Client POST request the response is a Client representation;
- The Request Body object, associated with the POST and PUT requests, is assumed to be a representation of the associated resource.

This section summarized the main processes of code generation for the resource-oriented grammar. Although the OAS-oriented grammar is substantially different, the code generation process follows the same philosophy, with the same structure of code generation classes. Given this assumption the following section focus on how the mapping between the OAS objects and the code artifacts is made, rather than the code generation.

4.4.3.3 OAS-oriented code generation process

To provide a better understanding on how the web service code is generated from the developed DSL and consequent model, an UML-OpenAPI base mapping approach is defined, where the involved rules are exposed in table format and are structured built on the UML artifacts defined in Figure 23.

The main components that must be identified and extracted from the specification are the resources that compose the substance of the web service, and the endpoints that needed to be exposed in each one.

Following the OpenAPI structure, and analyzing the existing objects, it can be observed that two of them provide the required info to identify the mentioned resources and services:

- **Paths object:**

```
paths:
  /foods:
    get:
      description:
      operationId:
      parameters:
      responses:
      (…)
    post:
      (…)
```

The Path element contains a relative path to an individual endpoint and the operations for the HTTP methods. The description of an operation (operation element) includes an identifier operationId, the MIME types the operation can consume/produce, and the supported transfer protocols for the operation (schema attribute). An operation includes also the possible responses returned from executing the operation (responses reference).

- **Components object:**

```
components:
  schemas:
  responses:
  parameters:
  requestBody:
  (…)
```

The components object behaves like an appendix where the re-usable details are provided. If multiple parts of the specification have the same schema, each of these references can be pointed to the same object in the components object, and in so doing you single source the content.

The following tables expose the most relevant mappings between the OpenAPI objects and the UML artifacts.

Table 16 reflects how the entities are mapped from the OpenAPI components object to UML classes, materializing the web service resources, and allowing the code implementation.

Table 16 - Mapping between classes and OpenAPI components artifact

UML/Code Artifact		**OpenAPI components artifact**
Entity	name	schemas field → key in Map[string, schema object | reference object]
	attributes	schemas field → schema object → properties field
	generalization class	schemas field → schema object → allOf (and the combining schemas)
Entity attributes	name	schemas field → schema object → properties field → name
	type	schemas field → schema object → properties field → type
	multiplicity	schemas field → schema object → properties field → type array

Supported by this mapping process the specification document is parsed and using Xtend templates the entities are generated.

Figure 27 illustrates the mapping result between the component artifact show in Code snippet 30, and UML concepts, originating a class diagram from which the resources are identified.

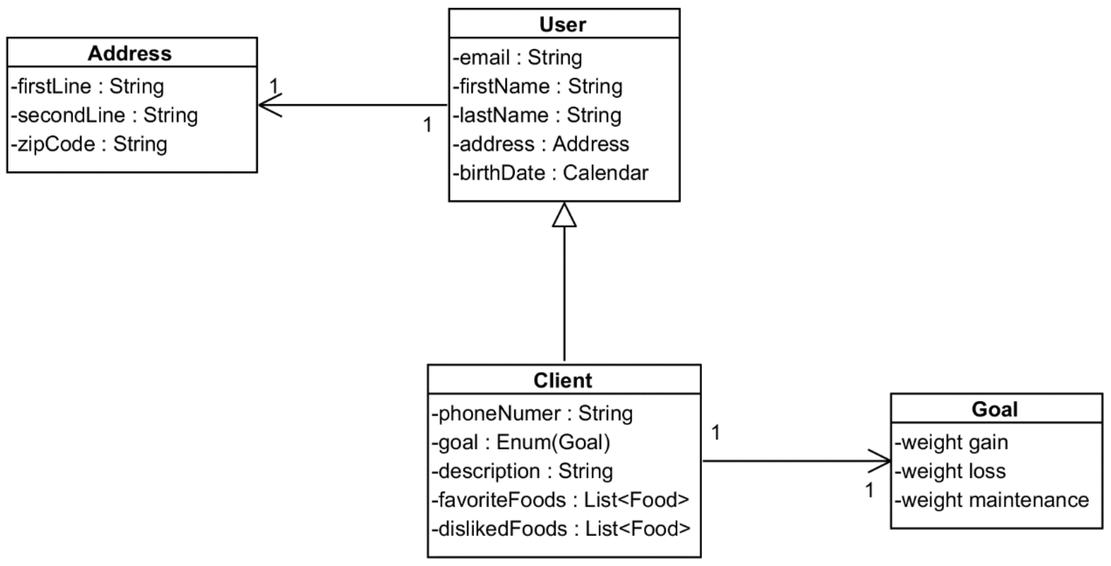


Figure 27 - UML class-OpenAPI component artifact mapping

This mapping allows the definition of the web service main resources (User and Client), its attributes and its relationships with other sub resources (Address). Even though the "required" attribute, inside the Client schema, has no relevance in the GET service, when creating a new Client - in the POST service - this constitutes an initial validation in the web service, where these attributes must be present in the body of the request associated. In the following code-snippet the generated validation is presented.

Code snippet 38 - Example of a generated validation.

```
public abstract class UserBaseValidation implements EntityValidator<UserEntity> {

  public final boolean validate(UserEntity entity) throws UserException {
    if (entity == null) {
      throw new UsertException(UserErrorMessage.NULL_RESOURCE);
    }

    if (entity.getEmail() != null) {
      throw new UserException(UserErrorMessage.MISSING_REQ_PARAM, "email");
    }

    if (entity.getId() != null) {
      throw new UserException(UserErrorMessage.MISSING_REQ_PARAM, "id");
    }

    return customValidations(entity);
  }

  protected abstract boolean customValidations(UserEntity entity) throws UserException;

}
```

The OpenAPI specification has a specific object - Paths object - where the available services are exposed, identifying the most relevant attributes:

- Endpoint;
- HTTP method;
- Parameters respective location;
- Possible responses.

In Table 17 the web service mapping from the paths object is presented, identifying the correspondence between the specification and the code artifact.

Table 17 - Mapping between web service classes and OpenAPI paths object

UML/Code Artifact		OpenAPI paths object
Web service	path	paths object → field pattern
	HTTP verb	paths object → path item object → http method
	parameters	paths object → path item object → operation object → parameters
	request body	paths object → path item object → operation object → request body
	responses	paths object → path item object → operation object → responses
	content type	paths object → path item object → operation object → responses → content type
	security	paths object → path item object → operation object → security

As referred the Resources Ws extends a BaseWs with all the common CRUD operations, but when necessary these methods can be overridden, and the mapping process can be achieved like in the following example. Based on the Code snippet 29, the associated implementation is presented in Code snippet 39.

Code snippet 39 - Web service class - OpenAPI path artifact mapping.

```
(…)

@GET
@Path("{clientId}")
public class ClientWs extends BaseWs<ClientApi, ClientEntity, ClientDbo>  {

  private ClientWorkflow clientWorkflow;

  @Override
  @GET
  @Produces({MediaType.APPLICATION_JSON})
  public Response retrieveClient(@PathParam("id") long id){

    ClientApi clientApi = null;
    Response finalResponse = null;

    try {
      clientApi = getClientWorkflow().retrieveClient(id);

      response = new Response<>(clientApi, Status.OK);
    } catch (Exception exception) {
      response = new Error<>(“unexpected error”, Status.BAD_REQUEST);
    }

    return Response.status(response.getStatus()).entity(response).build();
  }

}
(…)
```

The exposed endpoint is easily identified and is followed by the related HTTP verb. Then, the required parameters are exposed, the location in the request ("query", "header", "path" or "cookie") and the mandatory character of the parameter. Succeeding, the possible responses are detailed, with the reference pointing to the component object detailing the response schema, or even the schema itself.

These two mapping procedures are the most relevant ones in the web service building process, allowing the definition of the resources, and the endpoints to be exposed. From this two mapping processes, mainly the resources identification process, other code artifacts are inferred:

- Database entities and SQL scripts;
- API entities;
- Associated mappers.

Information about the server or servers that support the API function, with the URL that incorporates the available endpoints is detailed in the server object, allowing the definition of these structural parameters in the implementation.

Furthermore, the OpenAPI specification provides information regarding other components that can compose a web service, but whose scope is not the focus of this work:

- Security requirement - allows the specification of which security mechanisms can be used across the API;
- External documentation - reference to external documentation regarding the web service.

The next section provides a broad comparative analysis between the two grammars developed and associated code generation processes, electing some foreseeing advantages and disadvantages in their usage.

4.5 Broad comparative analysis

4.5.1 Grammar

The two developed grammars have a distinct syntax: one focused on the resource definition, while the other mimics the OpenAPI specification. This will lead to different usability levels given the developer knowledge in the OAS thematic.

To developers already familiar with the OpenAPI specification the OAS-oriented grammar usage may be preferred, while developers who do not know about the specification, given the simpler syntax of the Resource-oriented grammar, will prefer to use the last one. However, even for experienced developers, the resource-oriented grammar may prove itself as a valuable tool, given its capacity to generate the specification from a far simple model.

This may lead to a different flow in the web service development process, by using in a first approach a model of the intended API based on the Resource-oriented grammar, which will generate the OpenAPI specification, that can be used as an input in the OAS-oriented grammar.

4.5.2 Code generation

Focusing in the code generation, while it presents itself as a very similar process in both fashioned grammars, it can be alleged that the Resource-based one possesses a more “convention over configuration” character than the other, namely on:

- The services path definition, where they are inferred from the resource definition, while in the OAS grammar they are explicit defined;
- The states transitions intrinsically associated with the HTTP verbs of the respective service path;

- The web services responses, explicitly defined in the OAS grammar, are pre-defined:
    - Default response: error message "Bad Request" with some detail regarding the error;
    - POST response: the representation of the resource created;
    - PUT response: the representation of the resource updated;
    - GET response: a list of the resources representations;
    - GET with id: the resource representation with the associated id;
    - DELETE response: the deleted resource representation;

Chapter 5 draws some conclusions regarding the solutions evaluations, based on the points defined in Section 3.6.

Chapter 5

Solution Evaluation

This chapter is dedicated to the solution evaluation, focusing on the parameters defined in 3.6. Firstly, the usability concerns are addressed, then the generated code quality is compared with the results achieved through commercial solutions, and finally the web services QoS are evaluated against the case study, focusing on the performance of two concrete webservices, detailed in the respective section. All the evaluations that have code execution involved were executed under the same circumstances (local server) to avoid entropy caused by additional variables.

5.1 DSLs usability

Leveraging in the main concepts regarding DSLs' usability evaluation exposed in Section 3.6.1, it is necessary to first undergo some research activities (Barišić, Amaral, Goulão, et al., 2012):

1. Domain analysis;
2. Language design;
3. Testing - controlled experiment;
4. Deployment and maintenance;
5. Validation - iterative life-cycle.

The first two points were already covered in previous sections. Also, the mean to evaluate the quality in use of the DSLs was already identified in the section referred in the first paragraph. The last two points are considered to be out of scope regarding the context of this work. They are intimately associated with the continuous use of the DSLs in a production environment, where an iterative process of identifying improvement points and act accordingly will allow the DSL usability and overall performance of the DSL and code generation process to be bettered.

Due to time constraints it was not possible to conclude the evaluation assessment, but the methodology was delineated. This section focusses on presenting the process to undertake a DSL usability measurement, through a user-centered assessment of the DSLs usability.

5.1.1 Evaluation process

The first step of the evaluation consists in the identification of the users' profiles and context of the DSL usage. Following a similar approach as the one presented in (Barišić, Monteiro, Amaral, Goulão, & Monteiro, 2012), **Erro! A origem da referência não foi encontrada.** illustrates the user profile characterization using a Likert scale (1-unimportant, 5-very important) to prioritize their importance in the usability evaluation process.

Table 18 - DSL user's profiles.

Technical characteristics				**Profile characteristics**			
Knowledge about the business processes	4			Manager	3		
Knowledge about the business domain	4			Domain expert	4		
Knowledge of programming	3	Java programming	3	Developer	5	Engineer	4
		DSL usage	5			Programmer	3

Identified the users' profiles relevant to the DSL, the process starts with the participant recruitment, categorizing each one accordingly with the profiles identified. The next step is to organize the evaluation by determining which tasks must be done in order to provide meaningful results to latter analysis. The pilot session follows, meant to simulate the exam and provide some insights regarding the adequacy of the prepared material to be used in the evaluation procedures. An evaluation sitting then takes place, with a training session, where the languages are introduced. The exam involves some writing activities where the participants actions are observed and recorded, so that completion times can be tracked. After each group has been evaluated in the different languages they must fill a questionnaire, to obtain the user's qualitative perspective of the comparison between the languages. The evaluation process terminates with the results analysis. The previous described process is illustrated in Figure 28.

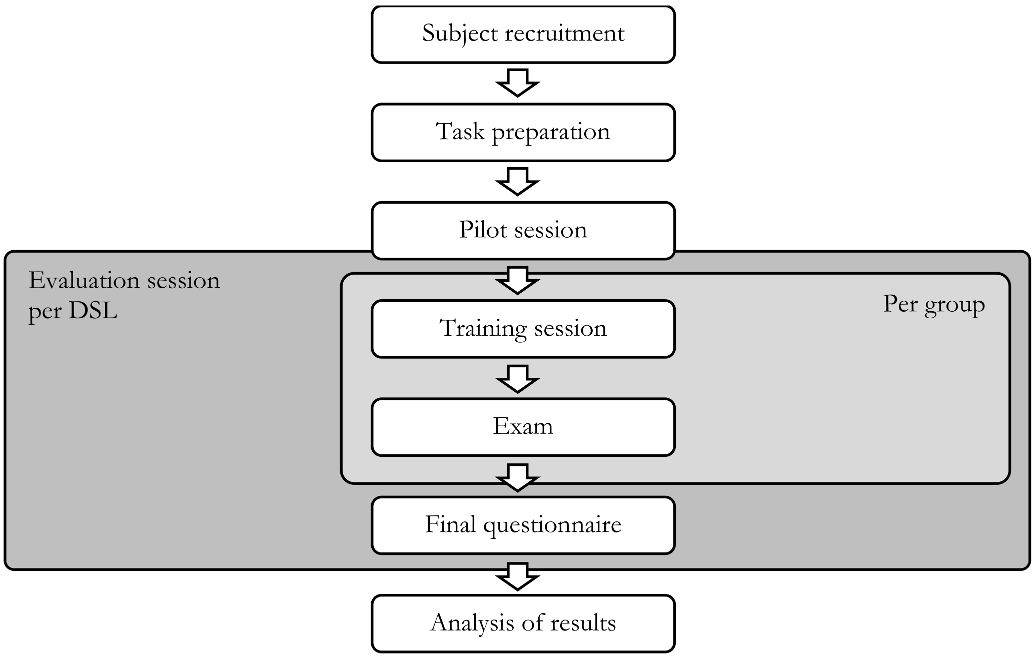


Figure 28 - Evaluation process steps.

Fixed on the task preparation sub process, it is important to establish what features to evaluate:

- OAS-oriented DSL vs Resource-oriented DSL;
- Common procedures to both languages:
  - Expressing the main entities;
  - Defining the entities attributes and their data types;
  - Stating the relation between entities;
  - Describe the available HTTP methods to each of the resources;
  - Define meta data related to the web service.
- OAS-oriented DSL specific evaluation
  - Defining the paths to be exposed in the web service;
  - Expressing the possible responses for each service;
  - Defining the parameters required to make the request to each service.
- Resource-oriented DSL specific evaluation
  - Define different type of relations between different entities;
  - Expressing custom web services involving two entities.

One important factor to refer in order to achieve the best results possible, is the necessity of split the participants in two different groups, one that will use the OAS-oriented analysis and the other that will use the Resource-oriented. Only through this way the influence of the first language while presenting the second is mitigated, ensuring an unbiased evaluation methodology for each language. After obtaining these impartial results, another two sessions would then be needed for the users of each language to switch the target one, allowing a comparison layer between the performance of the DSLs.

After the session the participants are asked to answer the questionnaire to judge the intuitiveness, suitability and effectiveness of the two languages. The purpose is to evaluate:

- Overall reaction to each of the languages;
- Rating how easy specific aspects of the language are to use;

And finally, to compare both languages:

- The participants are asked to compare specific aspects of both languages and rate the preferences they have;
- The participants are invited to comment freely on language that they have just used.

5.2 Code metrics analysis

The metrics of Chidamber & Kemerer (CK) (Chidamber & Kemerer, 1994) are based on solid measurement theory and are oriented to OO programming languages, like Java, focusing on some fundamental characteristics of this paradigm. The CK Metrics set consists of six metrics: Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Object Classes (CBO), Response For a Class (RFC), and lack of Cohesion in Methods (LCOM).

Each one of these metrics are detailed in the following section and later were calculated for the generated code and for the available commercial solution that supports the OpenAPI specification 3.0 - the Apimatic suite.

5.2.1 Chidamber & Kemerer metrics (C&K)

Chidmanber & Kemerer developed this set metrics with the objective of establishing the OO application design. The intent was to predict, when faced with two different designs for a same project, which would be the best. This reveals another facet of theses metrics, they can be design-based rather than code-based.

A simple description for each of this metrics follows:

- **Weighted methods per class (WMC)**

  The WMC magnitude represents the total complexity of the methods of a class. The value is given by the sum of the complexities of each method. A simplified way to calculate WMC is to consider methods with the same value for complexity, this value being equal to 1. In this case, the WMC value is equal to the number of methods (NOM) in the class.

  The number of methods in a class and the complexity of these classes helps to estimate the time and effort to develop and maintain the class. Classes with a large number of methods are more specific to certain applications and limit the possibility of reuse.

- **Depth of Inheritance Tree (DIT)**

  The inheritance tree depth for a class is defined as the maximum length of the node representing the class up to the root of the tree (more abstract classes).

  Inheritance, or generalization, can increase the complexity of a class because the developer must know, in addition to the methods of the class itself, all related methods and attributes that this class inherits. In this case, the deeper the inheritance tree, the greater the number of methods to be considered.

- **Number of Children (NOC)**

  The number of subclasses is the number of direct subclasses of a class. According to (Chidamber & Kemerer, 1994), the higher the number of direct subclasses of a class, the greater the possibility that the inheritance was misused for this class, in other words, inheritance tree levels are likely to be missing. A class with many direct subclasses has a very large potential for propagating the effects of change in one of its methods, requiring additional testing.

- **Coupling between object classes (CBO)**

  The coupling scale between objects of a class is defined as being the number of other classes with which this class is coupled (related through an association). Two objects are bound when one object's methods use methods or instance variables of another. In other words, the number of classes of which this class uses methods or instance variables.

  Excessive coupling between objects in a system impairs modular development and makes reuse difficult. Even if this coupling is through method calls, it causes the developer to focus on several classes other than the one he is designing. Encapsulation is a way to prevent coupling.

- **Response for a Class (RFC)**

  The response-to-class (RFC) is defined as the cardinality of the set of responses of a class. The response set of a class is the group of methods of a class that can potentially be executed in response to an incoming request. The greater the set of methods that can be called from a class, the greater the complexity. If a large number of methods of a class can be called by other classes, the testing and correction operations become more complex, requiring more experience on the part of the developer.
- **Lack of cohesion in methods (LCOM)**

  The LCOM is the difference between the number of pairs of methods in a class that do not share the same set of instance variables (attributes) and the number of pairs they share. LCOM measures the cohesion between the methods of a class. If LCOM is high, this may mean that the class can be divided into two or more sub-classes.

A study was conducted by the NASA Goddard Space Flight Center's Software Assurance Technology Center (SATC) on the CK metrics used to evaluate this center's projects (Rosenberg, Stapko, & Gallo, 2000). C++ and Java programs, in a total of 20000 classes and 15 programs, were collected and analyzed in a 3 years period, with the purpose to find acceptable limits for C&K metrics in order to help identify pieces of code difficult to maintain, test, or understand - Table 19.

Table 19 - Threshold values for the individual metrics.

Metric	**Threshold values**
NOM: Number of methods	≤ 20 (preferred); ≤ 40 (acceptable)
WMC: Weighted methods per class	≤ 25 (preferred); ≤ 40 (acceptable)
DIT: Depth of Inheritance Tree	< 2 → may represent poor exploitation of the advantages of OO design and inheritance
	> 5 → widely used inheritance but high complexity (for classes with DIT > 5 attention to the other metrics as they underestimate their complexity)
NOC: Number of Children	no good or bad value was found as it depends on other metrics
CBO: Coupling Between Objects	< 5
RFC: Response for Class	≤ 50
LCOM: Lack of cohesion in methods	not evaluated in the referred study

5.2.2 Collected data

(Lincke, Lundberg, & Löwe, 2008) presents a study regarding the main tools available to calculate the referred metrics. They gathered the most relevant software that allowed the determination of the C&K metrics at the time, and given the most recent available versions, the VizzMaintenance[1] ,formerly known as the VizzAnalyser, was chosen to proceed with his analysis.

5.2.2.1 Statistic summary

Presented in Table 20 are the calculate metrics from the generated code through the methodology presented in this work, and in Table 21 the metrics associated with the commercial solution from Apimatic. In both situations the input model is the OAS showed in Appendix E.

[1] http://www.arisa.se/vizz_analyzer.php

Table 20 - Generated code: metrics descriptive statistics.

Metrics	N	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
WMC	72	0	4	7	7.889	12	26
NOM	72	0	2.75	5	5.792	8	19
DIT	72	0	0	0	0.4028	1	2
NOC	72	0	0	0	0.3194	0	5
CBO	72	0	1	4	3.819	6	11
RFC	72	0	3	7	9.389	13.25	39
LCOM	72	0	1	23	36.19	48.75	257

Table 21 - Apimatic code: metrics descriptive statistics.

Metrics	N	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
WMC	38	0	3	5	11.03	11.75	70
NOM	38	0	3	4.5	7.711	9.750	29
DIT	38	0	0	0	0.3158	1	1
NOC	38	0	0	0	0.3421	0	3
CBO	38	0	1	2.5	3.158	5	9
RFC	38	0	4	8	9.921	12.750	36
LCOM	38	0	0	2	90.47	69.25	789

From the results presented in the previous tables some conclusions can be drawn regarding the code metrics from both solutions:

- The generated code from the methodology presented in this work (N=72) has almost twice as many classes as the number of classes of the Apimatic solution (N=38);
- Overall, both the solutions present satisfactory metrics when compared with the thresholds from Table 19;
- The values obtained for the median and mean are in the same magnitude, which could indicate that both the solutions have a similar quality regarding the C&K metrics;
- NIT reveals itself as the least performant metric in this study, since the values achieved are inferior to the threshold of 2, which could reveal a poor exploitation of the advantages of OO design and inheritance.

5.2.2.2 Hypothesis analyses

Based on what was described in section 3.6.2.4 the comparison analysis follows with a hypothesis test, where it is assessed if exists statistical evidences to corroborate H02 and H12. H01 and H11 hypotheses are not presented in this section, on the account of the different paradigms followed in the case study project and in the code generation process. The case study application follows a more functional paradigm, in detriment of an OO approach, since the language in which it was built (Ruby) skewed the development in that direction. This overturns the meaning that could be collected from C&K metrics in the evaluation of the code metrics significance.

Focusing attention on the hypotheses:

- H02: No significant difference exists between the performance of static metrics of the generated code and the commercial available solutions.
- H12: The generated code metrics performance is relatively better that the ones calculated from the commercial available solutions.

First the normality of the sample must be evaluated. For samples of high dimension, by application of the Central Limit Theorem, TLC, it is possible to infer its normality. For samples of reduced size, it is advisable to test the normality of its distribution by performing certain tests.

For the case in analysis the Shapiro-Wilk was conducted for each one of the metrics in study - Table 22.

The null-hypothesis of this test states that the population is normally distributed, so, if the p-value is less than the predefined significance level (5%), the null hypothesis is rejected, and it can be concluded that there are statistic evidences that the data does not follow a normal distribution.

Table 22 - Shapiro-Wilk test of normality.

	Generated code		**Apimatic generate code**	
Metric	**p-value**	**H_0**	**p-value**	**H_0**
WMC	0.005634	rejected	4.1e-08	rejected
NOM	0.0008207	rejected	1.567e-05	rejected
DIT	5.501e-12	rejected	3.722e-09	rejected
NOC	1.064e-15	rejected	3.437e-10	rejected
CBO	0.001809	rejected	0.004387	rejected
RFC	1.493e-06	rejected	0.001208	rejected
LCOM	1.044e-10	rejected	1.503e-09	rejected

Ensuing the normality tests results, it is obvious that the populations for each metric do not follow a normal distribution.

This invalidates the initial proposition of proceed with a parametric hypothesis test, leaving only the non-parametric tests available to conclude this analysis. The Mann-Whitney-Wilcoxon Test can then be used to assess the previous defined hypothesis since the conditions to validate its usage are fulfilled:

- Independent data samples;
- Variable is ordinal or continuous;
- The shape of the distributions be similar.

The two first conditions have already been stated in previous sections, while the shape similarity of the distributions can be confirmed in Figure 29 by analyzing the box plots for each metric, regarding the two solutions in evaluation.

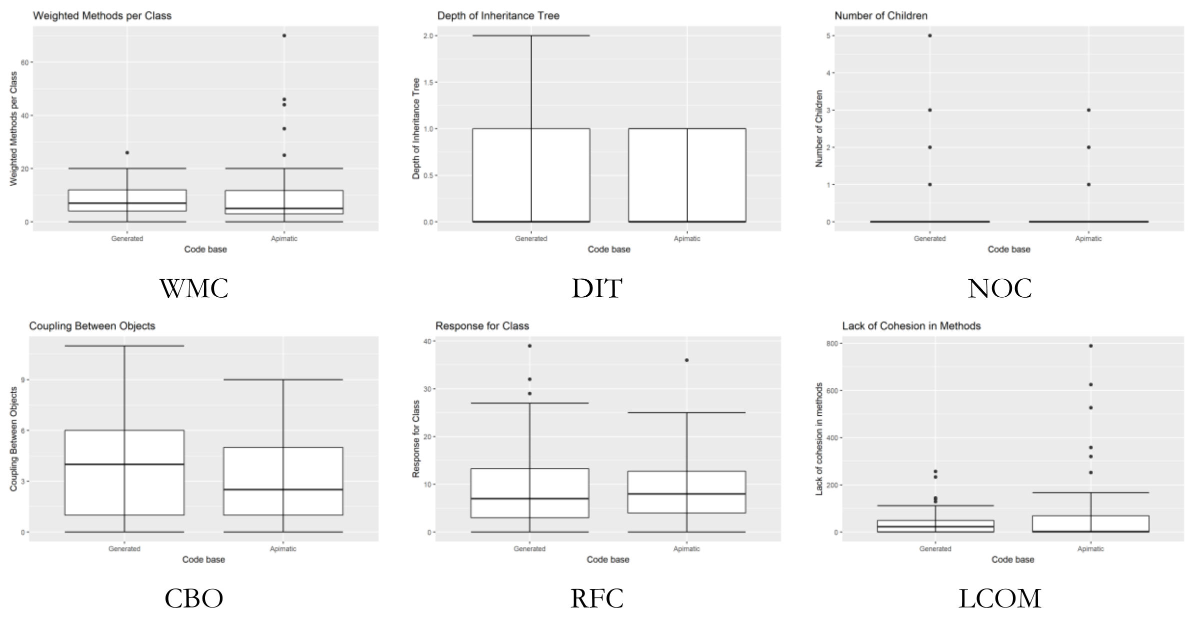


Figure 29 - Shape distributions for metrics values.

Table 23 illustrates the obtained results for each metric, considering a significance level of 5%. The null hypothesis states if the solutions code metrics can be said to be knowingly different, while the alternative hypothesis (alternative = less) will evaluate whether the mean (or location) of the first group (generated code) is lower.

Table 23 - Mann-Whitney-Wilcoxon test.

Metric	p-value	H_{02}	p-value	H_{12}
WMC	0.5747	accepted	0.7148	accepted
NOM	0.5787	accepted	0.2893	accepted
DIT	0.7548	accepted	0.6255	accepted
NOC	0.2925	accepted	0.8556	accepted
CBO	0.2558	accepted	0.8734	accepted
RFC	0.5392	accepted	0.2696	accepted
LCOM	0.1459	accepted	0.9279	accepted

For the null hypothesis, H02, p-value is greater than the significance level of 0.05, so it can be concluded that it does not exists sufficient evidence to conclude that the code metrics differ in each solution. As for the alternative hypothesis, H12, it can be stated that the results obtained seem to be showing a location inferior from the generated code to the Apimatic solution.

This indicates that the code metrics might be assuming lower values for the solution presented in this work, which, for the majority of the metrics indicates a better code base.

5.3 Web service QoS

Jmeter was the tool considered to proceed with the assessment of the QoS relevant metrics, for both, the generated code and the case study application. Two services were chosen to be analyzed, given the similarity between the implementations in both cases, and being two of the most used services in the case study application:

1. Search food:
   - `GET nutrition.clinic.gateway/api/foods ?filter=name:EQ:inhame`
   - `GET mypocketnutritionist/api/v1/tcas/search string=inhame`
2. Create client:
   - `POST nutrition.clinic.gateway/api/users`
   - `POST mypocketnutritionist/api/v1/registration`

5.3.1 Collected data

The tool selected allows to simulate multiple tests scenarios, by choosing the number of users (threads), time conditions (interval between requests, ramp-up time, etc.), variables that will be used in the requests, etc. For the current analysis a total of 50 requests were made, and the following QoS metrics registered, latency (ms), response time (ms) and throughput (number of requests per second).

5.3.1.1 Statistic summary

Table 24 and Table 25 show some descriptive statistics associated with each service analyzed and for each implementation considered.

Table 24 - Search food service QoS descriptive statistics (N=50).

QOS	Generated code						Case study					
	Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**	**Min.**	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
Latency, ms	5	5	5	5.7	6	15	7	7.25	8	7.78	8	9
Response time, ms	5	5	5	5.7	6	15	7	7.25	8	7.78	8	9
Throughput, nº requests/s	10.2						9.6					

Table 25 - Create user service QoS descriptive statistics (N=50).

QOS	Generated code						Case study					
	Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**	**Min.**	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
Latency, ms	5	7	7	7.16	7	20	7	10	11	10.32	11	13
Response time, ms	5	7	7	7.16	7	20	7	10	11	10.32	11	13
Throughput, nº requests/s	10.2						9.6					

Reading the obtained statistics for each service, and comparing the overall performance of the targeted implementations, at first sight it could be concluded that the generated solution provides a faster response time for both services. However, it is important to state that the execution time (Response time - Latency) is zero for both cases, which indicates that the time spent executing the code can be neglected and the difference registered is probably associated with other variables. Since both the implementations use a MySQL server as SGBD, the deployment servers where the solutions are deployed could be the origin of the registered differences: Glassfish in the generated code and Redis for the case study application.

This could indicate that, despite the original suspicions that the poor performance of some services in the case study application was related to the code implementation, the problem actually resides in the technological stack used in the implementation.

The aforementioned conclusions impact what can be inferred from the planned hypotheses, since they might not be related to the implementation itself.

5.3.1.2 Hypothesis analyses

As previous mentioned the hypotheses in analysis regarding the quality of service metrics for the services in analysis are:

- H03: No significant difference exists between the web services performance of the generated code and the case study project.
- H13: The generated web services performance is relatively better that the one provided by the case study project.

The approach follows the steps referenced in the previous section hypotheses analysis, by start testing the distribution normality through the Shapiro-Wilk test - Table 26 and Table 27.

Table 26 - Shapiro-Wilk test of normality: search service.

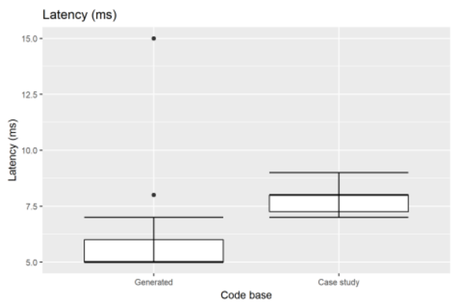
	Generated code		**Case study**	
Metric	**p-value**	**H0**	**p-value**	**H0**
Latency	8.215e-13	rejected	2.869e-09	rejected
Response time	8.215e-13	rejected	2.869e-09	rejected

Table 27 - Shapiro-Wilk test of normality: create service

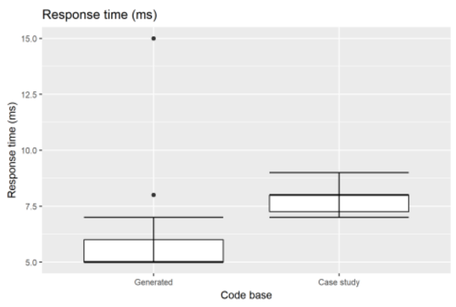
	Generated code		**Case study**	
Metric	**p-value**	**H0**	**p-value**	**H0**
Latency	5.551e-13	rejected	3.739e-06	rejected
Response time	5.551e-13	rejected	3.739e-06	rejected

Once again, the statistical evidences suggest that the data collected from both services do not follow a normal distribution. This leads the hypotheses evaluation through non-parametric tests.

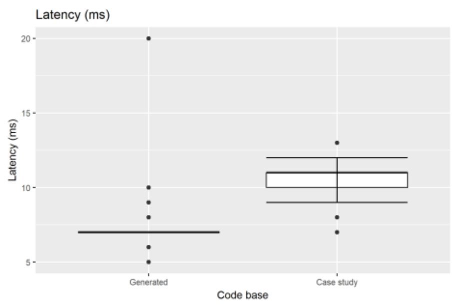
The analysis conducted follows the same steps that the one in Section 5.2.2.2. First the distribution shape - Figure 30 - is examined for its similarity. Then the Mann-Whitney-Wilcoxon tests are executed, for each service, and each implementation - Table 28 and Table 29.



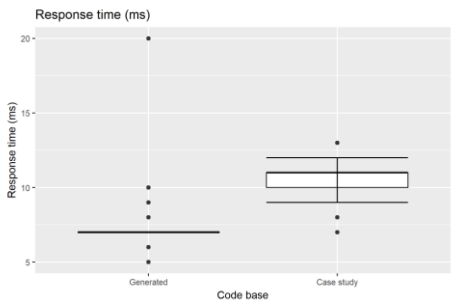
Search service latency



Search service response time



Create service latency



Create service response time

Figure 30 - Shape distributions for metrics values.

Table 28 - Mann-Whitney-Wilcoxon test: search service.

Metric	p-value	H_{02}	p-value	H_{12}
Latency	2.2e-16	accepted	2.2e-16	accepted
Response time	2.2e-16	accepted	2.2e-16	accepted

Table 29 - Mann-Whitney-Wilcoxon test: create service.

Metric	p-value	H_{02}	p-value	H_{12}
Latency	2.783e-16	accepted	2.2e-16	accepted
Response time	2.783e-16	accepted	2.2e-16	accepted

The conclusions that can drawn from the results attained are identical to the ones obtained before, in the comparison with the Apimatic solution. It can be said that statistical evidences advocate that both solutions do not show relevant differences in the overall performance of the web services QoS (H0). When executing the unilateral version of the test, the results suggest that the QoS for the generated solution performs relatively better than the case study.

Chapter 6

Conclusion and future work

This chapter aims to summarize the results of the work done, highlighting the completed objectives, difficulties encountered, limitations and future work. Finally, it concludes with some statements about the work carried out in the framework of the TMDEI course.

6.1 Summary

This document summarizes the different phases of execution of the developed solution, from the value analysis of the projected work in a contemporary context, to the requirements analysis and definition, grammars modelling and implementation, code generation processes and the solutions evaluation.

Before presenting the technical and development components at the level of the implementation itself, a contemporary framework is provided in the field of model driven engineering techniques applied to web services development This information is exposed in Chapter 2, where a detailed value analysis is presented, highlighting the benefits that the proposed methodology brings to the different stakeholders. The main purposed of this chapter was to give some deep context of the overall concepts connected in the final presented solution.

Chapter 3 emphasis the analysis of current available market solutions whose purpose is similar to what the author proposed to achieve with the development of this work. Following this preliminary analysis, the main requirements were raised, and a solution perspective was presented. With the resolution of giving a technological context, Chapter 3 also introduced the main technologies adopted in the execution of the project.

Chapter 4 engrossed the designed process, aiming to understand how the grammars could be built, by creating base meta-models, representative of the two major themes, the REST resources and the OpenAPI specification. From this initial design the grammars implementation follows, and then the code generation process is described.

Lastly, Chapter 5 takes on the evaluation procedures defined in previous sections and presents the obtained results, giving some perceptions over the code quality, usability evaluation procedures and web services QoS.

6.2 Goals achieved

Two different approaches based on MDE techniques were developed, one focused on the resource definition and other on the OpenAPI Specification. Both provide a solid, feasible and efficient MDE solution to aid in the development of RESTful web services, improving the overall performance of the development process, by ensuring a less error-prone environment, a faster implementation methodology, and by providing a common platform where both, the domain experts and the developers, can define the business concepts interactively, reaching a consensus for what will be implemented.

Remembering the four initial comprehensive goals that this work was envisioned to achieve during its development:

1. What MDE approach can be adopted to ensure a more efficient and reliable process of web services development?
2. What are the compromises to specify a language agnostic metamodel to represent and define the OpenAPI Specification?
3. Can a code generation process be developed over the DSL referred previously, and consequently aiding developers in web services development?
4. How does an OAS-based DSL and code generation process compare with a Resource-based one, with a simpler language?

The first question is answered in the beginning of this section, where it is denoted that two different approaches were considered and developed in the context of model driven techniques applied to the development of web services. Both are focused on domain specific languages implementation, followed by a code generation process.

Focusing on the second question, it was presented a custom metamodel illustrative of the OpenAPI specification, from which a domain specific modeling language was developed. This model is a simplification of the main concepts that integrate the OAS, giving a high-level interpretation of the relationships between them, while supporting the grammar development. This metamodel answers the mentioned question, on what were the main compromises in the specification of a language agnostic model to represent the OAS, since it deliberately leaves out some OAS concepts. While this concludes as a compromise in its current state, the metamodel can be evolved, integrating the missing components, and then incorporating them in the code generation process.

The ensuing question/objective is intrinsically related to the previous one and was successfully achieved: a DSL was built over the metamodel previously defined, providing a platform that aids developers in the design process of web services, qualified with some intelisense (intelligent code completion) capabilities, giving suggestions on the possible elements for each OpenAPI object, while ensuring fully specification compliance. From this DSL a code-generation process was defined, whose main objective was to generate a full web service implementation, compliant with the RESTful constrains.

Comparing the OAS approach with the Resource-based one, it can be stated, that, while the OAS isn't broadly accepted as the common specification to describe web services behavior and structure, the second offers a simpler solution, with a less complex syntax, and more comprehensible resource definition. It can also function as a first step towards the OpenAPI specification adoption, since it generates the document specification as a result of the code generation process, allowing the developer to make the connection between the resource definition, the concrete implementation and the associated specification.

Loose coupling between contract and implementation is possible in these approaches, providing also additional focus in the creative process of the technical solution construction.

Another advantage that was foreseen in OAS-oriented approach is related with the provision of a communication platform easily understandable by both parts (technical and business). This requires further analyses, namely through the usability evaluation, since it is not clear that this OAS approach really provides a language that the businesses stakeholders understand, given the steeping learning curve associated.

Regarding the more technical aspects of the solution evaluation, namely the hypothesis tests executed, it can be stated that the solution implemented achieved satisfactory results when compared with the commercial solution from Apimatic. In the webservices QoS assessment, a difference between the web services was identified, but its source maybe not associated with the implementation itself, but with the technological stack used in both applications.

Analyzing the more tangible goals delineated in section 3.4.1, Table 30 shows the status of each one at the end of the time frame reserved or the development of this work.

Table 30 - Functional requirements implementation status.

Nº	Description			Status
Req_01	Create web service reference implementation that will guide the DSLs development			realized
Req_02	OpenAPI specification support	Req_02.1	Define DSL grammar	-
			- OAS based DSL grammar	realized
			- Resource based DSL grammar	realized
		Req_02.2	Implement Xpect tests	realized
Req_03	Code generation	Req_03.1	Generate database tables creation script	realized
		Req_03.2	Generate project structure	realized
		Req_03.3	Implement RESTful compliant architectural style	partially
		Req_03.4	Generate gateways layer	realized
		Req_03.5	Generate presentation layer	realized
		Req_03.6	Generate business layer	realized
		Req_03.7	Generate database access layer	realized
		Req_03.8	Generate unit-tests and integration tests	partially
		Req_03.9	Generate the OAS spec from the Resource-based DSL	realized

Examining the objectives status, some due considerations are presented:

- Enforcing HATEOAS in the application services response was done with a custom implementation, made from scratch, an option that revealed itself as an obstacle to the overall productivity of the base project development. As many external frameworks already provide solutions to integrate this behavior, it should be pondered the replacement of the developed solution with one already available;
- It was initially envisioned the generation of unit tests that would partially cover the generated code. Since the application flow revolves essentially around HTTP calls, they were

considered an unnecessary overhead, and replaced entirely with integration tests based on the Jersey framework;

- The mentioned integration tests cover the base CRUD operations, leaving the custom actions untested;
- Overall the main public methods have some introductory comments. The generation of natural language texts as meaningful comments could be integrated in the code generation process, but, despite some solutions mainly devoted to summary comments (Sridhara, Hill, Muppaneni, Pollock, & Vijay-Shanker, 2010; Sridhara, Pollock, & Vijay-Shanker, 2011), it represents a different research and not totally covered yet. Thus, this point was considered out of the scope of the work presented in this document.

Summarizing all the components developed in the context of this work: two DSLs were developed, one focused on the resource definition and the other in the OpenAPI specification; a reference implementation was built to support the code generation process, providing a code base from which the templates used in the Xtend were extracted; the final solution it is really a combination between the generated code and a base application, where the main methods are materialized and from which the generated code extends and/or overrides their implementations, assuring a better maintenance and promoting code reuse along the different layers.

6.3 Limitations and Future Work

The OAS metamodel that was presented establishes itself as the base for the OAS-based DSL development. However, it can be improved by detailing additional relationships to ensure a better coverage of the OAS multiple objects.

For the grammar specification definition, and in order to improve the overall usability of the domain specific language when developing OpenAPI models, changes should be made to allow the unordered definition of the OpenAPI specification objects. While the use of unordered groups arises ambiguity questions in the grammar definition and consequently results in larger decisions trees for the parser, the possibility of defining the OpenAPI specification without needing to comply to a specific order in the objects definition, will vastly increase the grammar usability.

Another improvement that can be easily achieved is related to the inclusion of security layers using the OpenAPI Security Requirement Object (SmartBear, 2018) for the definition of the prescribed security schemas. Given that several security solutions are well standardized, defining a template for the most common ones would result in a robust addition to the overall DSL and code-generation process, allowing the definition of a security layer in the web service access, from the Open API specification. However, these and other aspects are planned to be more widely addressed by applying Model Driven Security (MDS) (Basin, Clavel, & Egea, 2011; Basin, Doser, & Lodderstedt, 2005, 2006) guidelines and technologies. MDS is a specialization of model-driven development that uses security design models to drive the built of secure applications.

Related to the Resource-oriented DSL it can be further improved with new functionalitiesto cover areas that were not the focus of this work: add a security layer, provide a documentation management component, or the support of additional HTTP verbs, for instance.

Due to difficulties in finding an adequate population to answer the envisioned enquiries to empirically test through experiments with test users, a common problem in this area (Barišić, Amaral, Goulão, & Barroca, 2011), the DSLs' usability evaluation was not completed but the process was delineated. This should probably be the first step in a near future, to try to comprehend the real potential of both developed DSLs, and which one should have more time and resources invested.

Focusing on the code generation process and to improve the integration of this MDE process in development environments, a strategy (Generation gap pattern, Xtext protected regions, etc.), for the generation of specific code sections while preserving the others already implemented, would vastly improve the usability of the DSL.

As the generated code is tightly coupled to the base implementation in an attempt to improve the code quality and reusability by employing good development practices, without it the generated code loses completely loses its functionality. It should be target of further analysis the real benefit of having the code generation process to create independent and fully functional core implementations, in comparison of the current adopted strategy.

Related to a recent subject, the European General Data Protection Regulation, 2016/679/EU (European Union (EU), 2016) that enforces the protection of user's private data, the code-generation process could also ensure that sensitive data would be properly used. At the time the OpenAPI specification does not possess any attribute that allows the easy identification of data that needs to be encrypted, which means that while no official solution is provided, the inclusion of a custom attribute can be used for this and only purpose.

6.4 Final remarks

Overall, the performed work allowed the author to apply numerous concepts acquired during the first three years of the bachelor, complemented with two years of master's degree in software engineering. The knowledge and concepts assimilated from several relevant classes in the context of the application/software development, allowed to present two different solutions, that can also be complementary used. In addition, the learned subjects in Domain Engineering course of the Software Engineering track, proved to be very valuable in the development of the main subjects of this work.

The need to build a reference application following good development practices, was evident in the preliminary analysis since it would have impact in the code generation process, in the code metrics analysis, relevant in the context of this work, and later in the evolution and maintenance operations of the application itself. These questions, widely explored in the context of several courses, are indeed essential in the construction of a future proof application.

At a more technical level and within the field of the technologies employed in the solution development, the use and exploration of model-driven technologies and techniques, an area in continuous growth, allowed the acquisition of a set of skills that have already proved to be an asset within the scope of the current professional activities of the author of this work.

Bibliographic References

Aljazzaf, Z. (2015). Bootstrapping quality of web services. *Journal of King Saud University - Computer and Information Sciences*, *27*(3), 323–333. https://doi.org/10.1016/j.jksuci.2014.12.003

Allee, V. (2002). A Value Network Approach for Modeling and Measuring Intangibles, (November).

API Evangelist. (2015). Comparison of Automatic API Code Generation Tools For Swagger. Retrieved January 28, 2018, from https://apievangelist.com/2015/06/06/comparison-of-automatic-api-code-generation-tools-for-swagger/

Barišić, A., Amaral, V., & Goulão, M. (2012). Usability evaluation of domain-specific languages. *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on The*, 342–347. https://doi.org/10.1109/QUATIC.2012.63

Barišić, A., Amaral, V., & Goulão, M. (2018). Usability driven DSL development with USE-ME. *Computer Languages, Systems and Structures*, *51*, 1339–1351. https://doi.org/10.1016/j.cl.2017.06.005

Barišić, A., Amaral, V., Goulão, M., & Barroca, B. (2011). How to reach a usable dsl? moving toward a systematic evaluation. *Electronic Communications of the EASST: 5th Int. Workshop on Multi-Paradigm Modeling (MPM 2011)*, *50*(January), 13. https://doi.org/10.14279/tuj.eceasst.50.741

Barišić, A., Amaral, V., Goulão, M., & Barroca, B. (2012). Evaluating the Usability of Domain-Specific Languages. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, 386–407. https://doi.org/10.4018/978-1-4666-2092-6

Barišić, A., Monteiro, P., Amaral, V., Goulão, M., & Monteiro, M. (2012). Patterns for Evaluating Usability of Domain-Specific Languages. *Proceedings of the 19th Conference on Pattern Languages of Programs (PLoP), SPLASH 2012*. https://doi.org/http://doi.org/10.5281/zenodo.889927

Barukh, M. C., & Benatallah, B. (2013). A toolkit for simplified web-services programming. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 8181 LNCS, pp. 515–518). https://doi.org/10.1007/978-3-642-41154-0_42

Brooke, J. (1996). SUS - A quick and dirty usability scale. *Usability Evaluation in Industry*, *189*(194), 4–7. https://doi.org/10.1002/hbm.20701

Chidamber, S. R., & Kemerer, C. F. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Cosentino, V., Tisi, M., & Izquierdo, J. L. C. (2015). A Model-Driven Approach to Generate External DSLs from Object-Oriented APIs, 423–435. https://doi.org/10.1007/978-3-662-46078-8_35

Dimitrieski, V., Terz, B., Dimitrieski, V., Kordić, S., Milosavljević, G., & Luković, I. (2017). MicroBuilder : A Model-Driven Tool for the Specification of REST Microservice Architectures, (May).

Ed-douibi, H., Izquierdo, J. L. C., Gómez, A., Tisi, M., & Cabot, J. (2016). Emf-Rest. *Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16*, *2*(3), 1446–1453.

https://doi.org/10.1145/2851613.2851782

Ed-Douibi, H., Izquierdo, J. L. C., Gómez, A., Tisi, M., & Cabot, J. (2015). EMF-REST Generation of RESTful APIs from Models. *CoRR*, *abs/1504.0*, 39–43. https://doi.org/10.1145/2851613.2851782

El-khoury, J., Gurdur, D., & Nyberg, M. (2016). A Model-Driven Engineering Approach to Software Tool Interoperability based on Linked Data, *9*(3), 248–259.

EMF-REST Documentation. (2015). Retrieved January 28, 2018, from https://som-research.uoc.edu/tools/emf-rest/documentation.html

Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. *Building*, *54*, 162. https://doi.org/10.1.1.91.2433

Gastwirth, J. L., Gel, Y. R., & Miao, W. (2009). The Impact of Levene's Test of Equality of Variances on Statistical Theory and Practice. *Statistical Science*, *24*(3), 343–360. https://doi.org/10.1214/09-STS301

Giessler, P., Gebhart, M., Sarancin, D., Steinegger, R., & Abeck, S. (2015). Best Practices for the Design of RESTful Web Services. In *ICSEA 2015 : The Tenth International Conference on Software Engineering Advances* (pp. 392–397). Barcelona, Spain.

Haupt, F., Karastoyanova, D., Leymann, F., & Schroth, B. (2014). A model-driven approach for REST compliant services. *Proceedings - 2014 IEEE International Conference on Web Services, ICWS 2014*, 129–136. https://doi.org/10.1109/ICWS.2014.30

Haupt, F., Leymann, F., Scherer, A., & Vukojevic-Haupt, K. (2017). A Framework for the Structural Analysis of REST APIs. In *Proceedings of the 1st IEEE International Conference on Software Architecture, ICSA 2017, 3-7 April 2017, Gothenburg, Sweden* (pp. 55–58). Gothenburg: IEEE Computer Society. https://doi.org/10.1109/ICSA.2017.40

Huhns, M., & Singh, M. P. (2005). Service-Oriented Computing: Key Concepts and Principles. *Proceedings - Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*, *13*(1), 1–7. https://doi.org/10.1016/j.compind.2009.07.006

Hussain, S., Wang, Z., Toure, I. K., & Diop, A. (2013). Web Service Testing Tools : A Comparative Study, *10*(1), 641–647.

Hutchinson, J., Whittle, J., & Rouncefield, M. (2014). Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure. *Science of Computer Programming*, *89*(Part B), 144–161. https://doi.org/10.1016/j.scico.2013.03.017

Koen, P., Ajamian, G., Burkart, R., Clamen, A., Davidson, J., D'Amore, R., … Wagner, K. (2001). Providing Clarity and a Common Language To the "Fuzzy Front End." *Research Technology Management*, *44*(2), 46–55. https://doi.org/Article

Kristopher Sandoval. (2016). What is the Difference Between API Documentation, Specification, and Definition? | Nordic APIs |. Retrieved February 25, 2018, from https://nordicapis.com/difference-api-documentation-specification-definition/

Kumari, S., & Rath, S. K. (2015). Performance comparison of SOAP and REST based Web Services for Enterprise Application Integration. *2015 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, 1656–1660. https://doi.org/10.1109/ICACCI.2015.7275851

Lincke, R., Lundberg, J., & Löwe, W. (2008). Comparing software metrics tools. *Proceedings of the 2008 International Symposium on Software Testing and Analysis - ISSTA '08*, 131. https://doi.org/10.1145/1390630.1390648

Manuel, M. (2011). Avaliação de usabilidade em lojas virtuais de nichos de mercado, *0*, 193. Retrieved from https://repositorio-aberto.up.pt/bitstream/10216/74600/2/31920.pdf

Mendes, L. (2014). Avaliação De Usabilidade Em Sistemas Web - Desktop, 209. Retrieved from https://repositorio-aberto.up.pt/bitstream/10216/74600/2/31920.pdf

Mohagheghi, P., & Dehlen, V. (2008). Where Is the Proof - A Review of Experiences from Applying MDE in Industry.pdf. *ECMDA-FA '08 Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, 432–443.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J., Collet, P., Combemale, B., … Cheng, B. (2014). The relevance of model-driven engineering thirty years from now, *8767*. https://doi.org/10.1007/978-3-319-11653-2

Nguyen, V.-C., Qafmolla, X., & Richta, K. (2014). Domain Specific Language Approach on Model-driven Development of Web Services. *Acta Polytechnica Hungarica*, *11*(8), 121–138. Retrieved from http://www.uni-obuda.hu/journal/Nguyen_Qafmolla_Richta_54.pdf

Oracle. (2018a). Jersey - RESTful Web Services in Java. Retrieved July 1, 2018, from https://jersey.github.io/

Oracle. (2018b). JSR-000370 JavaTM API for RESTful Web Services (JAX-RS) 2.1. Retrieved July 1, 2018, from https://jcp.org/aboutJava/communityprocess/final/jsr370/index.html

Pavan, K. P., Sanjay, A., & Zornitza, P. (2012). Comparing Performance of Web Service Interaction Styles : SOAP vs. REST. *Proceedings of the Conference on Information Systems Applied Research*, 1–24.

Rosenberg, L. H., Stapko, R., & Gallo, A. (2000). Risk-Based Object Oriented Testing. *Measurement*, 1–6. https://doi.org/10.1.1.10.7509

Saaty, T. L. (2008). Decision making with the analytic hierarchy process. *International Journal of Services Sciences*, *1*(1), 83. https://doi.org/10.1504/IJSSCI.2008.017590

Scheidgen, M., Efftinge, S., & Marticke, F. (2016). Metamodeling vs metaprogramming: A case study on developing client libraries for REST APIs. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9764*, 205–216. https://doi.org/10.1007/978-3-319-42061-5_13

Schreibmann, V., & Braun, P. (2014). Design and Implementation of a Model-Driven Approach for Restful APIs. In *5th {IEEE} Germany Student Conference, {IEEE} {GSC} 2014, June 26-27, 2014, Passau, Germany*. Passau. Retrieved from http://www.ieee-student-conference.de/fileadmin/papers/2014/ieeegsc2014_submission_8.pdf

Schreier, S. (2011). Modeling RESTful applications. *Proceedings of the Second International Workshop on RESTful Design - WS-REST '11*, 15. https://doi.org/10.1145/1967428.1967434

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software*, *20*(5), 19–25. https://doi.org/10.1109/MS.2003.1231146

Sharma, H., & Chug, A. (2015). Dynamic metrics are superior than static metrics in maintainability prediction: An empirical case study. *2015 4th International Conference on Reliability, Infocom Technologies and Optimization: Trends and Future Directions, ICRITO 2015*, 2–7. https://doi.org/10.1109/ICRITO.2015.7359354

SOA Work Group. (2016). Service-Oriented Architecture. Retrieved January 18, 2018, from http://www.opengroup.org/soa/source-book/soa/index.htm

Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., & Vijay-Shanker, K. (2010). Towards automatically generating summary comments for Java methods. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering - ASE '10*, 43. https://doi.org/10.1145/1858996.1859006

Sridhara, G., Pollock, L., & Vijay-Shanker, K. (2011). Generating parameter comments and integrating with method summaries. *IEEE International Conference on Program Comprehension*, 71–80. https://doi.org/10.1109/ICPC.2011.28

Stahl, T., Völter, M., Bettin, J., Haase, A., & Helsen, S. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. Wiley.

The Linux Foundation. (2017a). Open API Initiative. Retrieved October 21, 2017, from https://www.openapis.org/

The Linux Foundation. (2017b). The OpenAPI Specification. Retrieved October 22, 2017, from https://github.com/OAI/OpenAPI-Specification

Tihomirovs, J., & Grabis, J. (2016). Comparison of SOAP and REST Based Web Services Using Software Evaluation Metrics. *Information Technology and Management Science*, *19*(1), 92–97. https://doi.org/10.1515/itms-2016-0017

Torchiano, M., Tomassetti, F., Ricca, F., Tiso, A., & Reggio, G. (2013). Relevance, benefits, and problems of software modelling and model driven techniques - A survey in the Italian industry. *Journal of Systems and Software*, *86*(8), 2110–2126. https://doi.org/10.1016/j.jss.2013.03.084

Vasudevan, K. (2017). The Importance of Standardized API Design. Retrieved January 29, 2018, from https://swaggerhub.com/blog/api-design/the-importance-of-standardized-api-design/

W3C. (2004). Web Services Architecture. Retrieved January 9, 2018, from https://www.w3.org/TR/ws-arch/

Weisstein, E. W. (n.d.). Student's t-Distribution. Retrieved from http://mathworld.wolfram.com/Studentst-Distribution.html

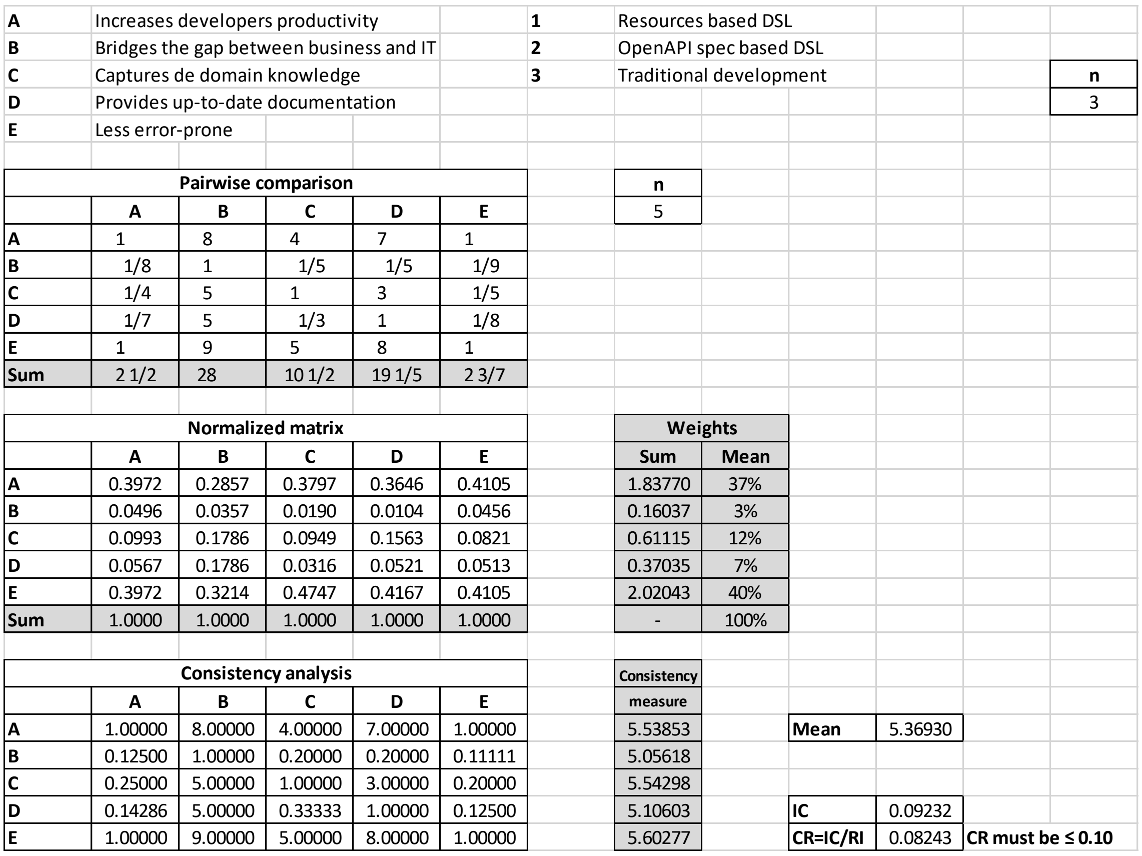
Whittle, J., Hutchinson, J., & Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Software*, *31*(3), 79–85. https://doi.org/10.1109/MS.2013.65

Zion Market Research. (2017). Global API Management Market Worth USD 3,436.16 Million by 2022. Retrieved February 17, 2018, from https://www.zionmarketresearch.com/news/api-management-market

Zolotas, C., Diamantopoulos, T., Chatzidimitriou, K. C., & Symeonidis, A. L. (2017). From requirements to source code: A Model-Driven Engineering approach for RESTful web services. *Automated Software Engineering*, *24*(4), 791–838. https://doi.org/10.1007/s10515-016-0206-x

Appendixes

Appendix A. AHP analysis



A	Increases developers productivity	**1**	Resources based DSL
B	Bridges the gap between business and IT	**2**	OpenAPI spec based DSL
C	Captures de domain knowledge	**3**	Traditional development
D	Provides up-to-date documentation		
E	Less error-prone		

n
3

Pairwise comparison

	A	**B**	**C**	**D**	**E**
A	1	8	4	7	1
B	1/8	1	1/5	1/5	1/9
C	1/4	5	1	3	1/5
D	1/7	5	1/3	1	1/8
E	1	9	5	8	1
Sum	2 1/2	28	10 1/2	19 1/5	2 3/7

n
5

Normalized matrix

	A	**B**	**C**	**D**	**E**
A	0.3972	0.2857	0.3797	0.3646	0.4105
B	0.0496	0.0357	0.0190	0.0104	0.0456
C	0.0993	0.1786	0.0949	0.1563	0.0821
D	0.0567	0.1786	0.0316	0.0521	0.0513
E	0.3972	0.3214	0.4747	0.4167	0.4105
Sum	1.0000	1.0000	1.0000	1.0000	1.0000

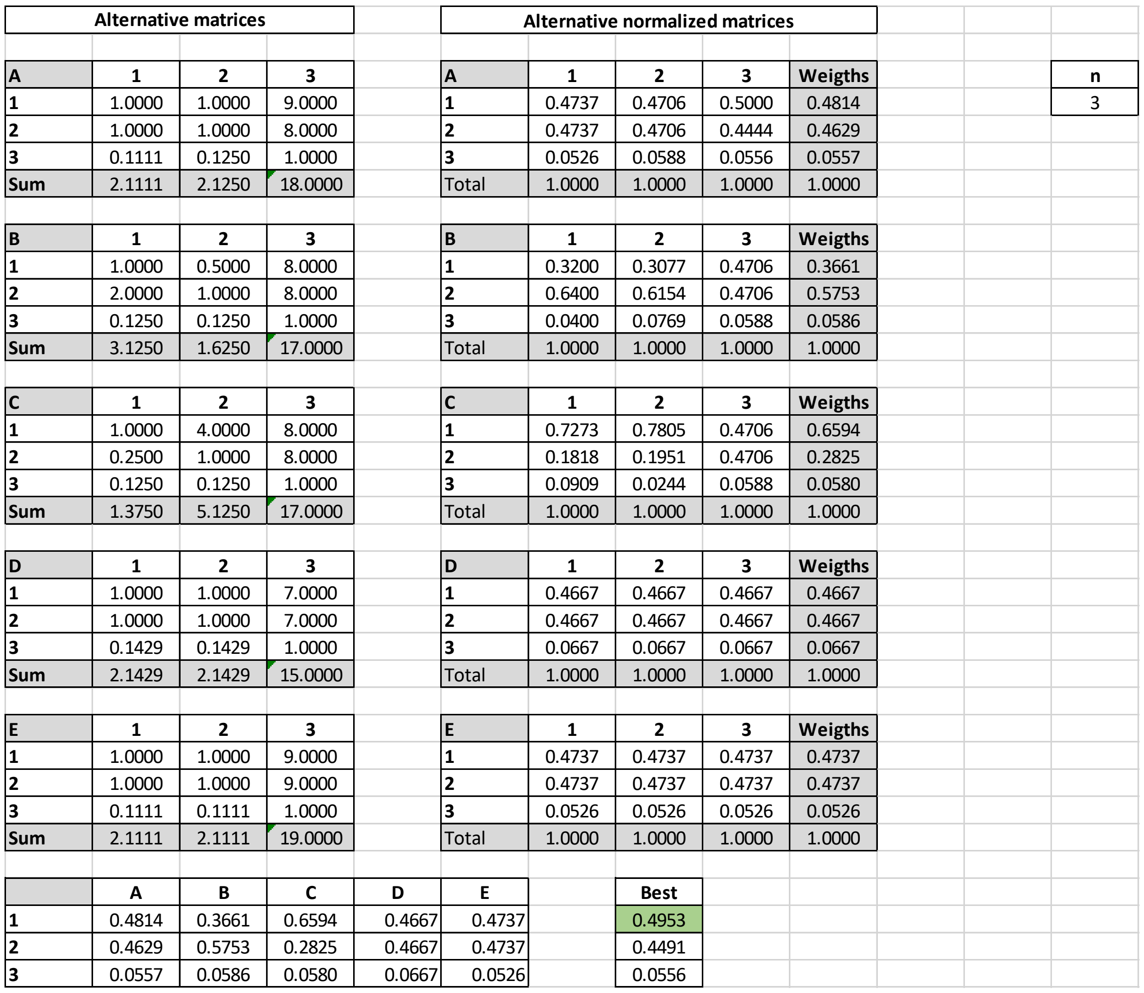
Weights

Sum	**Mean**
1.83770	37%
0.16037	3%
0.61115	12%
0.37035	7%
2.02043	40%
-	100%

Consistency analysis

	A	**B**	**C**	**D**	**E**	**Consistency measure**
A	1.00000	8.00000	4.00000	7.00000	1.00000	5.53853
B	0.12500	1.00000	0.20000	0.20000	0.11111	5.05618
C	0.25000	5.00000	1.00000	3.00000	0.20000	5.54298
D	0.14286	5.00000	0.33333	1.00000	0.12500	5.10603
E	1.00000	9.00000	5.00000	8.00000	1.00000	5.60277

Mean	5.36930	
IC	0.09232	
CR=IC/RI	0.08243	**CR must be ≤ 0.10**



Alternative matrices

A	1	2	3
1	1.0000	1.0000	9.0000
2	1.0000	1.0000	8.0000
3	0.1111	0.1250	1.0000
Sum	2.1111	2.1250	18.0000

B	1	2	3
1	1.0000	0.5000	8.0000
2	2.0000	1.0000	8.0000
3	0.1250	0.1250	1.0000
Sum	3.1250	1.6250	17.0000

C	1	2	3
1	1.0000	4.0000	8.0000
2	0.2500	1.0000	8.0000
3	0.1250	0.1250	1.0000
Sum	1.3750	5.1250	17.0000

D	1	2	3
1	1.0000	1.0000	7.0000
2	1.0000	1.0000	7.0000
3	0.1429	0.1429	1.0000
Sum	2.1429	2.1429	15.0000

E	1	2	3
1	1.0000	1.0000	9.0000
2	1.0000	1.0000	9.0000
3	0.1111	0.1111	1.0000
Sum	2.1111	2.1111	19.0000

Alternative normalized matrices

A	1	2	3	Weigths
1	0.4737	0.4706	0.5000	0.4814
2	0.4737	0.4706	0.4444	0.4629
3	0.0526	0.0588	0.0556	0.0557
Total	1.0000	1.0000	1.0000	1.0000

B	1	2	3	Weigths
1	0.3200	0.3077	0.4706	0.3661
2	0.6400	0.6154	0.4706	0.5753
3	0.0400	0.0769	0.0588	0.0586
Total	1.0000	1.0000	1.0000	1.0000

C	1	2	3	Weigths
1	0.7273	0.7805	0.4706	0.6594
2	0.1818	0.1951	0.4706	0.2825
3	0.0909	0.0244	0.0588	0.0580
Total	1.0000	1.0000	1.0000	1.0000

D	1	2	3	Weigths
1	0.4667	0.4667	0.4667	0.4667
2	0.4667	0.4667	0.4667	0.4667
3	0.0667	0.0667	0.0667	0.0667
Total	1.0000	1.0000	1.0000	1.0000

E	1	2	3	Weigths
1	0.4737	0.4737	0.4737	0.4737
2	0.4737	0.4737	0.4737	0.4737
3	0.0526	0.0526	0.0526	0.0526
Total	1.0000	1.0000	1.0000	1.0000

n
3

	A	B	C	D	E
1	0.4814	0.3661	0.6594	0.4667	0.4737
2	0.4629	0.5753	0.2825	0.4667	0.4737
3	0.0557	0.0586	0.0580	0.0667	0.0526

Best
0.4953
0.4491
0.0556

Appendix B. Resource-oriented grammar

```
grammar com.tmdei.xtext.dsl.RsrcDsl with org.eclipse.xtext.common.Terminals

generate rsrcDsl "http://www.tmdei.com/xtext/dsl/RsrcDsl"

Model:
  meta=Meta
  (resources+=Resource+)
  (enums+=Enum*);

Meta:
  'project:' project=STRING
  'version:' version=STRING
  'basePath:' basePath=Url
  'server:' server=ServerInfo
  ('mediaType:' mediaType+=MediaType+)?
  ('basePackage:' package=QualifiedNameWithWildcard)?;

ServerInfo:
  '{'
  'description:' description=STRING
  'url:' url=Url
  '}';

Resource:
  'resource' (abstract?='abstract')? name=ID ('table' table=ID)? (cache?='cache')? (extends?=Parent)?
  '{'
  (attributes+=Attribute)*
  (contains+=Composite)*
  (relations+=Relation)*
  (references+=Reference)*
  (customActions+=Action)*
  (actuatedBy+=Trigger)*
  (httpMethods+=HTTPMethod)*
  '}';

Parent:
  'extends' parent=[Resource];

Composite:
  'contains' name=ValidID type=[Resource] (multiple?='*')?;

Reference:
  'reference' name=[Relation|QualifiedName] ('column' column=ID)?;

Relation:
  'relation' name=ValidID type=[Resource|QualifiedName] (multiple?='*')? ('method' '[' actions+=HTTPMethod+ ']')?;

Attribute:
  'attribute' name=ValidID type=DataType (multiple?='*')? ('column' column=ID)? (mandatory?='mandatory')?;

Action:
  'action' name=ID 'on' resource=[Resource] 'over' attribute=[Attribute|QualifiedName]
  ('method' '[' actions+=HTTPMethod+ ']');

Trigger:
  'actuated' 'by' name=[Action|QualifiedName];

Url:
  url=STRING;

DataType:
  Type;

MediaType:
  JSON | XML;

Type:
  RsrcString | Integer | Long | BigDecimal | Calendar | Boolean | EnumType;

JSON:
  {JSON} 'Json';

XML:
  {XML} 'XML';

RsrcString:
  {RsrcString} 'String';

Integer:
  {Integer} 'Integer';
```

```
Long:
  {Long} 'Long';

BigDecimal:
  {BigDecimal} 'BigDecimal';

Calendar:
  {Calendar} 'Calendar';

Boolean:
  {Boolean} 'Boolean';

EnumType:
  'Enum' '(' enumSubType=[Enum] ')';

Enum:
  'enum' name=ValidID '{'
  (attributes+=EnumAttribute)+
  '}';

EnumAttribute:
  'name' name=ValidID
  ('value' value=INT)?;

enum HTTPMethod:
  get='GET' | post='POST' | put='PUT' | delete='DELETE' | patch='PATCH';

QualifiedName:
  ID ('.' ID)*;

QualifiedNameWithWildcard:
  QualifiedName '.*'?;

ValidID:
  ID | Keyword;

Keyword:
  'name';
```

Appendix C. Resource-oriented model example

```
project: "Nutrition Clinic"
version: "1.0"
basePath: "nutrition.clinic"
server: {
  description: "Nutrition clinic development server"
  url: "http://nutrition.clinic.dev.com"
}
mediaType: Json XML
basePackage: com.nutrition.clinic

resource abstract User table user {
  attribute firstName String column First_Name
  attribute lastName String column Last_Name
  attribute email String column Email
  attribute birthDate Calendar column Birth_Date

  contains address Address
}

resource Client table client extends User {
  attribute phoneNumber String column phoneNumber

  relation favoriteFoods Food* method [GET POST DELETE]
  relation dislikedFoods Food* method [GET POST DELETE]

  reference Professional.patients

  action rate on Professional over Professional.rating method [POST]

  GET POST PUT
}

resource Professional table professional extends User {
  attribute category Enum(Category) column Category
  attribute rating Integer column Rating

  relation patients Client* method [GET]

  actuated by Client.rate

  GET
}

resource Food table food {
  attribute name String column Name
  attribute calories Double column Calories
  attribute fats Double column Fats
  attribute proteins Double column Proteins
  attribute hydrates Double column Hydrates

  reference Client.favoriteFoods
  reference Client.dislikedFoods
}

resource Address table address {
  attribute firstLine String column first_line
  attribute lastLine String column last_line
  attribute zipCode String column zip_code

  NONE
}

enum Category {
  name Medic
  value 1

  name Nutritionist
  value 2
}

enum Goal {
  name WeighGain
  value 1

  name WeightLoss
  value 2

  name WeightMaintenance
  value 3
}
```

Appendix D. OAS-oriented grammar

```
grammar com.tmdei.xtext.dsl.OasDsl with org.eclipse.xtext.common.Terminals

generate oasDsl "http://www.tmdei.com/xtext/dsl/OasDsl"

OpenAPIObject:
  '{'
  documentOpenAPIVersion=OpenAPIVersionField
  infoField=InfoField
  (serversField=ServersField)?
  pathsField=PathsField
  (componentsField=ComponentsField)?
  (securityField=SecurityField)?
  (tagsField=TagsField)?
  '}';

OpenAPIVersionField:
  '"openapi":' openApi=STRING ',';

InfoField:
  '"info":' info=InfoObject;

ServersField:
  '"servers":' ('[' servers+=ServerObject+ (']' | '],') | servers+=ServerObject) ','?;

PathsField:
  '"paths":' '{' paths+=PathsObject+ ('}' | '},');

ComponentsField:
  '"components":' components=ComponentsObject ','?;

SecurityField:
  '"security":' '[' security+=SecurityRequirementObject (']' | '],');

TagsField:
  '"tags":' '[' tags+=TagObject ']';

InfoObject:
  '{'
  title=TitleField
  (description=DescriptionField)?
  (termsOfService=TermsOfServiceField)?
  (contact=ContactField)?
  (license=LicenseField)?
  (version=VersionField)?
  ('}' | '},');

TermsOfServiceField:
  '"termsOfService":' termsOfService=STRING ','?;

ContactField:
  '"contact":' contact=ContactObject;

LicenseField:
  '"license":' license=LicenseObject;

VersionField:
  '"version":' version=STRING;

ContactObject:
  {ContactObject}
  '{'
  (name=NameField)?
  (url=UrlField)?
  (email=EmailField)?
  ('}' | '},');

LicenseObject:
  {LicenseObject}
  '{'
  (name=NameField)?
  (url=UrlField)?
  ('}' | '},');

TitleField:
  '"title":' title=STRING ','?;

DescriptionField:
  '"description":' description=STRING ','?;

NameField:
  '"name":' name=STRING ','?;
```

```
UrlField:
  '"url":' url=AbsoluteUrl ','?;

EmailField:
  '"email":' email=STRING ','?;

SummaryField:
  '"summary":' summary=STRING ','?;

ServerObject:
  '{'
  url=UrlField
  (description=DescriptionField)?
  (variables=ServerVariablesField)?
  ('}' | '},');

ServerVariablesField:
  '"variables":' '{' variables+=ServerVariableMap+ ('}' | '},');

ServerVariableMap:
  STRING ':' serverVariable=ServerVariableObject;

ServerVariableObject:
  serverVariableEnum=EnumString
  (default=DefaultString)?
  (description=DescriptionField)?;

DefaultString:
  '"default":' default+=STRING+ ','?;

EnumString:
  '"enum":' '[' ^enum+=STRING+ (']' | '],');

PathsObject:
  url=Url ':' '{'
  paths+=PathItemObject+
  ('}' | '},');

ComponentsObject hidden(WS):
  {ComponentsObject}
  '{'
  (schemas=ComponentsSchemasField)?
  (responses=ComponentsResponsesField)?
  (parameters=ComponentsParametersField)?
  (examples=ComponentsExamplesField)?
  (requestBodies=ComponentsRequestBodiesField)?
  (headers=ComponentsHeadersField)?
  (securitySchemes=ComponentsSecuritySchemesField)?
  (links=ComponentsLinksField)?
  (callbacks=ComponentsCallbacksField)?
  ('}' | '},');

ComponentsSchemasField:
  '"schemas":' '{' schemas+=ComponentsSchemaMap+ ('}' | '},');

ComponentsResponsesField:
  '"responses":' '{' responses+=ComponentsResponseMap+ ('}' | '},');

ComponentsParametersField:
  '"parameters":' '{' parameters+=ComponentsParameterMap+ ('}' | '},');

ComponentsExamplesField:
  '"examples":' '{' examples+=ComponentsExampleMap+ ('}' | '},');

ComponentsRequestBodiesField:
  '"requestBodies":' '{' requestBodies+=ComponentsRequestBodiesMap+ ('}' | '},');

ComponentsHeadersField:
  '"headers":' '{' headers+=ComponentsHeadersMap+ ('}' | '},');

ComponentsSecuritySchemesField:
  '"securitySchemes":' '{' securitySchemes+=ComponentsSecuritySchemesMap+ ('}' | '},');

ComponentsLinksField:
  '"links":' '{' links+=ComponentsLinksMap+ ('}' | '},');

ComponentsCallbacksField:
  '"callbacks":' '{' callbacks+=ComponentsCallbacksMap+ ('}' | '},');

ComponentsSchemaMap:
  name=STRING ':' (schemaObject=SchemaObject | referenceObject=ReferenceObject);

ComponentsResponseMap:
  name=STRING ':' (responseObject=STRING | referenceObject=ReferenceObject);
```

```
ComponentsParameterMap:
  name=STRING ':' (parameterObject=ParameterObject | referenceObject=ReferenceObject);

ComponentsExampleMap:
  name=STRING ':' (exampleObject=STRING | referenceObject=ReferenceObject);

ComponentsRequestBodiesMap:
  name=STRING ':' (requestBodyObject=STRING | referenceObject=ReferenceObject);

ComponentsHeadersMap:
  name=STRING ':' (headerObject=STRING | referenceObject=ReferenceObject);

ComponentsSecuritySchemesMap:
  name=STRING ':' (securitySchemesObject=STRING | referenceObject=ReferenceObject);

ComponentsLinksMap:
  name=STRING ':' (linkObject=STRING | referenceObject=ReferenceObject);

ComponentsCallbacksMap:
  name=STRING ':' (callbackObject=STRING | referenceObject=ReferenceObject);

SecurityRequirementObject:
  STRING;

TagObject:
  STRING;

PathItemObject:
  (=> httpMethod+=HttpMethod ':' '{' operation+=OperationObject ('}' | '},'))+
  ('"$ref":' '{' ref=PathItemObject ('}' | '},'))?
  (summary=SummaryField)?
  (description=DescriptionField)?
  (serversField=ServersField)?
  (parameters=ParametersField)?;

OperationObject:
  (tags=TagsStringField)?
  (summary=SummaryField)?
  (description=DescriptionField)?
  (externalDocumentation=ExternalDocumentationField)?
  (operationId=OperationIdField)?
  (parameters=ParametersField)?
  (requestBody=RequestBodyField)?
  (responses=ResponsesField)
  (deprecated=DeprecatedField)?
  (serversField=ServersField)?
  ('"callbacks":' callbacks=STRING ','?)?
  ('"security":' security=STRING ','?)?;

TagsStringField:
  '"tags":' '[' tags+=STRING (',' tags+=STRING)* (']' | '],');

ExternalDocumentationField:
  '"externalDocs":' externalDocumentation=ExternalDocumentationObject ','?;

ExternalDocumentationObject:
  '{'
  (description=DescriptionField)?
  url=UrlField
  ('}' | '},');

OperationIdField:
  '"operationId":' operationId=STRING ','?;

ParametersField:
  '"parameters":' '[' (parameters+=ParameterObject+ | referenceObject+=ReferenceObject+) (']' | '],');

RequestBodyField:
  '"requestBody":' requestBody=RequestBodyObject;

ResponsesField:
  '"responses":'
  '{'
  defaultResponse=ResponseMapDefault
  responses+=ResponseMap+
  ('}' | '},');

ParameterObject:
  '{'
  name=NameField
  in=InField
  (description=DescriptionField)?
  (required=RequiredBooleanField)
  (deprecated=DeprecatedField)?
  (allowEmptyValue=AllowEmptyValueField)?
  (style=StyleField)?
  (explode=ExplodeField)?
  (allowReserved=AllowReservedField)?
  (schema=ParameterSchemaField)?
  (('"example":' STRING) | ('"examples":' STRING))?
  (content=ContentField)?
  ('}' | '},');

InField:
  '"in":' in=STRING ','?;

RequiredBooleanField:
  '"required":' required=Boolean ','?;

DeprecatedField:
  '"deprecated":' deprecated=Boolean ','?;

AllowEmptyValueField:
  '"allowEmptyValue":' allowEmptyValue=Boolean ','?;

StyleField:
  '"style":' style=STRING ','?;

ExplodeField:
  '"explode":' explode=Boolean ','?;

AllowReservedField:
  '"allowReserved":' allowReserved=Boolean ','?;

ParameterSchemaField:
  ('"schema":' schema=SchemaParameterObject | reference=ReferenceObject) ','?;

ContentField:
  '"content":' '{' content+=ContentMap+ ('}' | '},');

ContentMap:
  name=STRING ':' content=MediaTypeObject ','?;

RequestBodyObject:
  '{'
  (description=DescriptionField)?
  (content=ContentField)
  (required=RequiredBooleanField)?
  ('}' | '},');

ResponseObject:
  {ResponseObject}
  '{'
  (description=DescriptionField)?
  (headers=ResponseHeaderField)?
  (content=ResponseContentField)?
  (links=ResponseLinkField)?
  ('}' | '},');

ResponseHeaderField:
  '"headers":' '{' responseHeaderObject=ResponseHeaderObject ('}' | '},');

ResponseContentField:
  '"content":' '{' responseMediaTypeObject=ResponseMediaTypeObject ('}' | '},');

ResponseLinkField:
  '"links":' '{' responseLinkObject=ResponseLinkObject ('}' | '},');

ResponseMapDefault:
  '"default":' response=ResponseObject;

ResponseMap:
  name=STRING ':' response=ResponseObject;

ResponseHeaderObject:
  name=STRING ':' (headerObject=HeaderObject | referenceObject=ReferenceObject);

ResponseMediaTypeObject:
  name=STRING ':' (mediaTypeObject=MediaTypeObject | referenceObject=ReferenceObject);

ResponseLinkObject:
  name=STRING ':' (linkObject=LinkObject | referenceObject=ReferenceObject);

HeaderObject:
  {HeaderObject}
  '{'
  (description=DescriptionField)?
  (required=RequiredBooleanField)?
  (deprecated=DeprecatedField)?
  (allowEmptyValue=AllowEmptyValueField)?
  (style=StyleField)?
  (explode=ExplodeField)?
  (allowReserved=AllowReservedField)?
  (schema=ParameterSchemaField)?
  ('"examples":' examples=STRING ','?)? ('}' | '},');

MediaTypeObject:
  {MediaTypeObject}
  '{'
  ('"schema":' (schemaObject=SchemaObject | referenceObject=ReferenceObject))?
  (('"example":' STRING) | ('"examples":' STRING))?
  ('"encoding":' encodingObject=EncodingObject)?
  ('}' | '},');

LinkObject:
  {LinkObject}
  '{'
  ('"operationRef":' STRING)?
  ('"operationId":' STRING)?
  ('"parameters":' STRING)?
  ('"requestBody":' STRING)?
  ('"description":' STRING)?
  ('"server":' STRING)?
  ('}' | '},');

EncodingObject:
  '{'
  ('"contentType":' STRING)?
  ('"headers":' STRING)?
  ('"style":' STRING)?
  ('"explode":' Boolean)?
  ('"allowReserved":' Boolean)?
  ('}' | '},');

SchemaParameterObject:
  (referenceObject=ReferenceObject | schemaObject=SchemaObject);

SchemaObject:
  {SchemaObject}
  '{'
  (title=TitleField)?
  (description=DescriptionField)?
  ('"multipleOf":' multipleOf=INT ','?)?
  ('"maximum":' maximum=INT ','?)?
  ('"exclusiveMaximum":' exclusiveMaximum=INT ','?)?
  ('"minimum":' minimum=INT ','?)?
  ('"exclusiveMinimum":' exclusiveMinimum=INT ','?)?
  ('"maxLength":' maxLength=INT ','?)?
  ('"minLength":' minLength=INT ','?)?
  ('"pattern":' pattern=STRING ','?)?
  ('"maxItems":' maxItems=INT ','?)?
  ('"minItems":' minItems=INT ','?)?
  ('"uniqueItems":' uniqueItems=Boolean ','?)?
  ('"maxProperties":' maxProperties=INT ','?)?
  ('"minProperties":' minProperties=INT ','?)?
  ('"required":' '[' required+=STRING (',' required+=STRING)* (']' | '],'))?
  ('"enum":' '[' ^enum+=STRING (',' ^enum+=STRING)* (']' | '],'))?
  ('"type":' type=STRING ','?)?
  ('"allOf":' allOff=SchemaObject ','?)?
  ('"oneOf":' oneOf=SchemaObject ','?)?
  ('"anyOf":' anyOf=SchemaObject ','?)?
  ('"not":' not=SchemaObject ','?)?
  ('"items":' ('[' items+=SchemaObjectOrReference+ ']' | items+=SchemaObjectOrReference) ','?)?
  ('"properties":' ('{' properties+=ComponentsSchemaMap+ ('}' | '},')))?
  ('"additionalProperties":' additionalProperties=STRING ','?)?
  ('"format":' format=STRING ','?)?
  ('"default":' default=STRING ','?)?
  ('"nullable":' nullable=Boolean ','?)?
  ('"discriminator":' discriminator=DiscriminatorObject ','?)?
  ('"readOnly":' readOnly=Boolean ','?)?
  ('"writeOnly":' writeOnly=Boolean ','?)?
  ('"xml":' xml=STRING ','?)?
  ('"externalDocs":' externalDocs=STRING ','?)?
  ('"example":' example=STRING ','?)?
  ('"deprecated":' deprecated=Boolean ','?)?
  ('}' | '},');

SchemaObjectOrReference:
  schemaObject=SchemaObject | referenceObject=ReferenceObject;

ReferenceObject:
  ('{' '"$ref":' ref=STRING ('}' | '},'));

DiscriminatorObject:
  '{' '"propertyName":' STRING ','? '"mapping":' '{' mapping+=MapStringString+ ','? '}' ('}' | '},');

MapStringString:
  key=STRING ':' value=STRING;

Boolean:
  'true' | 'false';

enum HttpMethodEnum:
  get | put | post | delete | options | head | patch | trace;

enum HttpResponse:
  ok | created | accepted;

enum StyleValues:
  matrix | label | form | simple | spaceDelimited | pipeDelimited | deepObject;

enum OASdataTypes:
  array | integer | long | float | double | string | byte | binary | boolean | date | dateTime | password | number |
object;

enum OASdataFormats:
  int32 | int64 | float | double | byte | bynary | date | date_time | password | uuid;

terminal HTTP_STATUS_CODE:
  '0'..'9' '0'..'9' '0'..'9';

HttpMethod:
  {HttpMethod} httpMethod=STRING;

Url:
  url=STRING;

RelativeUrl:
  relativeUrl=STRING;

AbsoluteUrl:
  absoluteUtl=STRING;
```

Appendix E. OAS-oriented model example

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Nutrition Clinic",
    "license": {
      "name": "MIT"
    },
    "version": "1.0.0"
  },
  "servers": [{
      "url": "http://nutrition.clinic.dev.com",
      "description": "Nutrition clinic development server"
    }
  ],
  "paths": {
    "/clients": {
      "get": {
        "tags": [
          "clients"
        ],
        "summary": "List all clients",
        "operationId": "listClients",
        "responses": {
          "default": {
            "description": "Unexpected error",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Error"
                }
              }
            }
          },
          "200": {
            "description": "An paged array of clients",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Clients"
                }
              }
            }
          }
        }
      },
      "post": {
        "tags": [
          "clients"
        ],
        "summary": "Create a client",
        "operationId": "createClients",
        "requestBody": {
          "description": "Client to add to the system",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/Client"
              }
            }
          }
        },
        "responses": {
          "default": {
            "description": "unexpected error",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/Error"
                }
              }
            }
          },
   "201": {
    "description": "The created client",
    "content": {
     "application/json": {
      "schema": {
       "$ref": "#/components/schemas/Client"
      }
     }
    }
   }
  }
 }
},
"/clients/{clientId}": {
 "get": {
  "tags": [
   "clients"
  ],
  "summary": "Info for a specific client",
  "operationId": "showClientById",
  "parameters": [{
    "name": "clientId",
    "in": "path",
    "description": "The id of the client to retrieve",
    "required": true,
    "schema": {
     "type": "string"
    }
   }
  ],
  "responses": {
   "default": {
    "description": "Unexpected error",
    "content": {
     "application/json": {
      "schema": {
       "$ref": "#/components/schemas/Error"
      }
     }
    }
   },
   "200": {
    "description": "Expected response to a valid request",
    "content": {
     "application/json": {
      "schema": {
       "$ref": "#/components/schemas/Client"
      }
     }
    }
   }
  }
 },
 "put": {
  "tags": [
   "clients"
  ],
  "summary": "Update a client",
  "operationId": "updateClients",
  "parameters": [{
    "name": "clientId",
    "in": "path",
    "description": "The id of the client to retrieve",
    "required": true,
    "schema": {
     "type": "string"
    }
   }
  ],
  "requestBody": {
   "description": "Client to update",
   "content": {
    "application/json": {
     "schema": {
      "$ref": "#/components/schemas/Client"
```

```
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "201": {
        "description": "The updated client",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Client"
            }
          }
        }
      }
    }
  },
  "delete": {
    "tags": [
      "clients"
    ],
    "summary": "Deletes a client",
    "operationId": "deleteClient",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "201": {
        "description": "The deleted client",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Client"
            }
          }
        }
      }
    }
  }
},
"/clients/{clientId}/favoriteFoods": {
  "get": {
    "tags": [
      "clients",
      "foods"
    ],
    "summary": "Info for a specific client favorite foods",
    "operationId": "showFavoriteFoodsByUserId",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve the favorites foods from",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Foods"
            }
          }
        }
      }
    }
  },
  "post": {
    "tags": [
      "clients",
      "foods"
    ],
    "summary": "Add food to a specific client favorite foods",
    "operationId": "showDislikedFoodsByUserId",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to add the favorite foods to",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "requestBody": {
      "description": "Food to add to the favorives list",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Food"
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
```

```
            "schema": {
              "$ref": "#/components/schemas/Food"
            }
          }
        }
      }
    }
  }
},
"/clients/{clientId}/dislikedFoods": {
  "get": {
    "tags": [
      "clients",
      "foods"
    ],
    "summary": "Info for a specific client disliked foods",
    "operationId": "showDislikedFoodsByUserId",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Foods"
            }
          }
        }
      }
    }
  },
  "post": {
    "tags": [
      "clients",
      "foods"
    ],
    "summary": "Info for a specific client disliked foods",
    "operationId": "showDislikedFoodsByUserId",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "requestBody": {
      "description": "Food to add to the disliked list",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Food"
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Food"
            }
          }
        }
      }
    }
  }
},
"/clients/{clientId}/professionals/{professionalId}/rate/{rating}": {
  "post": {
    "tags": [
      "clients",
      "professionals"
    ],
    "summary": "Rate a professional",
    "operationId": "professionalRatingCreate",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client making the rating",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "professionalId",
        "in": "path",
        "description": "The id of the professional to rate",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "rating",
        "in": "path",
        "description": "The rating",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
```

```
            "schema": {
              "$ref": "#/components/schemas/Professional"
            }
          }
        }
      }
    }
  },
  "put": {
    "tags": [
      "clients",
      "professionals"
    ],
    "summary": "Update a professional rating",
    "operationId": "professionalRatingUpdate",
    "parameters": [{
        "name": "clientId",
        "in": "path",
        "description": "The id of the client making the rating",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "professionalId",
        "in": "path",
        "description": "The id of the professional to update rating",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "rating",
        "in": "path",
        "description": "The rating",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Professional"
            }
          }
        }
      }
    }
  }
},
"/professionals": {
  "get": {
    "tags": [
      "professionals"
    ],
    "summary": "List all professionals",
    "operationId": "listProfessionals",
    "responses": {
      "default": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "An paged array of professionals",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Professionals"
            }
          }
        }
      }
    }
  },
  "post": {
    "tags": [
      "professionals"
    ],
    "summary": "Create a professional",
    "operationId": "createProfessionals",
    "requestBody": {
      "description": "Professional to create",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Professional"
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "201": {
        "description": "The created professional",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Professional"
            }
          }
        }
      }
    }
  }
},
"/professionals/{professionalId}/clients/{clientId}": {
  "get": {
    "tags": [
      "professionals",
      "clients"
    ],
    "summary": "Professional clients",
    "operationId": "getProfessionalClient",
    "parameters": [{
        "name": "professionalId",
        "in": "path",
        "description": "The id of the professional",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Client"
            }
          }
        }
      }
    }
  },
  "post": {
    "tags": [
      "professionals",
      "clients"
    ],
    "summary": "Professional clients",
    "operationId": "getProfessionalClient",
    "parameters": [{
        "name": "professionalId",
        "in": "path",
        "description": "The id of the professional",
        "required": true,
        "schema": {
          "type": "string"
        }
      }, {
        "name": "clientId",
        "in": "path",
        "description": "The id of the client to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
```

```
              "$ref": "#/components/schemas/Client"
            }
          }
        }
      }
    }
  }
},
"/foods": {
  "get": {
    "tags": [
      "foods"
    ],
    "summary": "List all foods",
    "operationId": "listFoods",
    "responses": {
      "default": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "An paged array of Foods",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Foods"
            }
          }
        }
      }
    }
  },
  "post": {
    "tags": [
      "foods"
    ],
    "summary": "Create a food",
    "operationId": "createFoods",
    "requestBody": {
      "description": "Food to create",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Food"
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "201": {
        "description": "The created food",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Food"
            }
          }
        }
      }
    }
  }
},
"/foods/{foodId}": {
  "get": {
    "tags": [
      "foods"
    ],
    "summary": "Info for a specific food",
    "operationId": "showFoodById",
    "parameters": [{
        "name": "foodId",
        "in": "path",
        "description": "The id of the food to retrieve",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "responses": {
      "default": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "200": {
        "description": "Expected response to a valid request",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Food"
            }
          }
        }
      }
    }
  },
  "put": {
    "tags": [
      "foods"
    ],
    "summary": "Update a food",
    "operationId": "updateFoods",
    "parameters": [{
        "name": "foodId",
        "in": "path",
        "description": "The id of the food to update",
        "required": true,
        "schema": {
          "type": "string"
        }
      }
    ],
    "requestBody": {
      "description": "Food to update",
      "content": {
        "application/json": {
          "schema": {
            "$ref": "#/components/schemas/Food"
          }
        }
      }
    },
    "responses": {
      "default": {
        "description": "unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Error"
            }
          }
        }
      },
      "201": {
        "description": "The updated food",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/Food"
            }
          }
        }
      }
    }
  }
}
},
"components": {
"schemas": {
  "User": {
    "type": "object",
    "required": [
      "id",
      "email"
    ],
    "properties": {
      "id": {
        "type": "integer",
        "format": "int64"
      },
      "firstName": {
        "type": "string"
      },
      "lastName": {
        "type": "string"
      },
      "email": {
        "type": "string"
      },
      "birthDate": {
        "type": "date"
      },
      "address": {
        "type": "object",
        "items": {
          "$ref": "#/components/schemas/Address"
        }
      }
    }
  },
  "Client": {
    "allOf": [{
        "$ref": "#/components/schemas/User"
      }, {
        "type": "object",
        "properties": {
          "phoneNumber": {
            "type": "string"
          },
          "favoriteFoods": {
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/Food"
            }
          },
          "dislikedFoods": {
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/Food"
            }
          },
          "goal": {
            "type": "string",
            "enum": [
              "gain weight",
              "lose weight",
              "maintain weight"
            ]
          }
        }
      }
    ]

  },
  "Professional": {
    "allOf": [{
        "$ref": "#/components/schemas/User"
      }, {
        "type": "object",
        "properties": {
          "rating": {
            "type": "double"
          },
          "patients": {
            "type": "array",
            "items": {
              "$ref": "#/components/schemas/Clients"
            }
          },
          "category": {
            "type": "string",
            "enum": [
              "nutritionist",
              "medic"
            ]
          }
        }
      }
    ]
  },
  "Food": {
    "type": "object",
    "required": [
      "name"
    ],
    "properties": {
      "properties": {
        "name": {
          "type": "string"
        },
        "fats": {
          "type": "double"
        },
        "proteins": {
          "type": "double"
        },
        "hydrates": {
          "type": "double"
        },
        "calories": {
          "type": "double"
        }
      }
    }
  },
  "Address": {
    "required": [
      "zipCode"
    ],
    "properties": {
      "firstLine": {
        "type": "string"
      },
      "secondLine": {
        "type": "string"
      },
      "zipCode": {
        "type": "string"
```

```
          }
        }
      },
      "Clients": {
        "type": "array",
        "items": {
          "$ref": "#/components/schemas/Client"
        }
      },
      "Professionals": {
        "type": "array",
        "items": {
          "$ref": "#/components/schemas/Professional"
        }
      },
      "Foods": {
        "type": "array",
        "items": {
          "$ref": "#/components/schemas/Food"
        }
      },
      "Error": {
        "type": "object",
        "required": [
          "code",
          "message"
        ],
        "properties": {
          "code": {
            "type": "integer",
            "format": "int32"
          },
          "message": {
            "type": "string"
          }
        }
      }
    }
  }
}
```