REST API Auto Generation Using Model-Based Approach

التطوير الآلي لواجهات التطبيق البرمجية باستخدام تقنية "ريست" بالإرتكاز على نماذج الكيانات

Master Thesis in Software Engineering

Salah Hussein

February 03, 2020

Advisors: Dr. Samer Zein
Thesis committee: Prof. Adel Taweel and Dr. Ahmed Tamrawi

This Thesis was submitted in partial fulfillment of the requirements for the Master's Degree in Software Engineering From the Faculty of Graduate Studies, Birzeit University, Palestine.

Declaration

This thesis is an account of research undertaken between February 2019 and February 2020 by Salah Hussein, with registration number of 1155076, at The Program of Software Engineering, Faculty of Engineering and Technology, Birzeit University, Palestine.

Approved by the thesis committee:

Dr. Samer Zein, Birzeit University

____

Dr. Adel Taweel, Birzeit University

____

Dr. Ahmed Tamrawi, Birzeit University

____

Date approved: ______________________________

Abstract

Demands on software applications increase rapidly day by day, especial on mobile apps, where development environment is known for being very rapid, with short-time-to-market and fierce competition. Most applications rely on web services as a REST web APIs. In the recent years, usage of web services demonstrates exponential increment, which refers to huge increment on connections after considering Internet of Things (IoT), and also due to vast cloud usage that relies on SOA. But REST web services development requires much time and efforts, and it is not easy to understand. Therefore, backbone development is a very critical task, and it needs a set of various skills and vast knowledge to be developed, such as experience in SQL Database engines, tiers architecture, application servers, and REST APIs. This study aims to improve productivity, maintainability, and easy-to-build REST web services. Introduced approach based on a framework that abstracts layers in model, data access, business logic, and web APIs. Traditional code generation was avoided due its limitations. Based on the implemented survey and experiment, results over the proposed approach shows a significant improvements on productivity and easy to use, where development time was reduced to less than the fourth. Furthermore, proposed architecture facilitates maintainability, which is highly expected in results of survey.

Keywords: web services, REST APIs, SOA, code generation, backbone development, software framework, productivity, maintainability, and learnability.

الملخّص

يشهد الطلب على برمجة الانظمة و التطبيقات تزايدا مستمرا في الآونة الأخيرة، و خصوصا في سوق الهواتف الذكية، حيث يتسابق المزودون بقوة لتقديم الافضل. و غالبا ما يعتمد بناء التطبيقات و الانظمة على واجهات التطبيق البرمجية للويب، و خاصة نمط "ريست"، فقد اظهرت الاحصائيات الاخيرة تزيد كبير في بناء هذا النمط و استخدامه. ان الانتشار الواسع لإنترنت الأشياء و الحوسبة السحابية كان له دور كبير في زيادة استخدامات خدمات الويب في بناء التطبيقات، حيث تزيد الحاجة للروابط المتعلقة بالأشياء لتطبيق إنترنت الاشياء، و من جهة أخرى فان الحوسبة السحابية بحد ذاتها مبنية على مبدأ بناء خدمات الويب. ان بناء خدمات الويب "ريست" ليس بالامر السهل، فانه يحتاج الى وقت و جهود كبيرة، فان تعقيدات بناء هذه التقنية يتطلب من المطورين مهارات و معارف واسعة، مثل مهارات تصميم سجلات البيانات و الخوادم المركزية بنمط الطبقات. تهدف هذه الدراسة لتحسين الانتاجية و امكانية الصيانة، و ذلك بتسهيل عملية بناء خدمات الويب، حيث ان المنهج المطروح يعتمد اطار عمل يختزل بناء الكيانات، الوصول للبيانات، منطق العمل، و واجهات التطبيق البرمجية للويب. ان المنهج المقدّم لا يعتمد على الكتابة الآلية للكود لما في ذلك من انعكاسات سلبية عند القيام بعملية الصيانة و التطوير. أظهرت نتائج التجربة و الاستبيان تحسن كبير على الانتاجية و سهولة الاستخدام للمنهج المعروض في هذا البحث، حيث تم تقليل الوقت اللازم لاقل من ربع الوقت المطلوب باستخدام اساليب و أطُر اخرى، كما ان نتائج الاستبيان أكدت ان هذا المنهج يسهل عملية الصيانة و التطوير.

Acknowledgment

I would first like to thank my thesis advisor Dr. Samer Zein of the Engineering and Technology Faculty at Birzeit University. The door to Dr. Zein office was always open whenever I ran into a trouble spot or had a question about my research or writing. He consistently allowed this research to be my own work, but steered me in the right the direction whenever he thought I needed it.

I would also like to thank the experts and companies who were involved in the experiment and survey for this research project: ASAL Tech, Jaffa.net, EXALT Tech, Harri, and Eng. Hadeel Hussein. Without their passionate participation and input, the experiment and survey could not have been successfully conducted.

I would also like to acknowledge Prof. Adel Taweel and Dr. Ahmed Tamrawi of the Engineering and Technology Faculty at Birzeit University as the second readers and committee members of this thesis, and I am gratefully indebted to them for their very valuable comments on this thesis.

Finally, I must express my very profound gratitude to my parents and to my partner and children for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

شكر و عرفان

الحمد و الشكر لله عز وجل الذي أنعم عليّ بنعمة العلم و وفقني لإنجاز هذا البحث ويسر السبيل إليه. اتوجه بخالص الشكر وعميق التقدير والامتنان إلى الدكتور سامر الزين الذي أشرف على هذا العمل و وقف معي في جميع مراحل إنجازه وزودني بالنصائح والإرشادات.

كما اتوجه بالشكر و التقدير الى لجنة النقاش على ما قدموه من ملاحظات قيمة، و اشكر كل من شارك في تنفيذ التجربة و الاستبيان، و كل من وشجعني على إتمام هذا العمل، و خصوصاً شركة عسل، إجزالت، جافا، هاري، و المهندسة هديل حسين.

إهداء

إلى أعز الناس و اقربهم الى قلبي، الى والدتي العزيزة و والدي العزيز، اللذان كانا دوما عونا و سندا لي، و كان دعاؤهم مددا لي.

الى من ساندتني و خطت معي خطواتي، و يسرت لي الصعاب، الى زوجتي العزيزة التي تحملت الكثير.

الى فلذات كبدي، اولادي اللذان تحملوا طيلة الفترة التى قضيتها في اعداد هذا البحث.

الى أساتذتي و اهل الفضل علي، الذين غمروني بالحب و التقدير و النصح و التوجيه و الارشاد.

إلى كل هؤلاء، أهديهم هذا العمل المتواضع، سائلاً الله العلي القدير أن ينفعنا به و يمدنا بتوفيقه.

Contents

Declaration i

Abstract ii

Acknowledgment iv

Contents vi

List of Figures x

List of Tables xiii

Acronyms xiv

1 Introduction 1
- 1.1 Introduction and motivation . . . 1
- 1.2 Problem statement and main research projects . . . 3
- 1.3 Research Gap and study objectives . . . 3
- 1.4 Findings and contribution . . . 4
- 1.5 Main research phases . . . 5
- 1.6 Overview of this report . . . 5

2 Background 7
- 2.1 REST – Representational State Transfer . . . 7
  - 2.1.1 Principles . . . 7
  - 2.1.2 Operations . . . 9
- 2.2 ORM – Object Relational Mapping . . . 10
  - 2.2.1 Entity model . . . 10

2.3 MDE – Model Driven Engineering 10
2.3.1 Metamodel 11
2.3.2 Transformations 11
2.3.3 Advantages of MDE 12
2.3.4 Disadvantages of code generation in MDE 13
2.3.5 Resource metamodel 14
2.3.6 Deployment metamodel 14
2.3.7 REST APIs modeling language (RAML) 15
2.3.8 Domain Specific Languages (DSL) 15
2.3.9 EMF – Eclipse modeling framework 15
2.4 RDF ontology model based 16
2.4.1 Data Model 16
2.5 OData – Open Data Protocol 16
2.6 OpenAPI specification 17

3 Literature Review 18
3.1 Introduction 18
3.2 Literature search methodology 19
3.2.1 Method 19
3.2.2 Source database 20
3.2.3 Search strings 20
3.2.4 Study selection criteria 20
3.2.5 Critical literature review 24
3.3 Guidance approaches 25
3.4 EMF based approaches 28
3.5 Model-driven code generation 29
3.6 RDF ontology model based approaches 32
3.7 OpenAPI model based approaches 34
3.8 Highlight the gap of knowledge 36
3.9 Summary 39

4 Research Methodology 40
4.1 Introduction 40
4.2 Experimental research 41
4.3 Experiment design 42
4.3.1 Hypotheses 42
4.3.2 Procedure 44
4.4 Data Collection 45
4.5 Data Analysis 45

5 Implementation 46

5.1 Developed solution - RAAG Framework 46
5.1.1 Value added features 46
5.1.2 Technologies and platforms 47
5.1.3 Integration 49
5.1.4 Architecture and design 49
5.2 Proposed approach 52
5.3 Objectives realization 52
5.4 Reference application 53

6 Evaluation 56
6.1 Model example as learning material 56
6.2 Model exercise for experiment 58
6.3 Survey 59
6.4 Experiment 60
6.4.1 Participants 61
6.4.2 First group 62
6.4.3 Second group 62

7 Results and discussion 63
7.1 Survey Results 63
7.1.1 Sufficiency of learning material 63
7.1.2 Reliability of experiment 64
7.1.3 Framework quality 65
7.2 Experiment Results 66
7.2.1 Collected data 66
7.2.2 Analysis 66
7.3 Threats to Validity 70

8 Conclusions 73
8.1 Summary 73
8.2 Goals achieved 75
8.3 Recommendations 76

A Tabular Information 77
A.1 Related work 77
A.2 Implemented survey 79
A.3 Results of 1st experiment 79
A.4 Results of 2nd experiment 79

B Code Snippets 82
B.1 Model driven software development 82

B.2 Resource description framework 83

C Source Code **85**

C.1 Training example 85

C.1.1 Model 85

C.1.2 Services 85

C.2 Experiment assignment 85

C.2.1 Model 86

C.2.2 Services 87

Bibliography **98**

List of Figures

1.1	Main phases of research methodology.	5
2.1	MDA Abstraction Levels [55].	11
2.2	EMF URI string structure.	15
2.3	RDF Description [46].	16
3.1	Web APIs growth [54].	18
3.2	Applied research process.	19
3.3	Selection process.	22
3.4	Studies per publication year.	24
3.5	DaaS [61].	26
3.6	Prototype architecture with RESTFul web services [36].	27
3.7	API Composer Overview [21].	27
3.8	RDB to OData services Approach [23].	29
3.9	From Requirements to Source Code [71].	31
3.10	Rapid Realization of Executable Domain Models [69].	32
3.11	MicroBuilder Architecture [68].	32
3.12	data flow to generate APIs [25].	33
3.13	Process of the OpenAPI [63].	35
3.14	Example-Driven web API Specification Discovery [22].	35
4.1	Main phases of research methodology.	41
5.1	Framework Architecture (RAAG).	50
5.2	Class diagram for reference application.	54
5.3	Screenshots for reference application.	55
5.4	Code metrics for reference application.	55
6.1	ER diagram for learning purpose.	57

6.2 Class diagram for learning purpose. 57
6.3 ER diagram for experiment exercise. 58
6.4 Class diagram for experiment exercise. 59
6.5 Likert scale options. 60

7.1 Sufficiency of learning material. 64
7.2 Reliability of experiment. 64
7.3 Framework quality. 65
7.4 Framework quality. 65
7.5 (A) Experiment results for G1 and (B) Experiment results for G2. 66
7.6 (A) Spent time in G1 vs. G2 and (B) Time difference calculations. 68
7.7 (A) Spent time in G1.1 vs. G2.1 and (B) Time difference calculations. 68
7.8 (A) Raised model bugs in G1.1 vs. G2.1 and (B) Bugs difference calculations. 70

8.1 Contribution facets. 74
8.2 Approach categories among contribution facets. 75

A.1 Implemented survey. 80
A.2 Results of 1st experiment. 80
A.3 Results of 2nd experiment. 81

B.1 RAML API specification file Example. 82
B.2 YAML standard example – 1. 83
B.3 YAML standard example – 2. 83
B.4 RDF description syntax [46]. 84
B.5 RDF schema syntax [46]. 84

C.1 City entity model. 86
C.2 Country entity model. 87
C.3 Factory entity model. 88
C.4 Car entity model. 89
C.5 Part entity model. 90
C.6 Vehicle entity model. 91
C.7 Car web APIs. 92
C.8 Part web APIs. 92
C.9 Address entity model. 93
C.10 Student entity model. 94
C.11 Parent entity model. 95

C.12 Person entity model. 96
C.13 Student web APIs. 97
C.14 Parent web APIs. 97

List of Tables

3.1 Search strings that tried on library databases. 21
3.2 Study Include/Exclude Criteria. 22
3.3 Included articles of related work. 23
3.4 Approaches Pattern. 38

4.1 Cross-Group Design. 44

6.1 Survey Statistics. 61

7.1 Experiment descriptive statistics. 69

8.1 Research approach facets. 74

A.1 Related articles before filtration. 77

Acronyms

AOP Aspect oriented programming.. 39, 74

BNF Backus normal form. 34

BPEL Business process execution language. 34

CDI Contexts and dependency injection. 29

CIM Computationally independent model. 30

CRUD Create, read, update, and delete operation with database. 30

DaaS Data as a service. x, 26

DSL domain specific language. 29

EDM Entity data model. 29

EGL Epsilon generation language to generate code. 28, 29

EMF Eclipse model framework. 10

ETL Epsilon transformation language. 30

IoT Internet of things. 2

JAX-RS Java architecture for XML to build RESTful Web Services. 31

JAXB Java architecture for XML binding. 31

JET Java emitter templates for code generation. 28

JSON JavaScript object notation. 17

LDR Language data resource. 33

LRA Linked REST APIs. 33

MDE Model driven engineering. 10

OCL Object constraint language. 12

OData Open data protocol. 17

OpenAPI Format for REST APIs that used to describe your entire, it is based on formerly Swagger specification. 17

ORM Object relational mapping. 10

PA Parallel Agile. 76

PIM Platform independent model. 30

PSM Platform specific model. 30, 31

RAAG REST APIs auto generation. 41

RDF The resource description framework. 16

REST Representational state transfer. 1

SOA Service oriented architecture.. 2

SOLID Single responsibility, Open closed, Liskov substitution, Interface segregation, and Dependency inversion principles. 51

SWSG Safe web services generator. 34

WSDL Web services description language. 34

Chapter 1

Introduction

In this chapter, general overview will be presented about rapid REST API automatic generation *RAAG*, which introduced as master thesis in the program of Software Engineering. Firstly, a brief background with current status of REST web services and rapid development will be discussed to describe its importance for new technologies, then will summarize problem statement, and overview current research focus with a brief description about main research projects for this problem area, hence research gap can be identified. Finally, study goals with specific objectives will be addressed precisely, and then finish with findings, contribution, progress, and document outlines.

1.1 Introduction and motivation

Recently, significant attention and efforts paid off on software development techniques, to adapt and handle the huge demand on application development, its requirements, and to deliver feasible business applications and solutions. REST based is the most common style used for applications architecture. As opposed to these offers, challenges of development and maintenance. Required efforts to implement and manage software products leads to big cost and huge resources. For example, thousands of developed mobile apps online stores, require huge resources to support, where their operating systems come from a vast diversity and each one requires higher frequent upgrades [42]. Moreover, rapid development environment, short-time-to-market, and tough competition impose to find more productive and flexible techniques.

Software development requires remarkable potential and huge time, where

the most risky aspect is the maintenance fulfillment. In usual, it is difficult to implement software or to understand an implemented one for maintenance purposes. Software backbone development is a challenging area, where it needs significant efforts to properly handle thousands of simultaneous actions. For instance, data availability, integrity, confidentiality, and privacy must be handled precisely on time. Therefore, wide knowledge with strong competences is an essential qualifications to proceed in this context of development. Furthermore, high percent of developers are novice, and many of them are coming from non-computing background. In particular, big sector of Android developers are novice [32], so they tends to use auxiliary tools and frameworks.

In recent years, cloud architecture has seen rapid adoption in most enterprises, as systems infrastructure, which was a leading cause to reallocate backbone components and services for thousands of systems and applications. Backbone services of these systems have been migrated out of on-premise to remote hosts facility, such as cloud services, where cloud architecture relies on SOA web architectural style, and it interfaces world wide through web services APIs.

Consequently, using web services figures exponential increment [54], this increment refers to the huge adoption for the concept of Internet of Things (IoT), which increased number of connections, where various aspects of life and several kinds of things have been engaged with internet. Therefore, huge admittance for IoT, clouds backbones, and micro-services, all these increased usage of web services. REST style is the new common architectural style for web services, which made a big change in the world of web APIs, where most of time critical and safety domains in the world have been redeveloped again according to the REST architecture, such as government, finance, payment systems, health, and military domains.

Design principles of Representational State Transfer (REST) was identified by Roy Fielding [27], In 2000, REST architecture follows the most significant architecture, the World Wide Web architecture that based on HTTP [28]. REST architecture realizes HTTP constrains [26], such as address reliability, scalability, separation of concerns, and simplicity. Therefore, architects and developers prefer REST style more and more [34].

According to prior work, several studies focused on rapid REST APIs development, and introduced their approaches, techniques, tools, and frameworks. For instance, code generation was the most significant approach in this field of research [20, 29, 68, 69, 71], which will be discussed

in depth to avoid limitations and address research gaps. In this research, proposed approach can rapid REST APIs developments without violating rules of flexibility and maintainability.

1.2 Problem statement and main research projects

It is a very difficult task to build backbone services over data model and also to confirm consistent manipulations, since it requires deep knowledge, skills, and experience in different technologies and platforms, such as SQL Database engines, tiers architecture, application servers, and REST APIs. Such these development skills can be intimidating even for seasoned develops. On the other hand, frequent changes, rapid release cycles in terms of market stress, and limited resources, all these factors make a gap between design and implementation phases.

REST APIs development requires much time, and it is not-easy-to-use and understand. Development and maintenance for REST architectural software is a crucial, tedious, and error-prone work, so it requires huge efforts, where implementing APIs via CRUD transactions while de/serializing operations means a lot of repetitive code and complicated tasks. Moreover, realizing non-functional requirements and constrains based on SOA is a non-trivial task, where many constrains have to be considered, such as loosely coupled, interoperable, scale-able, and efficient, .. etc.

The main research projects that implemented in this problem area were; Segura, et al. [61], Ed-Douibi, et al. [23], and Ed-Douibi, et al. [22] studies that supported by Spanish government, Terzić, et al. [68] was supported Serbian government, and Zolotas, et al. [71] that funded by the European Commission.

Serrano, et al. [62] involved IBM Center in Canada Lab, Haupt, et al. [33] was funded by BMWi, and Sferruzza, et al. [63], [64] were supported by Startup Palace company. All these projects have been discussed deeply in the chapter of literature review.

1.3 Research Gap and study objectives

In general, the adopted approach in prior work was the auto code generation, but it has a major downsides on software development process,

which will be discussed in background chapter. While in parallel, most studies still needs more empirical studies to verify effectiveness.

The main goal of this study is the improvement on productivity, and easy to build REST services. Efforts focused to develop a useful framework that can create REST APIs rapidly, this approach should not affect software flexibility or maintainability. Further, senior developers can customize their own techniques over this approach without any restrict. Research dedicated for Backbone of applications to build backend REST APIs, where the detailed objectives as shown below:

- Reduce development time for backbone services, without impacting architecture, flexibility, or maintainability.
- Reduce time to learn and to understand, which simplify the required human qualifications and experience to build backbone services.
- Enable senior developers to customize their own code and implementation without restrictions.
- Avoid limitations of auto code generators, such as extra efforts, code rigidity, code overriding, technical complexity, and limited margins for senior developers.
- Enable loosely coupled manor with different frameworks, and keep upper layers intact all the time.

1.4 Findings and contribution

Introduced approach consists of an umbrella framework to abstract layers of model, data access, business logic, and web APIs, methodology has to use general and aspect oriented programming, polymorphism and AOP, which generalizes REST APIs over derived models at runtime, according to the particular designed patterns. And so, traditional automatic code generation will not be used absolutely, this framework will support most popular databases, and it can be extended for others easily and without restrictions or updates on methodology. Finally, study approved the concept through a referenced application, and empirically verified the effectiveness of methodology by an experiment with a survey that measure satisfaction and expectations.

1.5 Main research phases

- Used search engines to find related work from the most common database resources, while using a clear criteria to include actual related work precisely.
- Implemented a critical literature review for the documented prior work over a classified groups.
- Developed a framework to facilitate and speed REST APIs development.
- Designed and implemented a controlled experiment to evaluate the developed framework, as illustrated in Figure 1.1.

Figure 1.1: Main phases of research methodology.

1.6 Overview of this report

Chapter 2 explains all the related concepts and technologies to provide an adequate background for readers.

Chapter 3 introduces the relevant and documented prior work as set of groups, it discusses related resources to address knowledge gap and propose the new approach.

Chapter 4 discusses the methodology of study in order to gather and analyze the data. It provides a general idea about the research experiments and their design methods and components to conduct the research.

Chapter 5 discusses the empirical implementations through introducing the framework specification, and overview the reference application.

Chapter 6 describes the introduced training material for participants, and also discuses experiment's assignment to explain how it is comprehensive and fit to research objectives.

Chapter 7 discusses and analyzes results of survey and experiment, while explaining figured trends, tested hypotheses, and mitigated threats to confirm the contribution.

Chapter 8 summarizes results, highlights achieved objectives, and recommends for future work.

Chapter 2

Background

In this chapter, a general overview with adequate information will be introduced about REST principles, object relational mapping, model driven software development, RDF ontology model based, open data protocol OData, and OpenAPI specification.

The follow sections will introduce all relevant concepts to simplify understanding of discussions in literature review, where prior work that documented about "REST APIs rapid generation" is discussed deeply to propose a new effective approach.

2.1 REST – Representational State Transfer

2.1.1 Principles

In 2000, design principles of Representational State Transfer had been identified by Roy Fielding [27], specified principles known as REST architectural style. The REST architecture is based on the most significant architecture and the best known World Wide Web architecture [28], and based on HTTP. REST architecture involves certain constraints to address reliability, scalability, separation of concerns, and performance. The following sections show the fulfillment of REST constraints to HTTP [26], while each constraint was realized in WWW architecture, but some of them should be explicitly fulfilled by developers themselves.

Layered client server

Based on WWW architecture, separation of client components out of server aspects forms layered system structure on basis of HTTP specification, which is fulfilled by default as a very common architectural style, where building any web based application is implicitly following layered system.

Cache

Placing cache components between server and client components to handle cached data, by marking response data as cache-able or not. HTTP realized this constraint as part of WWW, it validates stale resources by using defined fields in header to control caching.

Statelessness

Servers can't keep session state for clients, but it can format data to provide client with required information to manage its state. In general this constraint has to be fulfilled over hosted applications by developer who design stateless application, for example, stateless or stateful session beans in Java EE [33].

Uniform and constrained interface

Where all interactions should be based on HTTP operations (verbs) as a small set of predefined known operations, which can be called uniform interface. This constrain fulfilled HTTP specification, which defines GET, PUT, POST, and DELETE set of methods [26], these methods procedure a uniform interface over WWW, developers have to understand and correctly use methods of HTTP to fulfill this constrain.

Addressable resources

In HTTP, URIs are used to identify resources [45], where each resource must be identified by a certain Uniform Resource Identifier (URI).

Representation oriented

HTTP realizes this constrain in the WWW architecture, where each resource may have different formats to be represented such as XML, JSON,

etc. and also representation is separated from its resource, HTTP payload labels type of message body by relevant header attributes, types of MIME media, and multiple representations [30].

Self-descriptive messages

Message includes all required information to understand and represent resource. HTTP realizes this constrain using header to separate metadata out of message data in its body, such as types of MIME media that specify message format.

Hypertext as the Engine of Application State (HATEOAS)

This constraint terms that application state of client is controlled on basis of resource contents itself, hence, application interacts according to metadata as part of resource data. HTML satisfies this constraint using its UI components e.g. hyperlinks and forms. For machine to machine interaction, representations are used. Therefore, developers have to design representations in an appropriate way to address HATEOAS constraint.

When developing a service on basis of WWW platform, most of the REST constraints are already satisfied by WWW standards over HTTP with URI and MIME, but other constraints should be considered in right service design and implementation, so developers apply uniform interface by using HTTP verbs properly, and the same for HATEOAS constraint, where realizing these constraints are non-trivial task [28, 43, 58].

2.1.2 Operations

On basis of these principles, each request is handled as an independent transaction and should be relied on group of HTTP methods only. The following are HTTP operations that used in REST: GET: to retrieve resource data. It is a read-only and safe operation. PUT: to update resource data on server. POST: used data body to create resource, it is unsafe operation of HTTP. DELETE: to erase a resource from server. HEAD: same as GET, but reply with response code only with header of request. OPTIONS: to request communication options e.g. security capabilities.

2.2 ORM – Object Relational Mapping

It provides an approach and technique to enable object oriented systems to interact with relational database, while considering concurrent, transactions, and cache control. ORM bridges models and use of constraints for the applications instead of doing that complex and repetitive task in each application. Nowadays, multi-threading is essential to achieve intensive system requirements, where interaction with database have to be very carefully implemented. On 2002, Java open source project (Hibernate) was started to realize ORM [50].

2.2.1 Entity model

Its properties acts as attributes of E-R, with persistent unique identifier, such as the corresponding database primary key. Relationships 1-1, N-1, and N-N cases are not directly managed by Entity Model, but can be bridged via entity instances. Where Entity Model abstracts the concept, and then provides configurable implementations. XML or annotations used to map class inheritance hierarchy with chosen database entities. ORM handles all details of associations at runtime, delivers object graphs for multi-nested objects, and tracks CRUD operations [50].

2.3 MDE – Model Driven Engineering

Model Driven Engineering MDE approach has achieved importance during the last decade. In MDE, a model is the main artifact and transformation is the major task in development process. Software is created by transforming top level models to the next one, and so on, until reaching to source code [67].

In MDE, UI, functionalities, and behaviors can be characterized as models [55], and models represent problem phenomena, where model is a something that can be visualized, manipulated and motivated upon. MDE based approaches, such as Model Driven Architecture MDA, Microsoft Software Factories, Object Management Group OMG, and Eclipse Modeling Framework EMF. Where MDA is a development methodology introduced by the OMG as initiative towards MDSD. Principle models are Platform Independent Models (PIM) and Platform Specific Models (PSM).

PIM identifies business logic independently, such as entities, associations, and capabilities in ERD, regardless of platform implementation

details. PSM covers all necessary information to create runnable software with environment details, such as class diagram, which can be generated by model-to-model transformation, and PIM to PSM.

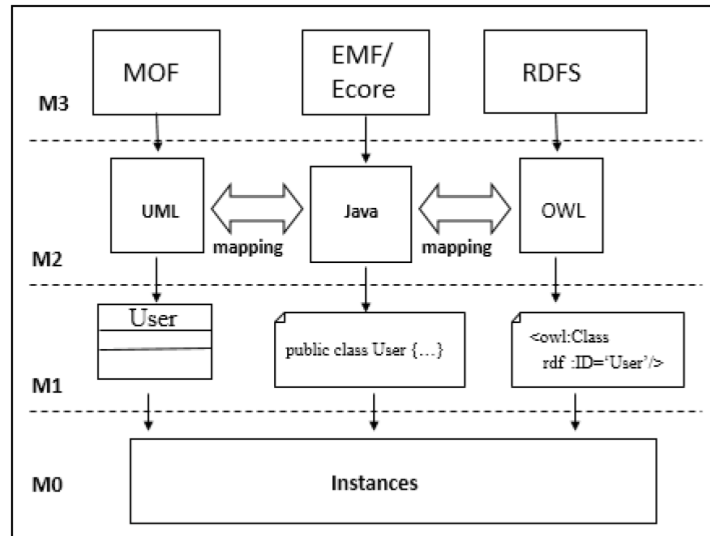


Figure 2.1: MDA Abstraction Levels [55].

The MDA based on four abstraction layers by using modeling languages, as illustrated in Figure 2.1. First layer M0 acts real world objects, including the un-modeled ones. 2nd layer M1 or model layer abstracts specific application logic, it is defined to conform to metamodel in layer M2. On third layer M3, meta-metamodels used to define metamodels of metamodels [66], where Meta Object Facility MOF is meta-metamodeling language of OMG and Ecore is of Eclipse Foundation.

2.3.1 Metamodel

It specifies structure, semantics, and definitions of models' family, such as Common Warehouse Metamodel CWM, Enterprise Distributed Object Computing EDOC, and Unified Modeling Language UML. Metamodels used for languages (e.g. C#, Java, WSDL, etc) and architectures (e.g. Web services, distributed, client-server, etc).

2.3.2 Transformations

It translates source metamodels to target once, model Transformation can be used to model system based on concepts as resources instead of

specific code syntax. Then technical complexity is hidden into model to code transformation. Hence developers face complexity once during transformation extensions only.

Transformations are based on rules of a language, rules identify mapping between elements in source and target metamodels, as semantic similarity, to be executed by an engine, such as EMF platform. One of these languages is Atlas Transformation Language ATL that defined by the Atlas Group [67], Query-View and Transformation Language QVT by OMG, BOTL, and Yet Another Transformation Language YATL. Engines interpret transformation rules. The ATL is a toolkit that can be used with Eclipse, integrated over its IDE [56], it provides Object Constraint Language OCL based language to facilitate rules definitions.

2.3.3 Advantages of MDE

Following are the main advantages of adopting MDE architectural style:

Separation of concerns

Separation between specification and implementation saves efforts when focusing on business rules [13], and also includes new platform, only new definition has to be extended as a separated transformation rules for it [67]. Therefore, Business concerns are defined on the PIM models. PIM can be transformed to PSM or to numerous PSMs that maintaining business logic, such as: UML to C#, Java, or WSDL PSM models. And also, it enables development of Domain Specific Language DSL to simplify domain classes' development, just as a new template of rules for new domain.

Uniform architecture

As per transformation definition, all components of the models are homogeneous and constructed in the same way, which is more easy to understand.

Rapid development

This approach rapids development, model to code transformation should be adapted one time, and then all relevant classes can be generated. But transformations as generators should be adapted, extended, and models have be developed and maintained. Moreover, when a system has been

developed, it should be maintained with large number of components, so speed becomes decried.

Reusability and interoperability

Same modeling languages and also generators can be reused for other systems. PSM can be reused to generate code for a different platform with same language, and also be interoperable for another platforms, but metamodel should be extended.

Software quality

Software can be easily tested over a uniform systems, usually experts develop transformation rules, and considering separation of concerns, all of these can improve quality.

2.3.4 Disadvantages of code generation in MDE

Unfortunately there are also downsides to model driven software development, especially in using code generators, as illustrated below:

High effort in initial stage

In the beginning of any project, environment have to be prepared as well as to setup required platforms, auxiliary tool, and enough for team understanding. In addition to architectural decisions of system, code generation and model driven software development require another architectural decisions to address concerns of MDSD. Therefore, modeling and auto code generation approaches have to define and develop modeling languages and transformations in the initial stage of each project.

Possibility of losing code

The generator is overwrites the already created classes with new generated versions. Therefore, code generators have to be adjusted to distinguish manually customized code, which means extra efforts and complexity. Otherwise, in advanced stages of development, the manual customized code might be overwritten after new code generation.

Code rigidity

Using code generators earnings many choices for developers, but it introduces rigidity. MDSD developers have a restricted style for imple-

mentation. So, development bugs can be only solved, if they considered it in the transformation or generation rules, otherwise, rules have to be violated to avoid adapting generated files after generations.

Technical complexity

Adopting auto code generators, such as MDSD, often introduces new developments MDSD tools in addition to systems development. Moreover, number of technologies have to be multiplied to define, process, and transform models or auto generators, which requires huge efforts to understand, maintain, and then it introduces further errors and bugs sources.

Impacting engineering team and managerial levels

Senior software engineers dislike environments of code generation, since it limits their experience, and makes them similar with junior, so they leave teams, which impacts business of software developments.

2.3.5 Resource metamodel

The Resource Model only identifies links and relationships among resources, which navigates API and then fulfill HATEOAS constraint. HATEOAS feature is very important to satisfy loose coupling concern, and also supports documentation process. Resource, Method, and Link are the basic elements of resource modeling. Where resource element represents REST resource that holds attributes of entity and model. Method element represents HTTP operations, and link element defines the relationship between two resources.

2.3.6 Deployment metamodel

The deployment model supports the relationship of resources with URIs, it defines URI of a resource to be relative to other resources not only to base URI, it includes mapping list of source and target elements. URL fragment realizes deployment model by referring to static URL fragment as string indicating a static path element, while dynamic URL points to entity attribute of a resource model. Deployment model forms a tree, which avoids cycles in the model and keeps only one path for resource.

2.3.7 REST APIs modeling language (RAML)

Used to explain parameters and endpoints of REST API and based on YAML-based language, which facilitates operations' representation in a hierarchy mode to support Reusability. It forms a platform for some tools, such as: online editor, automatic documentation generator, and code generator. An example of RAML in Figure B.1.

2.3.8 Domain Specific Languages (DSL)

DSL occupies generality for expressiveness in a restricted domain, its notations and constructions are provided to be relative to specific application domain, which offers extensive improvements of expressiveness to ease of use compared with General Purpose Languages GPLs, which increases productivity and reduces maintenance.

2.3.9 EMF – Eclipse modeling framework

It is a modeling framework used as domain-specific languages, EMF visualize metamodels as class diagrams with creation, opening, changing, and storing capabilities. Instances are saved as XML. The EMF unifies XML, UML, and Java in its Eclipse IDE [66]. Ecore used to highly abstract definitions of model as metamodel language over MOF implementation of OMG [24], and used to represent EMF models [17]. EMF is considered as a key reference of modeling in Eclipse [24], it also builds Java APIs to simplify their management.

EMF based on URI as structured string that consist of three parts; scheme, scheme-specific part, and optional fragment, as shown in Figure 2.2. The scheme is separated with colon ":" to identify used protocol. As shown in the below example: "https:// [applicationLink]/rest" is the URL of the Web application, model Id identifies model and Model Instance Id identifies model instance that being accessed. URL doings as an entry point for a specific model instance and, which points to the root element.

[scheme:][scheme-specific-part][#fragment] (1) https://[applicationLink]/rest/[ModelId]/[ModelInstanceId] (3)
platform:/resource/project/AddMovie.xmi (2) https://example.com/rest/IFMLModel/AddMovie (4)

Figure 2.2: EMF URI string structure.

2.4 RDF ontology model based

It is a metadata framework that consists of structured XML syntax, it supports data encoding, interchange, and reuse of data, and it provides unambiguous operations to precise semantics. RDF means and vocabularies (set of properties) have been designed to interface with both human and machine, and its metadata semantics are reusable and extendable. RDF transforms Web vast information from unstructured mass to be manageable and more useful. W3C supported RDF development in 1995 to describe contents of web info, which can discriminate confidential, nudity, or violence contents [46].

2.4.1 Data Model

Model describes resources, each resource has properties or attributes as any object, but it should be identified by specific URI. Description includes a resource linked with its properties that consists from pairs of types and values, where values might be atomic or another resource [46]. Figure 2.3 illustrates description by example as conceptual, syntax in Figure B.4 and Figure B.5.

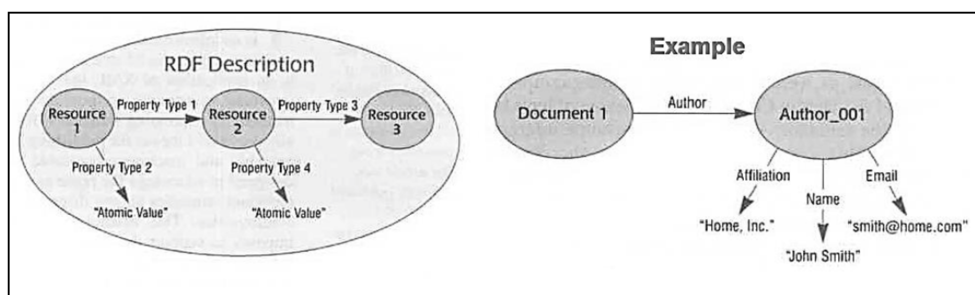


Figure 2.3: RDF Description [46].

2.5 OData – Open Data Protocol

It creates Web services with comprehensive APIs to query and update simply, it is according to a standard way as data access protocol. SQL over URLs is similar to standard to facilitate its usage [53], it can expose services out of different data sources, such as: file systems, relational databases, and content management systems, it simplifies data access

via services and RDF, it already used by SAP, IBM WebSphere, or JBoss. It is approved as an OASIS standard [52].

Creating OData services is boring and consuming time, particularly for relational databases. Where (i) data models have to be represented based on EDM format, and then (ii) implement query and update for business logic over URLs by query of OData, and (iii) transforming that queries into SQL. Finally, (iv) de/serialization to exchange messages over OData protocol as JSON or Atom format. OData SDKs are Apache Olingo3, RESTier2, SDL OData Frameworks4, OData server6, Cloud Drivers5, and Skyvia Connect7.

2.6 OpenAPI specification

It is known by its original name, where it was Swagger Specification, it was developed and maintained by SmartBear Company, it called Open API Initiative under Linux Foundation sponsorship. IBM, Google, and Microsoft are founding members. API documentation specify how each function or endpoint can be used with constrains [20].

REST APIs is being documented in different ways heterogeneously, based on vendor approach, vast variety of descriptions difficult their understanding to implementers and other stakeholders. And to provide a solution and standardize REST APIs Specification, on 1 Jan 2016, the Swagger specification was renamed the OpenAPI Specification, and RAML and API Blueprint were considered by the group. Where file specifies API by: general information, available paths; operations of each path (get/resources), and Input/output for each operation. OpenAPI assumed as formal specification for REST APIs to describe them by JSON or YAML format. As illustrated in Figure B.2 and Figure B.3.

Chapter 3

Literature Review

3.1 Introduction

Web services and Microservices are mostly based on REST APIs, REST have made a revolution in the world of web APIs, where most important systems over the world have been reengineered on basis of REST architecture, such as financial, governance, and control systems. As per the last study of ProgrammableWeb directory [54], study shows high interest in providing APIs, which exceeds the 19000 API mark in Jan, 2018, as shown in Figure 3.1.

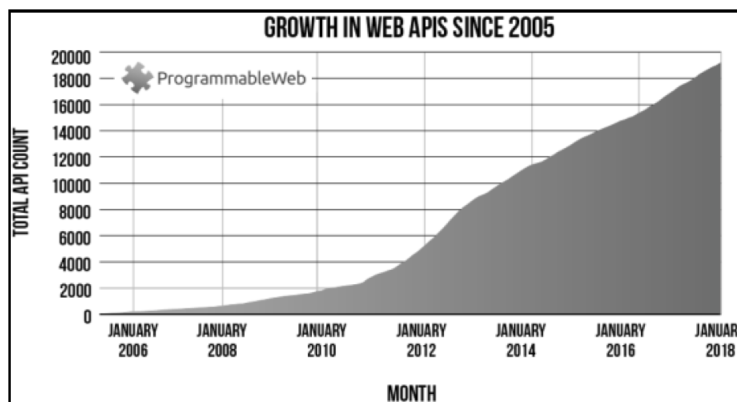


Figure 3.1: Web APIs growth [54].

In usual, SOA architecture considers different types of software clients that may handle backend services, these services should be enough robust to handle thousands of request simultaneously, where data avail-

ability, integrity, confidentiality, and client privacy should be processed precisely with well performance. Therefore, backend development is a very critical job, and it needs a several skills and vast knowledge.

In this chapter, the documented prior work around "REST APIs rapid generation" will be discussed in depth to propose a new effective approach. Several researches were supported by major institutions such as IBM, BMW, and some of them were supported by European governesses. In the beginning, first sections discussed tracked stages in this research, where these stages demonstrate how to apply critical review precisely, and also show how to eliminate bias.

3.2 Literature search methodology

In this section, critical review is considered to review literature [31, 37], discussions focus on the followed steps to collect and prepare material of strongly related work, moreover to explore the used search strings and the adopted criteria that include or exclude results, and hence categorize and present the accepted results in a comprehensive demonstration, which confirms completeness of prior work inclusion.

3.2.1 Method

Peterson and Kitchenham studies [47, 39] inspired the methodology for this review. Furthermore, data analysis and assembly based on the recommendation of other studies [15, 48], which enables this study to introduce a baseline in this research field.

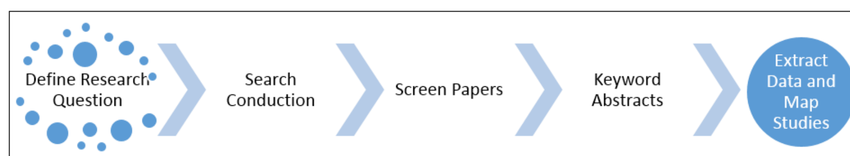


Figure 3.2: Applied research process.

According to Petersen [47]; define research questions or objectives, search conducts, screen papers, keyword abstracts, and data extraction are the main stages for mapping process and critical review, as shown in figure 3.2. Where search conducts select primary studies, screen articles classify scheme, some stages can be done in iterative manor.

3.2.2 Source database

The most used engines for academic databases search were:

- Google Scholar.
- IEEE Xplore.
- Springer Link.
- ACM Digital Library.

3.2.3 Search strings

Selection of keywords and terms enables to build an appropriate strings to run specific search, which have to be reliable and hence address whole related work. However, strings were well recognized over the related terminology of literature.

Used strings were edited very carefully during search operation, strings have been listed in a table and built accumulatively to be comprehensive string, search string was being typed and updated based on search results and previous used strings, and also strings were documented before using them in search to ensure the validity of context, as shown in Table 3.1.

3.2.4 Study selection criteria

Search execution was done over all the online databases, search string was built carefully according to results and used keywords or terms. Old studies were used as snow ball papers that point to prior work by references and followed researches from citations, and so on run through a chain until discovering all related work.

Papers filtration go through three main iterative phases, as shown in figure 3.3. Where preliminary phase to run search string online engines, first phase to filter studies on basis of title and the abstract, second phase to read more sections, such as introductions, approaches, and conclusions, and then filter gain and re-select. Finally, apply inclusion or exclusion criteria after completing the reading of remaining papers, as shown in Table 3.2.

After applying the above include – exclude criteria, 19 articles have been selected and categorized from the entire list of papers, as shown in Table 3.3.

Table 3.1: Search strings that tried on library databases.

Try	String
29.09 23:09	framework to minimize development lifecycle
29.09 23:14	speed developing rest api
29.09 23:15	speed developing REST APIs
29.09 23:17	REST APIs development life cycle
29.09 23:21	REST APIs software development life cycle
07.10 07:02	increase software productivity
12.10 07:13	framework to speed software development life cycle
12.10 07:16	REST development framework
12.10 15:38	Rapid REST API development
12.10 15:43	Rapid REST development
12.10 16:02	”Rapid RESTful development”
12.10 16:02	Rapid RESTful development
12.10 16:03	”Rapid development”
12.10 16:03	”rest Rapid development”
16.10 08:44	software development framework
18.10 16:45	Rest Rapid development
20.10 08:17	Rest Rapid software development
20.10 10:01	Wang: Rapid realization of executable domain models...
17.11 07:03	Musleha: Automatic Generation of Android SQLite Database...
17.11 07:13	Automatic Generation of Android SQLite Database Components
27.11 10:49	The wte+ framework: Automated construction and runtime...
27.11 17:32	Supporting model-driven development using a process-cente...
28.11 07:37	An approach to code generation from uml diagrams
28.11 08:21	Automatic code generation using uml to xml schema...
28.11 08:22	”Automatic code generation using uml to xml schema...
28.11 08:23	Model checking and code generation for uml diagrams...
05.03 21:58	build ”back end” services rapidly
05.03 23:25	Model as a service
08.03 11:09	Model as a rest service
08.03 21:25	entity as a rest service
08.03 22:30	entity to rest service
08.03 22:40	database table to rest service
08.03 22:50	oracle apex
09.03 09:00	apex rest
09.03 09:25	”Application Express”
09.03 09:35	Query as a service
09.03 09:50	NoSQL database collections
09.03 00:05	ERD to rest services
15.03 00:28	Database driven Rest web service
16.03 03:45	Model-Driven Engineering approach for RESTful Entity-based REST services generation

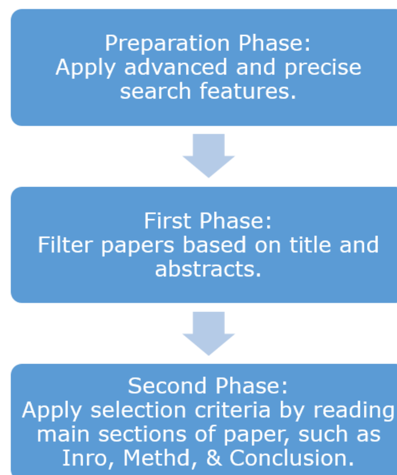


Figure 3.3: Selection process.

Table 3.2: Study Include/Exclude Criteria.

Criteria	Description	Comments
Empirical Article	To include empirical article and exclude conceptual ones, those have not any data for experiment, case study, or survey.	Very high ranked may be considered.
Publish Date	Consider new article and exclude old one, any paper was published before 2014, assumed as old article.	Closed to 2014 may be considered, based on its relevant rank.
Size	Very short article with less than 6 pages, like conceptual ones, will be excluded.	
Rank of Relevant	Articles that evaluated as more than 5 of 10 rank can be included in relevant work.	Evaluation based on contents judgment.

According to the applied filtration criteria, 19 studies selected as related work, most of these were published at 2017, 2018, 2014, and 2013, as shown in figure 3.4. Where old studies have been considered in case of strong relevant.

Table 3.3: Included articles of related work.

Approach	Article
Guidance	Developing a Prototype of Rest-Based Database Application for Shipbuilding Industry: A Case Study.
	A model-driven approach for REST compliant services.
	ODaaS: Towards the Model-Driven Engineering of Open Data Applications as Data Services
	Semi-Automatic Generation of Data-Intensive APIs
	APIComposer: Data-Driven Composition of REST APIs.
	A Model Driven Approach for the Development of Semantic Restful Web Services.
	Structural and Behavioral Modeling of Restful Web Service Interface Using UML.
EMF Based	EMF-REST: Generation of RESTFUL APIs from Models.
	Model-Driven Development of OData Services: An Application to Relational Databases.
Model driven with Code Generation	Rapid Realization of Executable Domain Models via Automatic Code Generation.
	From Requirements to Source Code: A Model-Driven Engineering Approach for Restful Web Services.
	Microbuilder: A Model-Driven Tool for the Specification of Rest Microservice Architectures.
	Model-driven Code Generation for REST APIs.
	Restful Web Services Development with a Model-Driven Engineering Approach.
RDF ontology Model Based	Linked REST APIs: A Middleware for Semantic Rest API Integration.
	(Semi) Automatic Construction of Access-Controlled web Data Services.
OpenAPI Model Based	Example-Driven web API Specification Discovery.
	Extending OpenAPI 3.0 to Build Web Services from Their Specification.
	A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models.

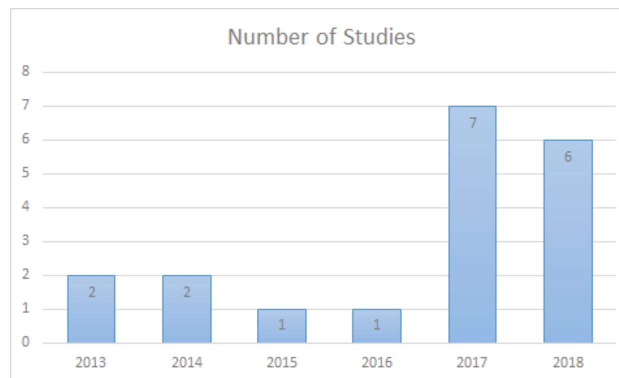


Figure 3.4: Studies per publication year.

3.2.5 Critical literature review

On basis of readings for abstract sections, the selected papers from entire results have been totaled 54 important papers for this study. After deep investigation, related studies were found in 31 articles, as shown in Table A.1, related papers have been evaluated from different perspectives, such as relevant rank and date of research, with comments. Finally, while applying include – exclude criteria, articles have been filtered to 19 exact related studies, as discussed in the next section.

By selecting the exact related work, review process can be started, while considering critical literature review [31, 37], but the big number of articles makes critical review more difficult to match and compare. Therefore, studies have been classified into five categories based on their contributions, which simplifies the process of critical review among few number of paper groups, this classification mainly considered contribution since review and analysis have to find gaps among different contributions, which should be addressed in this study.

First seven articles have proposed guides as a method to rapid productivity and simplified development process, so first category is considered as guidance approaches. The next two studies have introduced the extension of Eclipse modeling framework EMF, with some variance, as a solution for the same addressed problem in the prior group, so the second group of papers is considered as an EMF based approaches. Another next five studies have generally proposed model-driven code generator as a methodology, where other three studies have used RDF ontology model based as a method, and the last three studies have exploited OpenAPI specifications to provide solution that using OpenAPI

model based concept.

The five categories should be discussed deeply in the next sections based on their contexts, major addressed problems, contributions, assumptions, employed methodologies, and addressed limitations and gaps for future work.

3.3 Guidance approaches

In this section, seven studies were described [61, 33, 67, 19, 21, 36, 57], articles in this group proposed a guidance approach in general, except some of the introduced supplementary tools, utilities, and frameworks. Some of these researches were supported and sponsored, for instance Segura, et al. [61] was supported by Spanish ministry of economy, and Haupt, et al. [33] was partially funded by BMWi project Migrate. Half of these articles were published in IEEE.

Most papers in this group used empirical case study to proof their concept, such as Bill of Materials, hotel booking, and YSN (Yahoo News Search) for Tavares and Vale [67]. In general, they addressed the problem in a much time to explore and discover data model behind REST, where heterogeneous formats require too much particular pre-processing steps to identify and analyze that model structure. Moreover, they point to the crucial, tedious, and error-prone process to build and maintain that structure, and use various technologies halts interoperability. Furthermore, Haupt, et al. [33] diagnosed the problem as lack attention to build compliance REST that keeps loosely coupling, scalability, and efficiency.

Authors assume that web-based approach increases productivity by reducing development time and also minimizing maintenance efforts. Model driven software development (MDSD) leads for better code quality with less errors, better Reusability and maintainability over standardized code, and better portability based on separation of concerns (PIM & PSM). On the other hand, Haupt, et al. [33] assume that developing compliant REST applications is a crucial and non-trivial task, where object oriented structures and relational data structures are not mutually compatible.

Studies contribute in solving the above mentioned problems by introducing their approaches on basis of model-driven engineering. In general, proposed approach advises to use specified architecture as a set of utilities and technologies in a unified and extensible framework, which auto-

matically composes and orchestrates REST APIs while discovering data models, and it identifies matching concept and global model. Such these architectures use unified modeling language UML to design structural and behavioral REST services, and in the same time consider the separation of concerns through multi layered model design. Moreover, Tavares and Vale [67] approach raised up the level of development abstraction by providing meta-model language for resources and services, which supported interoperable development through model transformation concept. Costal, et al. [19] study goes farther while semi-automating the generation process for the data-intensive APIs.

The employed methodologies tends to guide practitioners how to use the most efficient and suitable technologies in a certain sequence and context, where the suggested technologies are already aligned with REST-based architecture, with three main layers as data mapping, business logic, and web-based APIs. In the first layer, guides focus on using common and mature platforms in frameworks, such as Grails ORM [36], data models derived by resource entities and relations, models derived by an abstract strategy, and extract source ontologies with variant format (CSV, XML, JSON, relational, and RDF) [19]. Second layer guides to employ certain language for middleware, such as Groovy on Grails framework, and use UML state machine to represent structural and behavioral aspect, it also guides to implement a model to model transformation based on Eclipse plugins for code generation (Figure 3.5), and use data injectors to engage reusable libraries, it also guides to produce a single global ontology by constructing, enriching, and refactoring it iteratively [19].

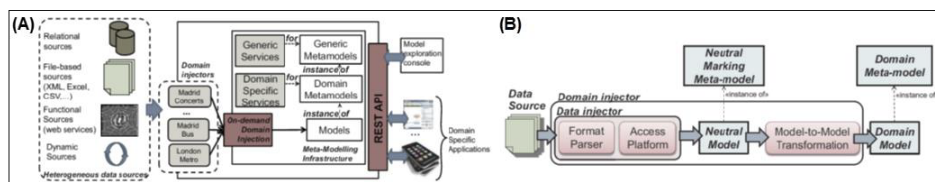


Figure 3.5: DaaS [61].

In third layer, web-based APIs rely on REST architecture and use JAX-RS Grails with Spring security Grails plugins [36], which provides transformation rules to generate data-intensive Web APIs, such as GraphQL, REST APIs [19], OpenAPI extension, and binding metamodels over EMF implementation.

Introduced maintenance and change process by Jeon and Chung [36] requires manual edit for Groovy code in the domain model class, and also needs to update corresponds properties with data entity.

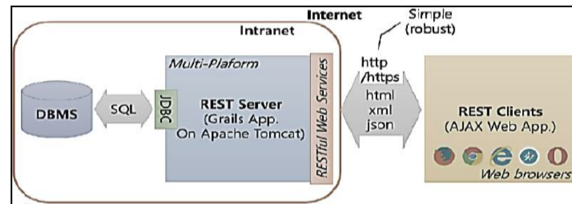


Figure 3.6: Prototype architecture with RESTFul web services [36].

And moreover, developers should manually modify code of JavaScript and Groovy server pages to maintain intended REST service, where the overall methodology illustrated in Figure 3.6.

API composer approach that used by Ed-douibi, et al. [21] takes inputs of REST APIs as OpenAPI definitions to compose them, where these definitions are supplied by API provider and generated by API discoverer. OpenAPI based over EMF, and OData relies on Apache Olingo13. Composer includes two components, an importer and resolver, where importer creates OData metadata by UML and binds model to integrate REST APIs with global, meanwhile resolver requests over OData service as data model. Unfortunately, this approach is suitable for data retrieval only, as shown in Figure 3.7.

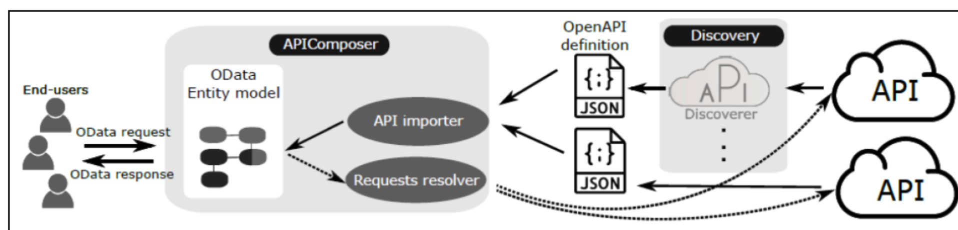


Figure 3.7: API Composer Overview [21].

Limitations for future work

Some gaps have been addressed to be covered in future works, authors themselves addressed these limitations, such as limitations in CRUD operations, transactions processing, files attachments [36], and incomplete automation. Moreover, more evaluation is needed to evaluate and proof

the proposed concepts [19].

3.4 EMF based approaches

In this section, two studies were discussed [23, 24], in general proposed approaches rely on Eclipse modeling plugins. Ed-Douibi, et al. [23] research was supported by Spanish government, and it have been published recently, 2018.

Eclipse plugin have been developed as an empirical case study to proof the concept of Ed-Douibi, et al. [23]. Articles addressed the problem in a time-consuming and tedious task to build services, where all the following tasks are not easy or trivial, such as presenting data models in service format, implementing business logic and transform service request into SQL statements, and serializing de-serializing exchanged messages. Ed-Douibi, et al. [24] in first publication, 2016, identified the problem, at that time, desktop-based scenarios used instead of web-based, which restricts framework capabilities.

Assumptions in this section consider that web-based architecture facilitates collaborative development between modelers, such as cloud models, and also model driven engineering paradigm contributes in raising the level of abstraction and then promote automation process of software development. On the other hand, SDKs and OData services still require advanced and deep knowledge to be used, such as SDL OData Frameworks4, Apache Olingo3, RESTier2, Cloud Drivers5, OData server6, and Skyvia Connect7.

Contributions in this section can be summarized in a proposed approach to generate REST-based web APIs over EMF data models, depending on common libraries and standards to facilitate maintainability and comprehension, and also to integrate model with validation and security feature and capabilities. Following to model definition, all the required artifacts have been derived from UML to semi-automate OData services out of relational database [23].

The employed methodologies in this section automate code generation for REST implementation, where EMF-REST approach [24] generates REST APIs out of Ecore models by using JET and EGL templates for model-to-text transformations, (i) start in Maven-based project, (ii) extend EMF to include JAXB and validation, (iii) use JET templates to produce corresponding code for JAXB annotation and OCL validation

methods, (vi) use EGL model-to-text transformations to generate JAX-RS, CDI and EJB code.

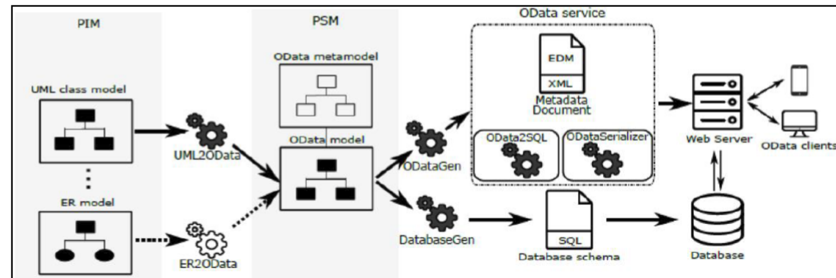


Figure 3.8: RDB to OData services Approach [23].

OData models were used by Ed-Douibi, et al. [23], where (i) OData meta-model over EMF to define Entity Data Model (EDM), SQL transformation, sterilizer, and formatter, (ii) OData services relies on Apache Olingo20 for query and serialization, JOOQ21 provides a DSL to build SQL queries, (iii) OData server is generated by simple-odata-server29 and JayDATA30 out of entity model with corresponding database, (vi) proof-of-concept by Eclipse plugin to generate meta-model, database DDL, and OData service in Maven-based project. Figure 3.8 illustrates approach of relational database to OData services.

The major addressed gaps and limitation to be covered in future works were such as adopt unsupported libraries, and enable DSL configuration, as mentioned by authors.

3.5 Model-driven code generation

In this section, five studies were discussed [20, 29, 68, 69, 71], articles proposed approaches that based on model-driven code generation. Most articles have been published in the last two years. Zolotas, et al. [71] research was supported by Collaborative Project that funded by the European Commission, and Terzić, et al. [68] research was supported by Ministry of Education in Republic of Serbia. Studies have used method of empirical case study to proof their concepts, where da Cruz Gonçalves and Azevedo [20] built their case study using application of health and wellness for Android and Web, and they compared developed solution with the experimental manual work.

Articles addressed the problem in a time-consuming and tedious task to

build services, where the uniform interface style of REST services means a lot of repetitive code, and also the similarity between written code for CRUD operations leads to tedious and repetitive tasks, and in the same time, service functionalities such security, database transactions, and model driven approach, these are crucial and require huge efforts to be developed and maintained. Wang, et al. [69] exceed these definitions to diagnose a gap between design and implementation, they believe that it is due to frequent changes and rapid release cycles in terms of limited time and resources.

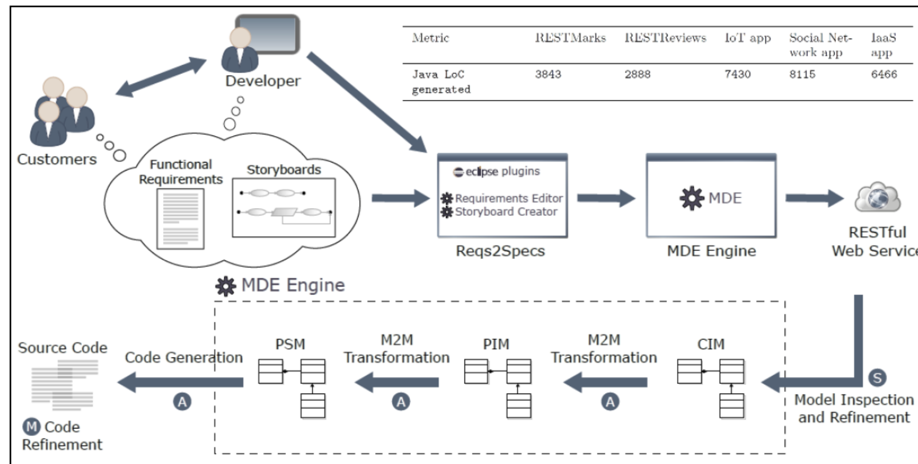
Articles assume that model driven software development approach is more difficult than traditional approach [69], which needs training and more identification since most developers don't have these skills. By refereeing to practitioners [35], Zolotas, et al. [71] assume that model driven engineering MDE facilitates code automation, reduces defects, and improves productivity, quality, understand-ability, [20] consistency, and maintainability. Other articles [20, 29, 68, 69, 71] also assume that compliance of web services with cloud and IoT makes REST very important and essential.

Major proposed contributions were addressed in using UML to model system domain, then automatically generate code of common operations for database access, and then wrap generated code within RESTFul APIs, hence measured efforts will be positively impacted and applications become more flexible and reusable. Transformation from model to model was also proposed to keep the flexibility and portability [29].

The employed approach in this section can be structured into three main stages, The development of a (i) platform specific model with CURD operations, (ii) transformation from model to model or to text, such as code syntax, and (iii) reference API. For the first stage, UML domain modeling used to clear uncertainty and to be more compliant with SOA, and based on the model, database functions can be assigned to manipulate data (CRUD: create, read, update, and delete), Hibernate and Lucene ORMs were used [71]. In general, authors ensured that design of domain model is reusable, loosely coupled, independent, and not rigid.

For the second stage, two types of transformation were used, a model to model and model to text that used to generate application code, for model to model, Fischer [29] used Epsilon Transformation Language ETL, it used to translate platform independent model PIM to platform specific model PSM, and also Zolotas, et al. [71] deigned three transformations, CIM-to-PIM, PIM-to-PSM, and then generates code by trans-

forming PSM to source code, as shown in Figure 3.9, with related results.



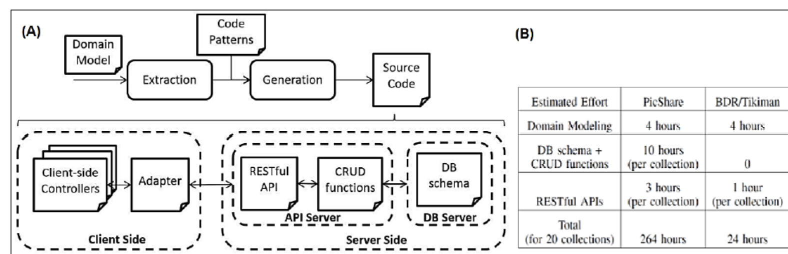
Metric	RESTMarks	RESTReviews	IoT app	Social Network app	IaaS app
Java LoC generated	3843	2888	7430	8115	6466

Figure 3.9: From Requirements to Source Code [71].

In the last stage, the approach of splitting large service into small ones was adopted, which makes micro-services more suitable for REST accessibility, and based on this methodology, code generators creates executable program. As per Zolotas, et al. [71], PSM models will compress Java, Hibernate and Lucene ORMs, JAXB for XML to Java, and JAX-RS over Jersey implementation, but unfortunately, in most cases the developer needs to add manual code. Terzić, et al. [68], da Cruz Gonçalves and Azevedo [20] used DSL to provide architecture specification for REST micro-service. DSL extends OpenAPI with its grammar that is in conjunction with base Java grammar foundation.

Wang, et al. [69] approach is compatible with NoSQL database only, backend was deployed on Amazon cloud (AWS) with MongoDB and NodeJS, according to the above described methodology, as shown in Figure 3.10, with results.

MicroBuilder tool was built over EMF, Terzić, et al. [68], they used MicroDSL over Xtext language to consist a generated code of user-defined and Netflix micro-services over Spring (Cloud) and Amazon NetflixOSS frameworks, with MongoDB database, where the generated code spread over four layers as REST APIs that acts as the controller, service as business logic, repository as data management, and class as object values for business entities. Abstraction on the level of controller is over all user micro-services of business entities to specify basic CRUD operations of



Estimated Effort	PicShare	BDR/Tikiman
Domain Modeling	4 hours	4 hours
DB schema + CRUD functions	10 hours (per collection)	0
RESTful APIs	3 hours (per collection)	1 hour (per collection)
Total (for 20 collections)	264 hours	24 hours

Figure 3.10: Rapid Realization of Executable Domain Models [69].

REST APIs, as shown in Figure 3.11, with its related results.

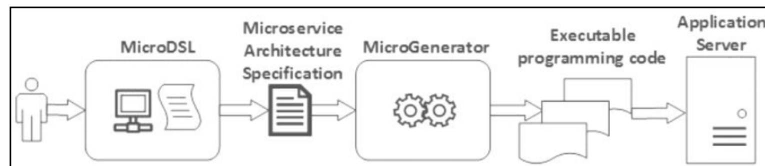


Figure 3.11: MicroBuilder Architecture [68].

Limitations for future work

Several gaps have been addressed for future works, such as verifying feasibility of proposed solution with relational and graph database [69], farther evaluation for effectiveness (product quality and time reduction) of the proposed methodology [71], and investigate detailed to analyze proposed approach using case study [68].

3.6 RDF ontology model based approaches

In this section, two studies were discussed [62, 25], authors used the concept of Linked-Data RDF ontologies as a semantic layer to generate REST based services. Center for Advanced Studies of IBM in Canada Lab was involved in Serrano, et al. [62] research.

Empirical case studies were used to proof their concepts, research-data management application used by Serrano, et al. [62] and Eng, et al. [25] used fitness application. Addressed problems in this section were as the difficulty of building SOA applications, which needs huge number of underlying services, and so, these services overlapped and become hard

to reuse and maintain its mapping [62], while in parallel, market stress increased dramatically to harness economic value [25].

Authors assumed that developing web APIs still needs a considerable efforts to be done, middleware of linked REST APIs can reduce manual work of software backend developers [62], and it is a complicated task to keep confidentiality and privacy constraints while developing REST web services [25].

Major contributions were proposed in the developed middleware to compose APIs calls that respond to data queries, and also in the developed RDF model that characterizes access control over these APIs [62]. Eng, et al. [25] developed model driven approach creates web data APIs by using automatic generation out of relational databases.

The common proposed methodology in this section, was divided into four main stages, (i) data model extraction out of database by using ORM library, (ii) manual mapping for the extracted data model with relevant ontologies, (iii) four manual steps to assign role based access control permissions via annotations, and (vi) use identified resources and template engine to generate code of REST services automatically. For the second mapping stage, Eng, et al. [25] built an interactive editor tool called Karma, which list the un-mapped tables in a visualized interface, and enable developer to manually assign or confirm the most appropriate RDF concept for each table, as shown in Figure 3.12.

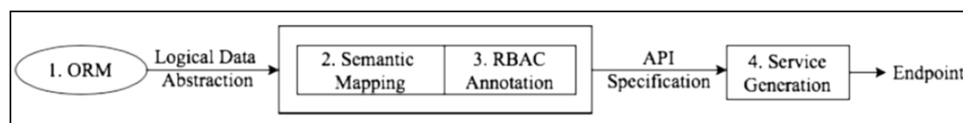


Figure 3.12: data flow to generate APIs [25].

By Serrano, et al. [62] the four stages were combined rigidly in a one middleware layer, which utilizes Linked Data RDF ontology to annotate REST APIs semantically, LDR was selected for its flexibility and to express association among service elements. LRA middleware provides APIs to handle SPARQL query that considers constraints of access control.

Limitations for future work

Addressed gaps that may be covered in future works, such as quantify effectiveness of the proposed middleware and to mature how much is it

easy to be used by developers [62]. Use automatic generation to extend usage of MDE principles to cover streaming APIs instead of REST APIs [25].

3.7 OpenAPI model based approaches

In this section, three studies will be discussed [22, 63, 64], authors built their approaches to be complaint with OpenAPI specifications. Ed-Douibi, et al. [22] work has been supported by the Spanish government, [63, 64] supported by company of Startup Palace, a web agency that evolves in this context.

Problems Addressed in this section were as the following, most REST APIs released without any specifications, which can be used automatically by machine or developers, specs facilitate understanding and use for integration purposes [22], manual documentation often causes misalignment between model and web services, and OpenAPI model requires manual update in case of change on service APIs [63, 64].

Authors assumed that web APIs are becoming as the backbone of web applications, cloud services, and mobile applications. The leading approach for web APIs is the REST architecture [22], and so abstraction is a neediness for software development process to express, independent, reuse, and change features [64].

Articles in this section proposed the following contributions, using an example to call REST web APIs to fetch and generate specifications of model based OpenAPI, which proposed as an example driven discovery approach and implemented through a tool to discover APIs [22], extending OpenAPI specification to build web services based on MDE, it uses visualization and code generation to express high-level representation of web services, it is based on SWSG tool expansion (Safe Web Services Generator). They built a tool to verify consistency of extended OpenAPI models with code generation for corresponding services [63, 64].

The common proposed methodology in this section, was as the following, (i) extends meta-model of web services and also extends the SWSG tool as per Sferruzza, et al. [63], as shown in Figure 3.13, (ii) defines a metamodel for web services using BNF grammar without relying on any standards descriptive, such as WSDL or BPEL, (iii) then introduces a compacted and readable concrete syntax for the model and services, and (vi) extends process and service parts of OpenAPI 3.0 with good

abstraction for Reusability and flexibility, but due to rigidity, it will override any customization based on specific business requirement, where the above three steps seems as re-inventing the definition of web API [64].

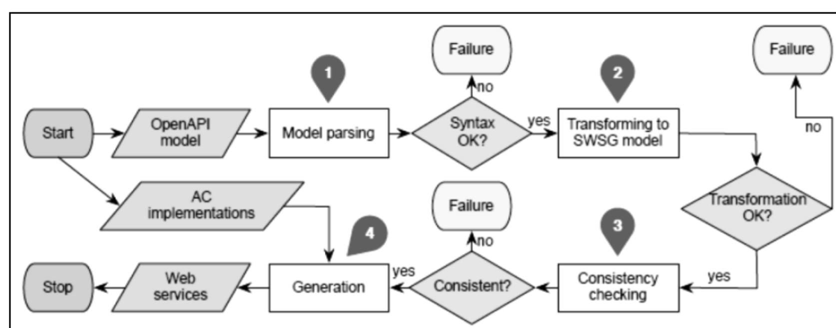


Figure 3.13: Process of the OpenAPI [63].

But Ed-Douibi, et al. [22] followed different approach over OpenAPI, by (i) creating an intermediate out of OpenAPI specifications that based on model representation, intermediate can generate, transform, analyze, and validate the discovered specifications, (ii) generating definitions of OpenAPI as JSON Schema, (iii) integrating APIs into model driven process, where discovered specifications added incrementally into OpenAPI model, as shown in Figure 3.14.

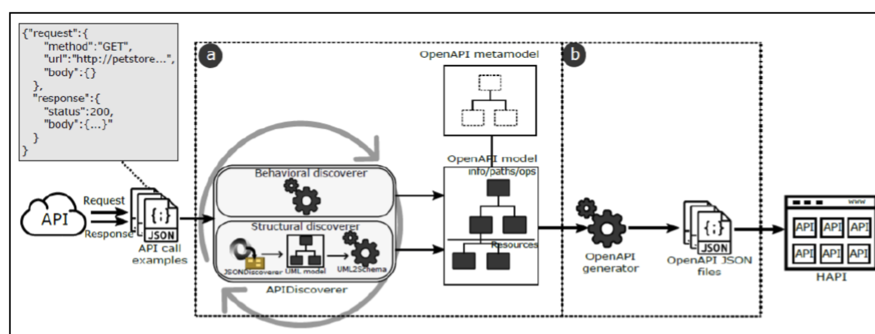


Figure 3.14: Example-Driven web API Specification Discovery [22].

Limitations for future work

Addressed gaps to be covered in future works, where the proposed approach [64] have to be evaluated in more realistic and larger case stud-

ies, these studies should rely on enough correlated metrics with the proposed approach.

3.8 Highlight the gap of knowledge

Prior work has been classified among five different groups, as mentioned in the above sections, work documented in around twenty articles, as exact related papers. Contributions as per each category were addressed as guidance, Extending Eclipse modeling EMF, model-driven code generator, RDF ontology model based, and OpenAPI model based approaches. Contribution of each category has already been explained in the previous sections with their limitations and gaps, as per authors themselves, and also it will be discussed from the perspective of this study, as followed below.

Guidance approaches in prior work [61, 33, 67, 19, 21, 36, 57] documented their methodology as a sequence of recommended stages, in each stage a common or standard libraries should be used. Some articles consider the contribution by following guides as an umbrella framework [23, 24], but technology selection without any automation or abstraction to improve productivity effectively, which can at most be considered as strategic shortest path, and it may be classified as minor improvement over a manual work, even though, these article haven't included an empirical assessment for efficiency impacts [67, 36, 57]. Tavares and Vale study [67] used abstraction, but with many manual implementations. Other articles used model to model transformation frameworks as a code generator [61, 33], such as one of Eclipse IDEIII, unfortunately both of them haven't done an empirical proof of concept. The implemented proof of concept for semi-automatic generation was not enough as per Costal, et al. [19] themselves. Even though Ed-douibi, et al. [21] approach is suitable for data retrieval only, and moreover it relies on a very complex tool, which limits the maintainability and support.

EMF based approaches in prior work [23, 24] have documented their methodology based on extending EMF as a model driven approach. Both of these studies have not used an empirical proof of concept, where Eclipse plugin just a tool, and can't be considered as an empirical proof of concept for the study of Ed-Douibi, et al. [23].

Model-driven code generator approaches in prior work [20, 29, 68, 69, 71] have documented their methodology based on generating REST APIs code for data models, automatically. All the published studies in this

context have built code generators to generate REST APIs with all necessary backend artifacts. Although of generators benefits, it has a major downsides on the development process, such as: huge initial efforts, code rigidity, possibility of losing code, technical complexity, and impacts engineering team and managerial levels, these downsides have been discussed with details in Section 2.3.4. Moreover, Wang, et al. [69] study feasibility has been only verified on object oriented databases, so it should be verified on relational database, which is most common and requires more efforts. Abstraction level in the study of Terzić, et al. [68] is not clear enough if it needs manual implementations or not, and also it needs empirical proof of concept.

RDF ontology model based approaches in prior work [62, 25], have documented their methodology on basis of RDF ontology with linked data model. Developed middleware in Serrano, et al. [62] study forms an extra stage, it correlates REST APIs with a special data query to handle CRUD operations using query style, which means extra efforts to implement, support, maintain, and train. Although of that, it needs verification by controlled empirical studies to quantify the effectiveness and the ease-of-use, as per authors themselves. Eng, et al. [25] have documented a flexible and robust methodology, but they use automatic approach to generate syntax code of REST APIs, which impacts the development process, as discussed in Section 2.3.4, and also it needs empirical studies to measure its effectiveness.

OpenAPI model based approaches in prior work [22, 63, 64], have documented their methodology based on OpenAPI model specifications. This approach strongly depends on the code generation methodology, rigidity is highly appears in addition to the other downsides of automatic code generation, since any new code generation may override customized code. Moreover, this approach in general relies on a very complex platform, it includes special grammar [64], which needs deep understanding, and continuous support with maintenance. From the results perspective, approach should be evaluated through a more realistic and larger case studies with a solid correlated metrics, as per authors Sferruzza, et al. [64].

In general, most approaches used code generation methodology, and on the other hand most studies have not verified the effectiveness by empirical proof of concept, evaluation of productivity and easy-to-learn is an essential to proof such of these approaches. Except the first group of studies as a guidance approaches, and Terzić, et al. [68], all the other

Table 3.4: Approaches Pattern.

Approach	**Used code generator**	**Applied effectiveness POC**
Guidance	1	0
Extending Eclipse modeling EMF	2	0
Model-driven code generator	5	3
RDF ontology model based	2	0
OpenAPI model based	2	0

researches used code generation, and on the same time, they haven't applied an effectiveness proof of concept, except few ones [20, 29, 69], where articles demonstrate this conclusion, as shown in Table 3.4.

The introduced approach consists of an umbrella framework to abstract layers of model operations, data access, business logic, and web APIs. Methodology use general and aspect oriented programming AOP, which generalizes REST APIs on basis of derived models at runtime and according to particular design patterns, this approach can be verified in a solid empirical proof of concept.

In this study, abstraction approach is adopted to generate REST APIs automatically, which can cover more than 90% of required coding. Designed patterns have been integrated in a framework as an umbrella to engage all the required utilities and other frameworks. Utilizing different frameworks in a unified and sustainable way, which enables developers to base their applications on the latest standard frameworks and tools, which can exploit others' efforts in different aspects while engaging it under the umbrella in a loosely coupled manner. The upper tiers of umbrella exposes a sustainable interface on which developers can build their layers independently in an effective way. This approach gains all benefits of other's work, and avoids rigid code of generators that have downside, where details can be found in research methodology and implementations chapters.

This approach abstracts complex, dummy, and repetitive coding in a reusable and upgradable manner to rapid REST APIs developments in easy way, which keeps built systems up-to-date and understandable when relying on a standard, common, and supported frameworks, such as SpringSTS, ORM, JAX-RS, and Modeling. However, traditional auto-

matic code generation will not be used absolutely, this framework will support most popular databases, and it can be easily extended for others and without restrictions or updates on methodology. Finally, methodology effectiveness will be verified empirically.

3.9 Summary

In the beginning of this chapter, importance of REST architecture have been explained as the main used approach now a days, where its usage have being increased exponentially as an excellent compliance with cloudy and IoTs platforms.

Critical literature review was adopted to recognize knowledge gap and to highlight limitations that can be considered in this research, search strings have been built and documented carefully to avoid missing or gaps. Search process has been applied over the main data sources and engines, such as Google scholar, IEEE Xplore . . . etc. collected articles have been evaluated, ranked, and filtered based on a clear include – exclude criteria, where the accepted studies have been classified in a groups, each group explained as an overall with a deep comparisons.

All groups have been discussed deeply to show gaps, limitations, and conflict of concerns, where most approaches used auto code generators, which has major downsides, as discussed in Section 2.3.4. Meanwhile, most studies still need more empirical a proofs to measure impacts of productivity and easy to use and learn.

Introduced approach consists of an umbrella framework to abstract layers of model operations, data access, business logic, and web APIs. Methodology has to use general programming and aspect oriented programming AOP, which generalizes REST APIs on basis of derived models at runtime and according to particular design patterns, this approach can be verified in a solid empirical proof of concept.

Chapter 4

Research Methodology

To determine the appropriate methodology, problem nature has to be investigated [60]. So, to address the specific nature of problem, research focus should be conducted through the research subject, objectives, and field, as per design categories of Pérez [51]. And based on the literature review and the proposed approach, hypotheses will be defined to determine experiment with the appropriate method and adequate components.

4.1 Introduction

It is very difficult for developers to build and deploy backbone services using data models with its data manipulations, since it needs deep knowledge, skills, and experience in different technologies and platforms, such as SQL Database engines, tiers architecture, application servers, and REST APIs, such these development skills can be intimidating even for seasoned develops. While in parallel, frequent changes and rapid release cycles in terms of market stress, and so limited time and resources makes gap between design and implementation phases. REST APIs development requires much time, and it is not-easy-to-use and understand. Therefore, development and maintenance for REST architectural software is a crucial, tedious, and error-prone work, and requires huge efforts.

In general, prior work adopted the auto code generation approach, but code generation has a major downsides on software development process. While in parallel, most studies still need more empirical studies to verify effectiveness. In this study, the main goal is the improvement of

productivity, maintainability, and learnability for REST services development to be easy-to-used and understand. Efforts will be dedicated to develop a useful framework to create REST APIs rapidly. Research will focus on Backbone of applications to build backend REST APIs, where the detailed objectives as the following:

- Reduce development time for backbone services, without impacting flexibility of architecture.
- Reduce time to learn and to understand, and simplify the required human qualifications and experience to build backbone services.
- Avoid restrictions for senior developers while using their own customized coding and implementations.
- Avoid downsides of auto code generators, such as extra efforts, code rigidity, code overriding, technical complexity, and limited margins for developers.

After completing literature review and specifying research objectives, as mentioned above, and in the next stage, the approach will be applied through developing a dedicated software framework that called RAAG for Rapid REST APIs Auto Generation, it can be used to a proof the concept of our approach by using empirical controlled experiment, as shown in Figure 4.1. But for the last two objectives that consider certain way of software design, these objectives will be examined by using a survey in a brief questionnaire.

Figure 4.1: Main phases of research methodology.

4.2 Experimental research

Experiments assist us to directly compare interested conditions, in this experiment, it will be RAAG Framework productivity, and how RAAG is easy to be used to create REST APIs. Conditions used to mitigate experiment bias or error risks to the minimum, the variable of used framework have to be controlled to obtain better indication about development time causation over RAAG framework or over other equivalent

framework, and also it will indicate how much it is easy to be used and understood. Experiments consist of three components: conditions, units, and methodology [40, 49]. Conditions will be discussed later as per independent variables, where participants and instruments form the units of this experiment, and methodology will based on generated hypotheses.

Number of variables and values is essential to adopt appropriate experimental methods, for instance one-level design is proper for one independent variable, but for two or more, factorial design is the best method. And so, one-level design is the appropriate method for our defined variable "Used Framework". Number of conditions may lead to use between-group or within-group design. In between-group, each group of participants corresponds to one condition, but in within-group, one group corresponds to whole experiment [40].

In this experiment, there is one single independent variable, thereby basic one-level is the appropriate design. As mentioned before, creating REST services is not an easy task and needs a lot of efforts for each condition, especially for condition of common used frameworks. Therefore, between-group is the adopted method to avoid fatigue, and learnability, and also to make more control over the effect of variable, where each group will be exposed by one condition tasks.

4.3 Experiment design

To decide design specifications of this experiment, hypotheses should be identified precisely, and then variables with conditions can be derived from the specified hypotheses, which is essential to proceed in the experiment design, as followed next.

4.3.1 Hypotheses

Based on the discussed results in literature review, the research hypothesis can be generated to derive variables and conditions. Where number of conditions equal: variables number x variable values [40]. Research objectives lead to formulate a certain hypotheses, where the below statements are formulated precisely as a null hypotheses, which used to investigate impacts of the proposed approach (*RAAG* framework) on the development time of REST services:

H1 *There is no significant difference in the development time for REST services when using traditional proposed RAAG framework, or other*

approaches.

H2 *there is a significant difference between development time (over the proposed approach) in terms of experience.*

Variables

Based on the generated hypotheses, H1 and H2, variables can be defined clearly to be measured accurately. Measured outcomes for the creation time of REST services acts as dependent variable. Controlled conditions of used framework acts as an independent variable.

Independent variables

That refer to change factors or causes on values of dependent variable, that are independent against a participant's behavior [49]:

Independent single variables is: *Used Framework.*

And possible values are: *RAAG framework or Different frameworks.*

Dependent variables

It refers to the concerned results or effects of experimentation, which dependents on the performance of participants or status of independent variables, and used to induce impacts of independent variables changes on dependent variables, such as efficiency of building REST APIs, and easy to learn and understand RAAG framework:

Variables: *(**i**) creation Time of REST services, (**ii**) Easy-to-use Factor.*

Conditions

After generating hypotheses and defining independent variables, conditions can be specified. In basic one-level design, possible values of the single independent variable "Used Framework" determine experiment conditions [40]. So, conditions are: common used frameworks and RAAG framework.

Between-group is the adopted method for the basic design, in which each participant corresponds to only one condition, so learnability is avoided, since user will not learn from prior task conditions, so it is cleaner [40]. Moreover, participants avoid tiredness or fatigue since each one have to complete tasks of only one condition on the contrary of

Table 4.1: Cross-Group Design.

Group	Used Framework	Tasks
1	RAAG Framework	Create (Student & Parent) REST APIs
2	Different Frameworks	Create (Student & Parent) REST APIs

within-group design. But in between-group, individual differences may variant results and then noise evaluation of group. Noise can be mitigated by increasing number of members in the groups, but it leads to large size of sample, which is negligible in this experiment [40]. Hence, as per the mentioned facts above, groups will be designed as illustrated in Table 4.1.

4.3.2 Procedure

After identifying the hypotheses of research, and specifying its design, we should sufficiently prepare for the experiment to mitigate risks and avoid obstacles, and then application stage can be started successfully.

- Start design of tasks carefully to be adequate and short as much as possible.
- A pilot study will be run to find snags and missing issues by testing the design pre-configure required IDEs and database schema.
- Implement training material as five-minute movie, short document, and test procedure.
- Setup study instruments in the lab using VM ware to encapsulate all requirements in isolated environment with a complete similarity, such as OS, installed databases, and IDEs.
- Select and recruit participants carefully, and schedule the session.
- Start session by running the prepared movie and then monitor progress to record spent times using stop watch instrument.
- Analyze collected data by using MS-Excel to report the results, as discussed in the next Section 4.4 and Section 4.5.

4.4 Data Collection

Experimenter will measure spent time for each participant, using stop watch instrument, where tasks are the same for all of them, but over two different frameworks, so experimenter will record all these information as per each participant.

4.5 Data Analysis

MS-Excel will be used to analyze result, which is adequate tool for such analysis. Analysis will focus on the causal relations between required time to create REST services and the independent variable, which should be the used framework to create REST services.

To verify realization of first two research objectives, quantitative analysis will be used on the gathered data from the experiment to investigate the effect of using RAAG framework on the REST services creation time, and the same time can be used to indicate easy of learn and use for RAAG framework. Analysis for survey results will be used to verify achievement of design objectives, as per the last two objectives.

Chapter 5

Implementation

This chapter discusses the empirical implementations, we firstly discuss framework specification, since *RAAG* framework is the developed platform that used to execute all other empirical exercises. After describing the framework, we give an overview about the reference application as a pilot example, which demonstrates framework capabilities to enable junior developers to build a difficult applications with short time and high quality of code.

5.1 Developed solution - RAAG Framework

REST APIs Automatic Generation framework reduces the required manual coding and provides high quality of design and implementation. Major value added features were considered in this framework, which will be discussed in the next sections. FW have utilized common technologies and platforms, it combined them into one frame as a platform for back-end developers. And on the other hand, the development environment enabled using different useful and helpful tools as plugins, these tools facilitate the integration with data sources and models to simplify repetitive coding. Implemented diagram of the architecture and design visualized all the built components and employed aspects in the developed framework, as followed next.

5.1.1 Value added features

Introduced solution is applied through an integrated framework, which combines most needed components and aspects into one place as infras-

tructure platform. Architecture built in a specific way and in a loosely coupled with certain technologies to achieve and realize below features and advantages:

- Compliance with Parallel Agile (PA) approach for parallel development among teams [14], which facilitates allocation of teamwork tasks into sprints.
- Compatibility with new trends of technologies and service platforms, such as Microservices architecture, Cloud services, NoSQL and Big Data platforms, where REST became the key component that providing an infrastructure for IoT world.
- Centralism in this FW enables easy control to manage logging, security, and permissions.
- Realized unification enables architects to build and integrate unit test and also to automate testing on back-end level.
- Reusability for work of others in easy way, where the FW already integrated the most common and useful utilities and frameworks.
- Unify code style that may be implemented by different developers' levels, which makes the codes easy to read and understand, since the applied abstraction technique coped complexity [59], whereas automation is the most defective method to boost productivity with quality.
- Minimize efforts to learn and simplify required experience and qualification to use complex frameworks, where novice developers can utilize these platforms indirectly over this FW.
- Provide enough margin for senior developers to innovate their techniques and tools while gain a complete benefit and support from RAAG framework in the same time and in parallel without any conflicts or restrictions.

5.1.2 Technologies and platforms

RAAG framework reused the following platforms and technologies in a unified and decoupled way, framework integrated whole packages of them as one infrastructure layer, and plugins of them were integrated in the used STS Eclipse IDE:

Spring Framework

It provides a platform to develop Java applications, and handles the below infrastructure as abstract to focus development efforts on the application layer. Spring enables building apps from POJOs as J2SE and J2EE models. For example, method can be implemented to execute in a database transaction without need to deal transaction APIs explicitly, and also invoke procedure without need to deal with remote APIs, which is same for JMX and JMS APIs [38].

Hibernate ORM

Hibernate is a solution to map Java object with relational data entity, where working on OO and DB is a time-consuming due to mismatch between object's data versus relational entity. Hibernate maps classes to tables, and covers Java attributes' types to SQL data types. In addition, it significantly facilitates CRUD operations and eliminate manual work.And so, Hibernate is useful for object-oriented domain models, which encapsulates SQL and translates result into objects' graph instead of tabular [4].

Jackson Object Mapper

It is a solid library that serialize/de-serialize JSON to Java. Where Object Mapper responses JSON objects by using annotations. Hence it can parse string or stream into JSON, and vise versa create Java object from JSON [7, 8].

Springsource Tool Suite (STS)

It is a preinstalled plugins on Eclipse IDE to provide supplementary features for Spring developers, such as validators, visual editors, and dashboards [3]. Where the main STS plugin is Spring IDE that provides Spring tooling features, STS is already configured with other helpful plugins, such as Maven, JUnit, Data Tools Platform (DTP), Web Tools Platform (WTP), and AspectJ Development Tools.

Eclipse DTP (Data Tools Platform)

It gives developers standard tools to interact any SQL database over Eclipse IDE. DTP is a vast domain by which developers can easily integrate database in the environment to manage data, after making the

connection, developer can explore data source, make and debug changes through CLI or GUI actions. Therefore, it bridges gap between relational and object, or other structures [2].

JBoss Hibernate Tools

It is an umbrella that supports JBoss and related technologies for a group of plugins inside Eclipse, such as Hibernate, JPA, Apache Camel, Maven, (X)HTML, CDI, JSF, JBoss AS, OpenShift, Docker, Red Hat JBoss Fuse, and more [2], where RAAG framework employs JPA tool [9].

Apache Maven

It is a project management tool to uniform building of Java based projects. Maven sets a convention over the configurations through a project object model (POM.XML file) for each project, which holds all project's information. Maven has a plugin for Eclipse IDE. It is used to manage different utilized tools and frameworks, which allows developers to understand the complete state of a development progress in the shortest time, where it simplifies build process to uniform built system, and introduces quality project information with guidelines for best practices in development [11].

5.1.3 Integration

In this solution, Spring framework features are integrated into Eclipse IDE through STS, hence developer can simply use Spring feature and focus on application logic. DTP is also plugged into IDE to integrate SQL database, and then it facilitates data management from the environment side. Moreover, DTP offers a database connection for JPA tool to fetch schema meta data for the intended entities. Where JPA tool is a JBoss Hibernate Tool that used to generate data model from database entities, then model can be used from both sides, Hibernate ORM and Spring business logic. On the other hand, Jackson object mapper used over REST APIs to marshal and de-marshal data Model into JSON syntax object.

5.1.4 Architecture and design

One of the most important phases in software development life cycle is the design, it impacts other phases, and it influences support alongside

with maintainability, quality, and performance. In usual, design quality depends on developer experience and expertise only, so this framework applies principle, standards, and patterns, which can improve the reusability, maintainability, and scalability.

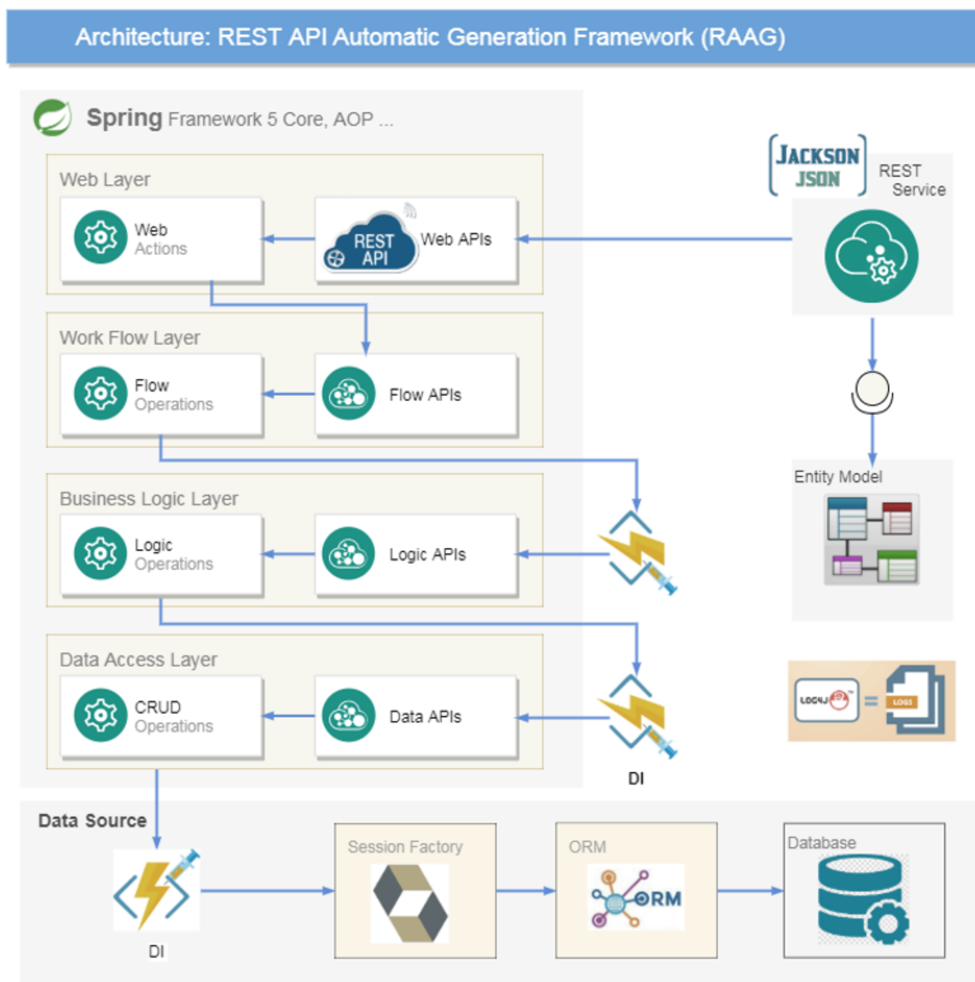


Figure 5.1: Framework Architecture (RAAG).

Framework architecture consists of three main aspects that are; data source, core framework, and data services with their model. In general, the three aspects are based on Spring Framework version 5 as a platform that abstract all infrastructure implementations, as shown in figure 5.1.

Data source aspect is the responsible of handling data from its marts, this part is based on Object relational Mapping framework to parse and marshal data entities into Java objects, where Hibernate framework is

used to manage these roles. ORM functionality and database engine can be replaced with any equivalent tools, or it may be developed from scratch if needed. Therefore, data source is bridged with core aspect in a loosely coupled mode, which realize an extreme flexibility, where dependency injection DI of Spring is applied as aspect oriented architecture.

Core aspect is based on four layers architecture, layers ensure separation of concerns into data access, business logic, work flow, and web APIs. Each layer services the above one, and it is separated with Spring dependency injection DI to enable customization and replacement with different layer or layers.

On top of web APIs tier, module of "data services and model" is designed to handle common web APIs over each entity model, then it can expose common APIs as-is, or override, customize, or expand them based on system and software requirements. This architecture enables to abstract database CRUD operations at least (Create, Retrieve, Update, and Delete), which already done, and more common operations can be implemented and encapsulated as common APIs in web tier.

Therefore, this architecture can speed REST APIs generation and reduce their development time and required efforts. Moreover, it achieves many constraints and non-functional requirements, such as flexibility, Agility, simplicity, Reusability, maintainability, and other more values that mentioned in section 5.1.1. Efficiency of this methodology and its impact on the process of software life-cycle have been investigated and discussed, as shown in section 6.3 and section 7.2.

In general, the introduced framework is aligned with SOLID principles [44, 16] that lead to high quality of software according to the empirical validation of Singh and Hassan [65]. The principle of *Single Responsibility SRP* is applied to limit responsibilities inside each layer to coherence interactions and to cancel tightly coupled relations. *Open Closed OCP* is used to mitigate tied relations between extended classes as REST services and their based class in web APIs layer, where the abstract code is closed to intact designed coherence. *Liskov Substitution LSP* is commonly utilized, especially in data model, where the entity interface exposes wanted methods only. Core layers are segregated according to *Interface Segregation ISP* principle to ensure proper inheritance structure and polymorphism, which later avoids unmatched interaction between incompatible objects. *Dependency Inversion DIP* is used to minimize inter-dependencies between related aspects, such as data resource,

data-access, and business logic, which can be injected on the instantiate time.

5.2 Proposed approach

According to the developed solution, as discussed in the prior sections, developers can build or maintain REST web services in a simple and professional manor without impacting code quality. The following steps summarize the recommended procedure to build REST APIs in a short time:

1. Setup the environment as mentioned in section 5.1.2 over STS Eclipse IDE while using Maven project.
2. Integrate and configure the auxiliary tools and plugins to support the prepared IDE, which required to be done for one time only, as discussed in the integration section 5.1.3, where ORM Hibernate support DB interactions and data tool with JBoss Hibernate tools help to generate data model.
3. Generate model source code by using the integrated tools in IDE, where it can be used to connect with database and then generate model classes for the selected tables.
4. Implement abstract methods in data model to be compatible and plug-able into *RAAG* framework, these methods are empty or very simple in default, as shown in appendix C, section C.2.1.
5. Extend Web APIs layer, which discussed in section 5.1.4, where SpringWebAPI can be extended to create the required service based on certain model object/s, default empty service class inherits all the common predefined APIs, such as CRUD and search APIs, as shown in section C.2.2.
6. Finally, implementer can customize or maintain the created service by adding to the existing functionality or overriding the predefined APIs, which also can be expanded inside framework to include an other common functions.

5.3 Objectives realization

In this section, would like to pay attention for the realized and achieved objectives, using *RAAG* framework implicitly fulfills the following objec-

tives of this research:

- *Avoid restrictions for senior developers:* based on this architecture, seniors can expand the already predefined REST APIs (CRUD), or they may replace them, and simultaneously they can utilize *RAAG* framework from capabilities, furthermore seniors can customize or replace complete layers on basis of AOP.
- *Avoid downsides of auto code generators: RAAG* employs general programming for REST services without any code generation, where model code is trivial and shouldn't include any logic, it can be accomplished for only one time regardless of approach.
- *Enable loosely couple for other frameworks or replace them smoothly:* as mentioned above in this section, a complete aspects can be replaced, such as data source, and also backbone tools, frameworks, or platforms can be replaced without any impact on the built APIs, such as Spring.

5.4 Reference application

In this section, introduced an overview about the reference application, where its back-end was developed over RAAG Framework, this project is supported by EASY SOLUTIONS INFORMATION TECHNOLOGY company [5]. Overview will focus on four perspectives; structure of data services with model, complexity of user interaction, code quality, and the required experience for implementation.

The structure of data model is not a flat level, but it includes too many relations with nested levels (inheritance) and different types. Where the applied relations are one-to-one, many-to-many, one-to-many, many-to-one. As usual, model consists of entities that include attributes with their setters and getters, which is trivial code and may be generated automatically by JPA JBoss Tool, as discussed in section 5.1.3. Although of complex model, design implementation for REST APIs services was very simple, as shown in figure 5.2.

From the perspective of user interaction, application is computerized to manage relationship between students, teachers, and parents. Therefore, school management system should allow students to view their grades, attendance, classes, and interact with their teachers and vice versa. Furthermore, parents allowed to keep track with their children's grades, behavior, and financial records. All the mentioned above confirms that

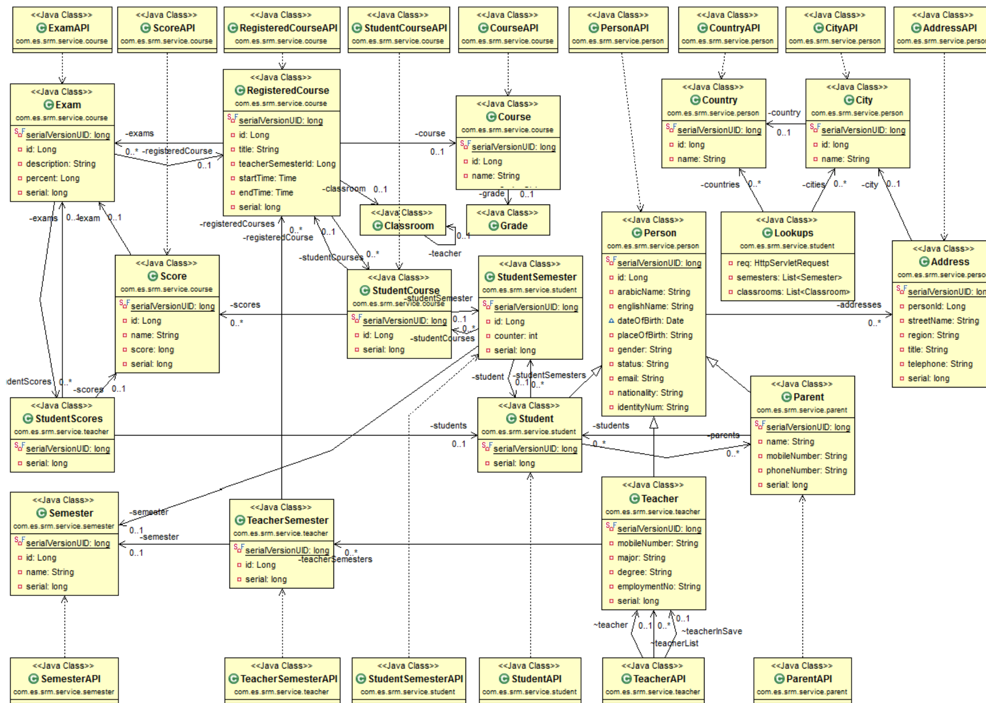


Figure 5.2: Class diagram for reference application.

is not an easy application, as shown in figure 5.3, where screenshots spirit that developing of back-end requires certain level of experience to be implemented.

For required experience, although of the reported difficulty in the system requirements, a fresh engineer have implemented all the back-end REST APIs for the application, he was supported with a brief recommendations only. Meanwhile, design of the implemented APIs is aligned with all essential and SOLID principles [44], as discussed in section 5.1.4. Thereby, design treats layered architecture, high coherence with low coupling, batched transactions, and concurrency modes. In addition, code seems very easy and simple, which indicates an excellent impact by using the proposed approach.

For code quality, it is based on packages of RAAG framework, so developed REST code will gain all characteristics of FW. In particular, code metrics [18] shows that most classes of REST services are default without any customization, these only include two empty methods. Hence, 85% of REST source code services are empty (13 services out of 15 ones), which means a very low complexity in REST code, as shown figure 5.4.

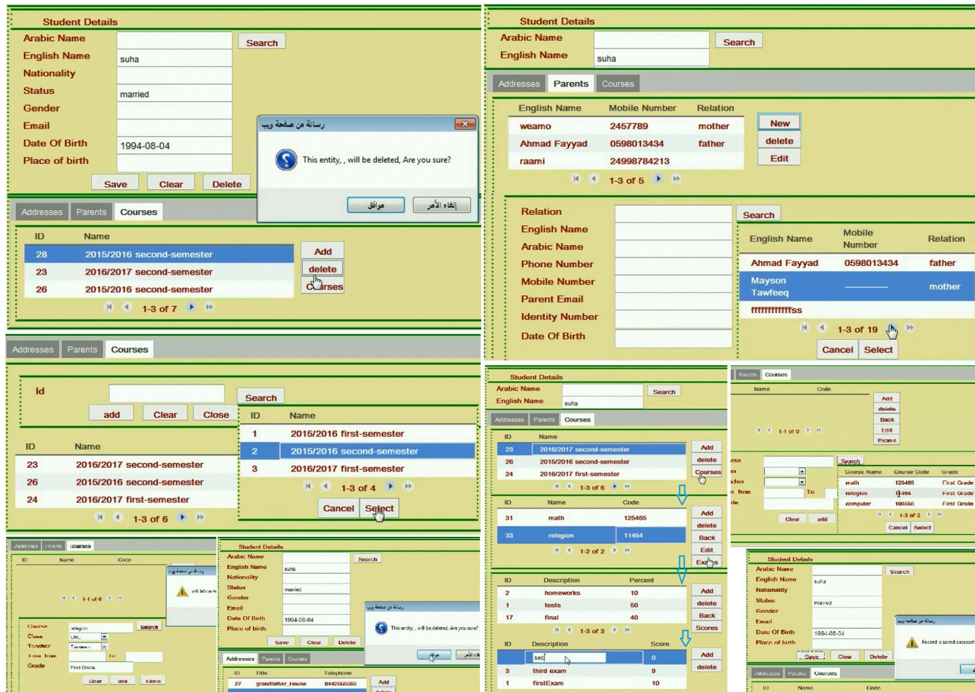


Figure 5.3: Screenshots for reference application.

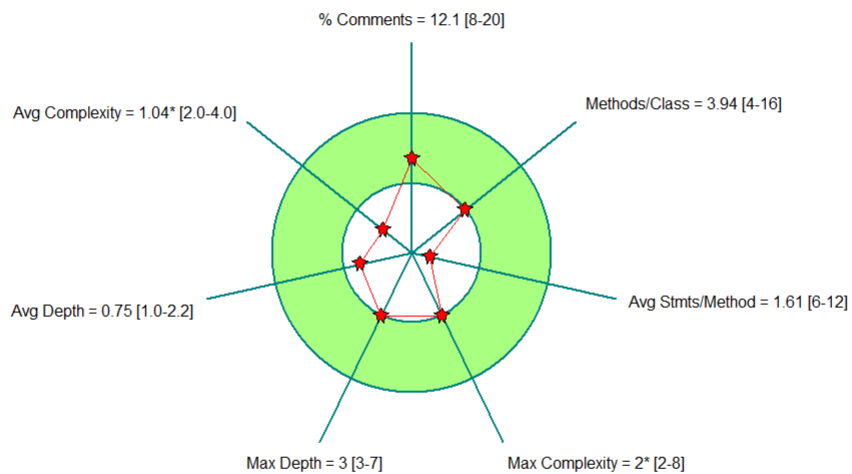


Figure 5.4: Code metrics for reference application.

Chapter 6

Evaluation

In this chapter, evaluation is discussed over the implemented framework, while using two empirical approach, survey and experiment. An introductory training material for the involved participants will be described, especially for who use the proposed approach over the implemented framework, material is focused around an illustrative example over the framework, which presented in a movie to learn experimentees. Similar to the learning example, the assignment of experiment is described to demonstrate a comprehensive scenarios and strong conduction with research objectives.

6.1 Model example as learning material

This section represents the prepared training material for participants of the experiment, learn by example is the followed approach. The selected example is a vehicle model that consists of Vehicle, Car, Part, Factory, Country, and City entities. Model example is designed in a certain way to cover all possible relations between entities, where making a comprehensive awareness for participants enables them to implement any exercise. For example, Vehicle is a master data entity, which joint with Car entity as detailed composition in a one-to-one relation, also Car refers to Part through many-to-many breakdown entity, and so on. Entity relational diagram illustrates all these relations, as shown in figure 6.1.

Model and services class design is implemented over the predefined database entity relational structure, design views mapped objects with entities and built services over them. From class perspective, Vehicle is

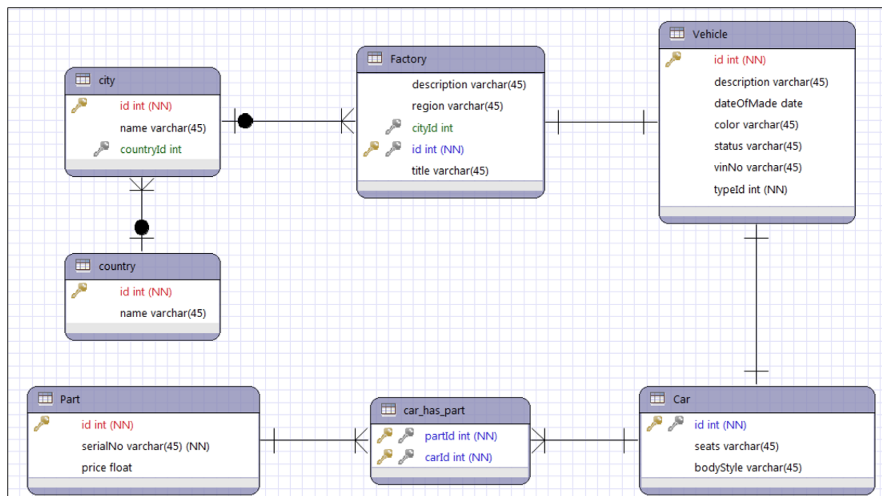


Figure 6.1: ER diagram for learning purpose.

a super class for Car child, which contains list of Part objects and vice versa. Object design structures an integrated REST service for Car and Part with relevant model, as shown in figure 6.2.

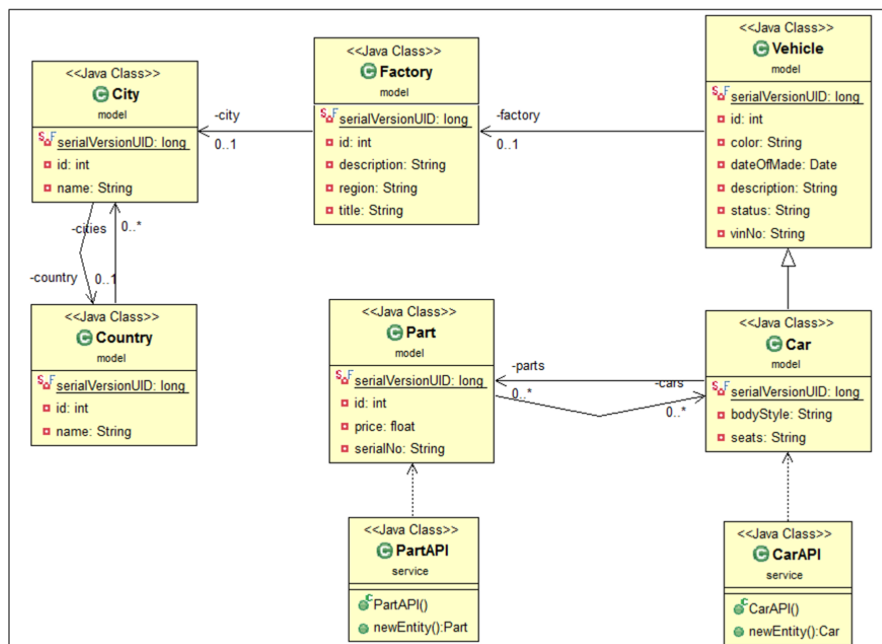


Figure 6.2: Class diagram for learning purpose.

On basis of the discussed example, we have documented diverse types

of materials to introduce a complete and compliance material with all learning styles. A shore 15 minutes movie was uploaded on YouTube [12] to provide fast, parallel, similar, and sustainable material for each participant. Moreover, Java source code and Java API documentations were packaged and provided for participants.

6.2 Model exercise for experiment

According to experiment design in section 4.3, specific exercise should be designed as a comparable task between the two groups of participants, the prerequisite knowledge should already be transferred to them through the prepared material, as discussed in the last section 6.1. The designed exercise is a student model that consists of a Person, Student, Parent, Address, Country, and City entities. Model exercise is structured in a specific way to cover all possible relations between entities, building a comprehensive exercise is a very important task to measure valuable indicators between groups. For instance, Student is a detailed data entity, which joint with Person entity as master composition in a one-to-one relation, also Student refers to Parent through many-to-many breakdown entity, and so on. Entity relational diagram demonstrates these relations, as shown in figure 6.3.

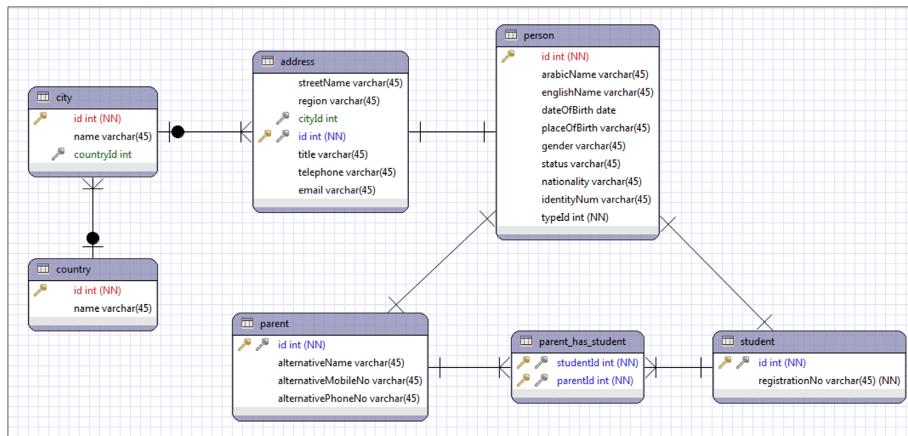


Figure 6.3: ER diagram for experiment exercise.

Experiment model with relevant services is designed over the predefined database structure while considering relations, class design views mapped objects with entities and built services. From class perspective, Person is a super class for Student child, which contains list of Parent

objects and vice versa. Object design shapes an integrated REST services for Student and Parent with their relevant model, as shown in figure 6.2.

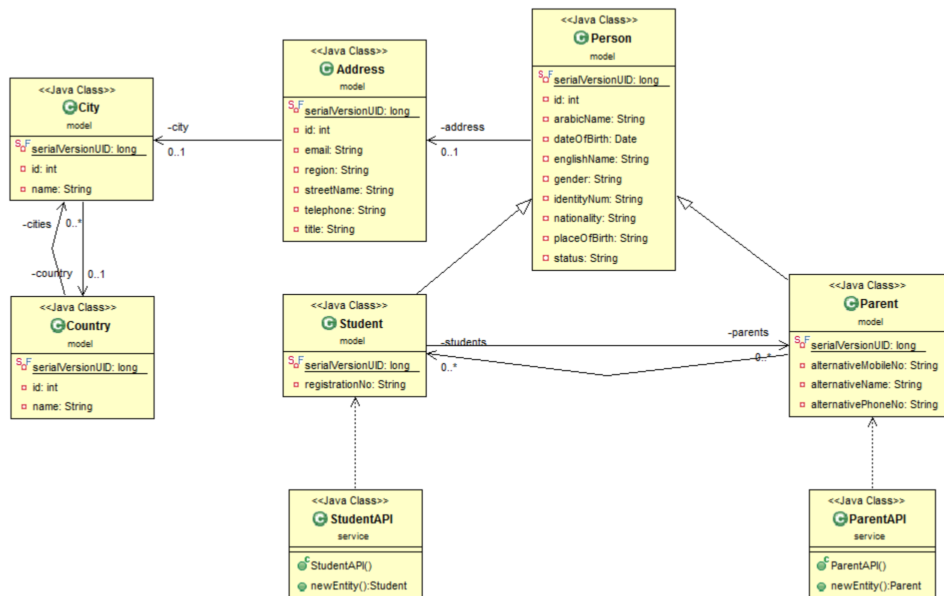


Figure 6.4: Class diagram for experiment exercise.

According to the discussed experiment design, all required materials are packaged and documented for both groups. In addition to the learning materials that discussed in section 6.1, we provided model ER diagram for participants in both groups, and give a survey (see section A.2) for the first group. Furthermore, database dump was introduced for participants in the second group (as SQL script) who use common approaches, such as Spring Boot, Apache Camel, Python, ... etc.

Design considered reasonable time to implement the experiment, a trial test was done to ensure design validity and to guarantee environment readiness, where the test procedures were also prepared. Environment setup is encapsulated in VMware instance to keep similarity and portability for whole platforms with setting, such as OS, DB, and IDE with plugins.

6.3 Survey

A survey was introduced for the experimentees in first group, those who obtained training material about building REST services, as mentioned

in section 6.1. Questions of survey were designed to explore opinions around three main topics; sufficiency of learning material, reliability of experiment, and framework quality. Around the first topic, opinions of the involved implementers was very useful to indicate achievement range for the below objective:

Objective: *Reduce time to learn and to understand, and simplify the required human qualifications and experience to build backbone services.*

Gathering opinions around the other two topics was a supportive indicator, these opinions used to demonstrate correspondence between experimentees' expectations and design assumptions, which assumes objectives realization when applying the proposed architecture, as discussed in section 5.3.

Point of scale	1	2	3	4	5
Agreement	Strongly disagree	disagree	undecided	agree	Strongly agree

Figure 6.5: Likert scale options.

Table 6.1 illustrates the statistical information of the introduced survey, which have been analyzed based on Likert rating scale, where Likert rating scale applied five options, which included a neutral option [41, 10], as shown in figure 6.5. Data was gathered from seven participants after finishing their experiment over the proposed approach, where the following sections will analyze and discuss survey statistics.

6.4 Experiment

The objective of the designed experiment is to investigate the performance of building REST services, where the development is based on the proposed approach, and over the presented *RAAG* framework, which mentioned in section 5.1. This experiment is part of a larger investigation, since impacts on development life-cycle may be examined from different perspectives over the proposed approach, such as productivity, flexibility, maintainability, compatibility, ... etc. Several empirical experiments may be done, so a survey was implemented to interrogate around other perspectives, as discussed in section 6.3.

The experiment was executed though two weeks, a specific exercise was assigned for all participants in each group, as discussed in section 6.2. The collected data was recorded into excel sheets. A prepared survey

Table 6.1: Survey Statistics.

Category	Question	Mean	SD
Sufficiency of Training Material	The training session (such as demonstrative video and docs) was clear.	4.14	0.69
	The information in the training video was helpful to complete the tasks and experiment.	4.57	0.53
	I needed more help to complete the tasks during the experiment.	2.14	.069
Average		**3.62**	**0.64**
Experiment Reliability	The required exercise included real-world scenarios.	**4.43**	**0.53**
Framework Performance and Quality	I am satisfied with how it was easy to use this framework to build REST API.	4.86	0.37
	This framework minimized usual efforts to build REST APIs.	4.71	0.48
	This framework reduced the time needed to create REST APIs.	4.57	0.89
	Making changes to the generated REST APIs required little time.	4	1
	I could fix problems easily and quickly in this framework.	4.14	0.69
	I believe that proposed framework can improve productivity.	4.7	0.48
	This framework has all the functions and capabilities that I expect to build REST APIs.	4	0.58
Average		**4.43**	**0.63**

was introduced at the end of each experiment, survey designed to interrogate participant around the introduced practice and around the followed approach. Most experiments were run within working hours in an enterprise environment, while others were run outside an enterprise. Therefore, coordination within an available time slots had spent two weeks to execute experiments.

6.4.1 Participants

Before starting the experiment, developers who are willing to share in the exercise have to be found. Actually, it's very important to choose

developers who are motivated and willing to share in whole experiment, thereby it is preferable to select developers whose work is similar to the field of experiment. Participants were allocated into two groups, each group includes diverse individuals in the years of experience, following sections will discuss setup of groups.

6.4.2 First group

First group involved seven participants, group will use the proposed approach to build the required Student model and REST web service on top of the specified model, which is exactly according to the experiment exercise 6.2. Three guys of the group are fresh graduate engineers, they work as trainee since few weeks in ASAL Technologies company [1]. An other three engineers have long experience, more five years, where most of their experience is relevant to the exercise. The last developer has one year of relevant experience, for more information about participants refer the detailed in table A.2.

6.4.3 Second group

Second group involved four experience engineers, this group considered different approaches, where each participant use his approach according to his experience. Database dump was provided for each one, dump precisely leads to structure the intended model for Student and the REST web service on top of that model, just like the exercise of a first group. Hence similar test procedure can be used to avoid threats of validity. Two developers of this group are experts in their followed approaches, one of them used Spring Boot framework and the other used CherryPy with Alchemy ORM in the environment of ASAL Technologies company [1]. The other two engineers have mid experience, the first has two years in ASAL. The last participant has one year experience, further information about participants is detailed in table A.3.

Chapter 7

Results and discussion

In this chapter, results were discussed and analyzed in depth, based on the introduced evaluation in chapter 6, results of the survey and experiment will be figured to deduce and illustrate trends, and also to overview statistical data for different approaches, where hypotheses have been tested to a proof the contribution of proposed approach. Moreover, threats to validity are explored to describe the considered actions to mitigate negative impacts.

7.1 Survey Results

Following sections will discuss survey results, which concentrated on three main topic; learning material, experiment reliability, and framework quality.

7.1.1 Sufficiency of learning material

As illustrated in figure 7.1, There is a high values of responds around learning material, which indicates a good understanding of proposed approach and its usability before starting the experiment, as shown through Mean (4.14), and through standard deviation (SD=0.69) of below question:

Q1: *The training session (such as demonstrative video and docs) was clear.*

As well as second survey question, the Mean = 4.57, and SD = 0.53, which is:

Q2: *The information in the training video was helpful to complete the tasks and experiment.*

For third question, the results of answers are (Mean =2.14 and SD= 0.69), which confirm the understanding of learning material, it shows the intermediate percentage of seeking help from the participants who performed the experiment, this degradation mostly refers to the high complexity of model exercise, since model was designed to include all types of relations to simulate real-word scenarios:

Q3: *I needed more help to complete the tasks during the experiment.*

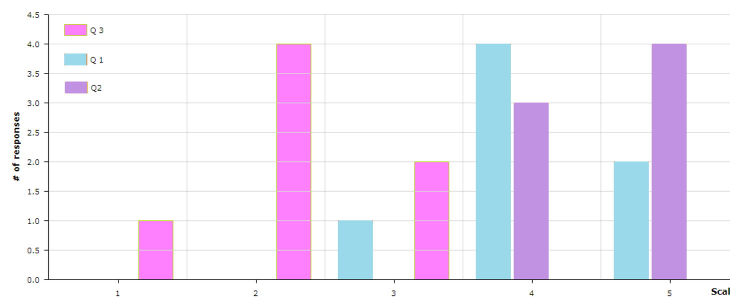


Figure 7.1: Sufficiency of learning material.

7.1.2 Reliability of experiment

The comprehensiveness of the proposed solution can illustrated after implementing the experiment that simulates real world scenarios, as shown in figure 7.2, where results of assessment are (Mean =4.43, and SD=0.53) for the fourth question.

Q4: *The required exercise included real-world scenarios.*

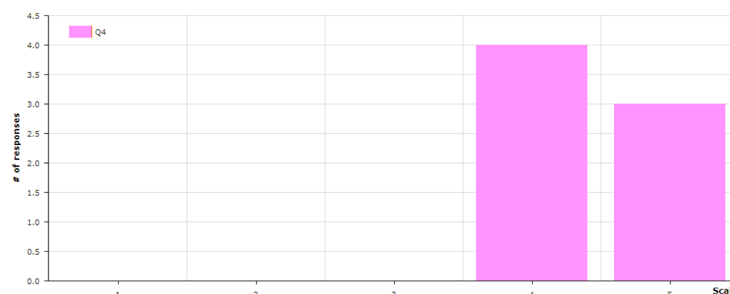


Figure 7.2: Reliability of experiment.

7.1.3 Framework quality

Figure 7.3 and figure 7.4 demonstrate results of participants' assessment for framework performance and quality that based on study approach, where most ranges of respondents are between 5 to 4 out of 5, which means high participants satisfaction.

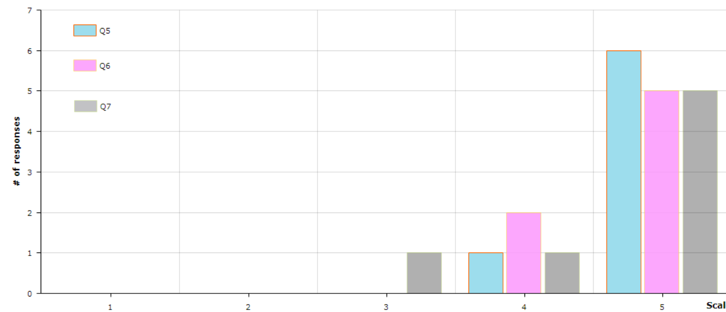


Figure 7.3: Framework quality.

The most of "agree" responses appear in questions 5, 7, and 10, as shown in table 6.1 with Means (4.86, 4.57, 4.7) and SD (0.37, 0.89, 0.48), respectively. Hence, responses indicate that the developed framework is easy to use, and it reduced the required time and efforts for REST development, which leads to improve software productivity.

Whereas the lowest agreement was with the question eight:

Q8: *Making changes to the generated REST APIs required little time.*

However, it's a good evaluation rate, Mean = 4 and SD = 1.0, where some of participants are fresh graduate, and work as a trainee on front-end aspect.

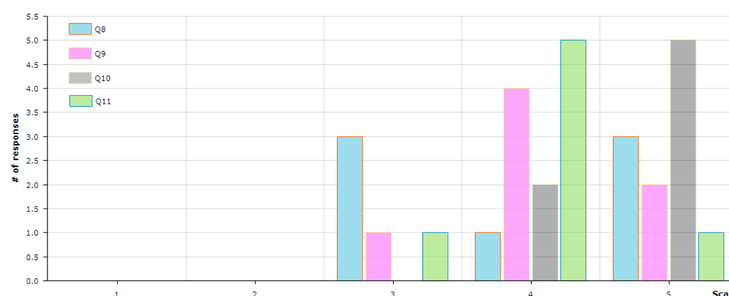


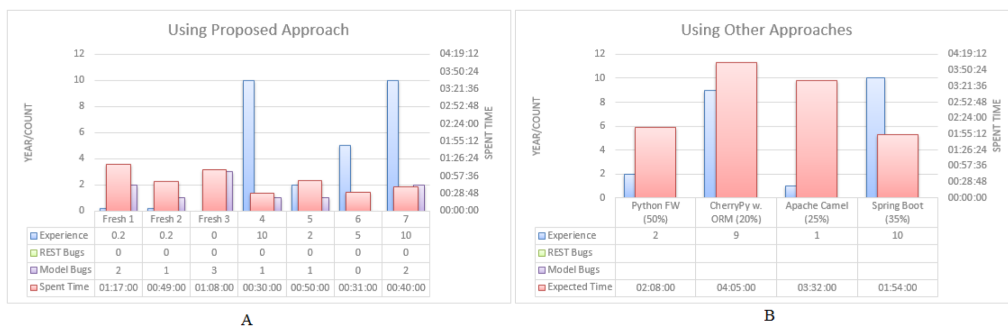
Figure 7.4: Framework quality.

7.2 Experiment Results

7.2.1 Collected data

Data gathered face to face during an interview with experimentees themselves, which achieved a complete useful responses. Experimenter recorded spent time, progress percent, years of experience, number of help requests around model implementation, number of help requests on REST development, count of REST bugs, count of Model bugs, and the worthy comments. Data is detailed for the two group into table A.2 and table A.3.

Data visualization demonstrates a significant variance on the required time between the two groups. Although of high prior experience in the second group, the mean of spent time is more than twice of the first group, and also prior experience in first group is not a prerequisite, where the group involved fresh engineers. Furthermore, data of the first group shows easiness of learn, where a little impact on the required time versus many years of experience, which seems as neutral factor in the first group. For REST development aspect, charts demonstrate zero help requests with zero bugs, except one brief help for two of the fresh engineers, as shown in figure 7.5.



	Fresh 1	Fresh 2	Fresh 3	4	5	6	7
Experience	0.2	0.2	0	10	2	5	10
REST Bugs	0	0	0	0	0	0	0
Model Bugs	2	1	3	1	1	0	2
Spent Time	01:17:00	00:49:00	01:08:00	00:30:00	00:50:00	00:31:00	00:40:00

	Python FW (50%)	CherryPy w. ORM (20%)	Apache Camel (25%)	Spring Boot (35%)
Experience	2	9	1	10
REST Bugs				
Model Bugs				
Expected Time	02:08:00	04:05:00	03:32:00	01:54:00

Figure 7.5: (A) Experiment results for G1 and (B) Experiment results for G2.

7.2.2 Analysis

Results analysis will focus on the impact of used approach on development time, and so causal relation should be investigated between independent and dependent variables, as used approach and time creation

for REST services, respectively. Quantitative analysis is used to realize and confirm below research objectives:

- *Reduce development time for backbone services, without impacting architecture, flexibility, and maintainability.*
- *Reduce time to learn and to understand, which simplify the required human qualifications and experience to build backbone services.*

Above mentioned objectives lead to formulate two or more hypotheses, which is very important to know and state what have to be evaluated in this experiment [70]. Here, it is determined to focus on the below hypotheses, informally:

- ***H_1:*** *Developers from first group have used the proposed approach, and hence it is expected that they have a faster productivity than their peers from other group.*
- ***H_1:*** *there is no significant difference between development time or raised bugs (over the proposed approach) in terms of experience.*

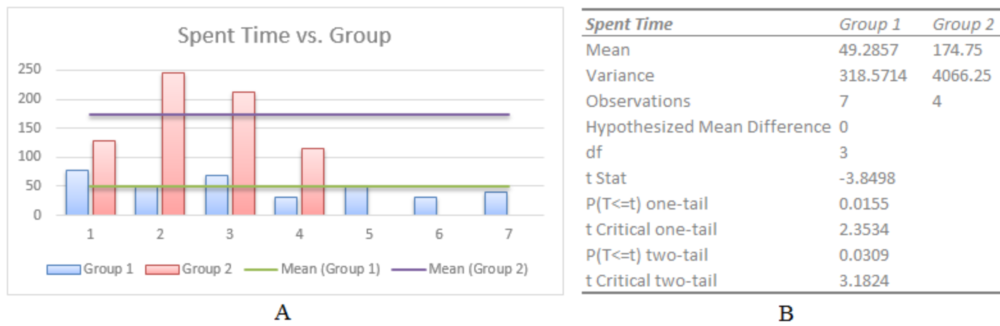
Based on the above informal hypotheses' statement, it is possible to formulate the hypotheses and define the required measures for evaluation, as shown below:

- ***Null Hypothesis (H_0):*** *there is no significant difference between REST development time in terms of used approach. (Spent time is independent of used approach)*
  Alternative Hypothesis (H_1): *REST development time changes with used approach.*
  Required Measures: *Spent time and followed approach.*

- ***Null Hypothesis (H_0):*** *there is a significant difference between development time or raised bugs (over the proposed approach) in terms of experience. (Spent time and raised bugs depend on experience for first group)*
  Alternative Hypothesis (H_1): *REST development time and raised bugs are independent of experience for the first group.*
  Required Measures: *Spent time, raised bugs, and years of experience.*

We will use descriptive statistics to visualize collected data, and then analyze results, as shown in figure 7.7, and figure 7.8.

Figure 7.6 shows the spent development time for the two study groups, as per each participant. From figure 7.6, it is obvious that developers from first group (G1) seem to spend a shorter period. Moreover, it is



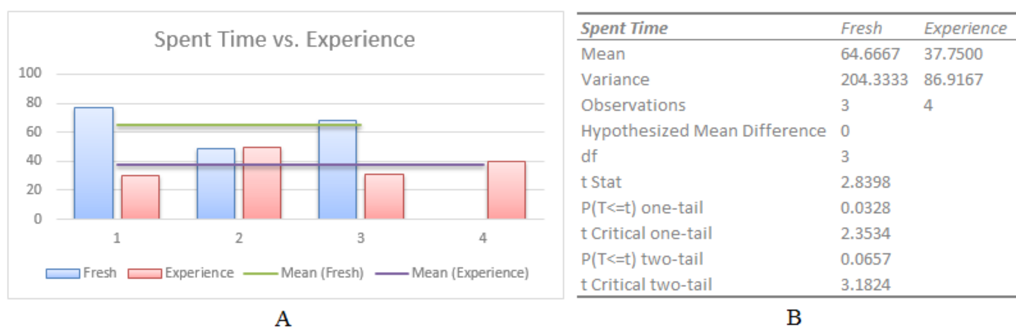
Spent Time	Group 1	Group 2
Mean	49.2857	174.75
Variance	318.5714	4066.25
Observations	7	4
Hypothesized Mean Difference	0	
df	3	
t Stat	-3.8498	
P(T<=t) one-tail	0.0155	
t Critical one-tail	2.3534	
P(T<=t) two-tail	0.0309	
t Critical two-tail	3.1824	

B

Figure 7.6: (A) Spent time in G1 vs. G2 and (B) Time difference calculations.

noticeable that the distribution variation seems to be larger among the developers from the second group (G2). In total, there are seven G1 participants and four in G2. Where the mean time value for G1 developers is 49.3 minutes with a standard deviation of 17.9, and for the G2 mean time is 174.8 minutes with a standard deviation of 63.8. Huge mean's variance between G1 and G2 inspires that there is more than two times improvement on the productivity of G1. Thus, it is possible to investigate the difference statistically in the first hypothesis test.

On the contrary of time results, years of prior experience is not a prerequisite in first group, where mean years of experience in G1 is 3.9 Years, and 5.5 Years for second group. Results means that first group superior with less time and less prior experience too.



Spent Time	Fresh	Experience
Mean	64.6667	37.7500
Variance	204.3333	86.9167
Observations	3	4
Hypothesized Mean Difference	0	
df	3	
t Stat	2.8398	
P(T<=t) one-tail	0.0328	
t Critical one-tail	2.3534	
P(T<=t) two-tail	0.0657	
t Critical two-tail	3.1824	

B

Figure 7.7: (A) Spent time in G1.1 vs. G2.1 and (B) Time difference calculations.

Further, the first group (G1) was divided into two sub-groups, G1.1 and G1.2. The sub-group G1.1 has the 4 experts participants, while G1.2

Table 7.1: Experiment descriptive statistics.

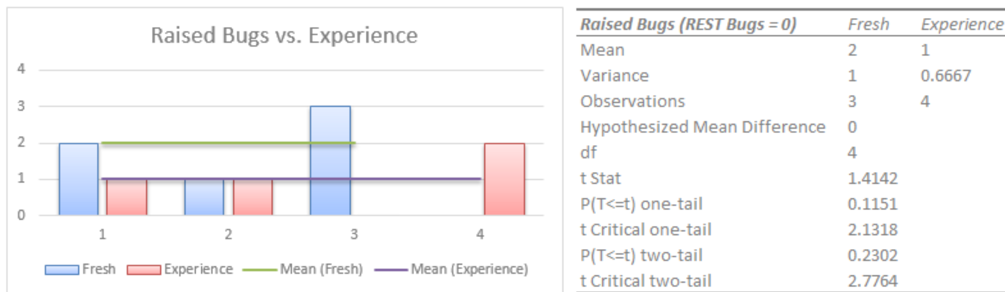
Variable	Description	Fresh	Experience
Participants	Count	3	4
	Experience Mean (Years)	0	6.75
Spent Time	Mean	64.67	37.75
	SD	14.29	9.32
Model Bug	Mean	2.00	1.00
	SD	1.00	0.82
REST Bug	Mean	0.00	0.00
	SD	0.00	0.00

has 3 remaining novice participants. It can be seen from Figure 7.5 that sub-groups G1.1 and G1.2 have high difference in average years of experience. From table 7.1, and although of huge difference between means of experience (0 against to 6.75 Years), it can be seen that both sub-groups almost took same time for development during the session. figure 7.7 and figure 7.8 demonstrate the low difference in spent time between expert and novice participants. When looking into mean values of spent time and model bugs, both variables seem to be a little tendency according to heavy development duties in software, which indicates that no significant difference in effort to learn and understand in terms of experience. There is no extreme large deviations, and somehow both are closed together. Actually, data confirms that no REST bugs found and no request for help to implement REST APIs either from experts or fresh engineers, absolutely.

While the presented descriptive statistics have introduced an excellent insight into data, both variables tends toward what can be expected from the hypothesis testing, as the following.

Hypothesis Testing

Both hypotheses were evaluated by using a t-test, MS Excel was used for the statistical computing [6]. For used approach vs. productivity, results of two-tailed for unpaired t-test show (p-value = $0.0309 < 0.05$), as shown in table 7.6. Therefore, we can conclude that the no significant difference hypothesis (H_0) is rejected, which means that there is a signif-



Raised Bugs (REST Bugs = 0)	*Fresh*	*Experience*
Mean	2	1
Variance	1	0.6667
Observations	3	4
Hypothesized Mean Difference	0	
df	4	
t Stat	1.4142	
P(T<=t) one-tail	0.1151	
t Critical one-tail	2.1318	
P(T<=t) two-tail	0.2302	
t Critical two-tail	2.7764	

Figure 7.8: (A) Raised model bugs in G1.1 vs. G2.1 and (B) Bugs difference calculations.

icant difference in productivity (with less spent time) for developers of first group who used the proposed approach. Since p-value is less than 5% (0.05), results of spent time are highly significant.

For spent time vs. experience, results of two-tailed for unpaired t-test confirm that (p-value = 0.0657 $>$ 0.05), as shown in table 7.7. And also for bugs vs. experience, t-test results illustrate that (p-value = 0.2302 $>$ 0.05), as shown in table 7.8. Based on both results, we can conclude that the significant difference hypothesis (H_0) is rejected, which means that there is no significant difference in the spent time and raised bugs in terms of experience in first group. Hence, new approach is easy to use either from experts or fresh graduate. Since p-value is more than 5% (0.05), results of spent time and also raised bugs are not significantly different.

7.3 Threats to Validity

In this section, experiment settings will be discussed to consider significant factors that can achieve the validity of results, and to avoid mistakes that may significantly reduce reliability of results, at least validity risk can be mitigated to the minimums. Based on the experiment's goal, several types of validity can be involved. According to the adopted empirical method, four types of validity are analyzed [70]: internal, external, construct, and conclusion validity.

In the internal validity, concern is focused on the actual study by avoiding matters that may influence the independent variables, and hence impact causal relationship between treatment and outcome, without attention from researchers. In this experiment, two threats are belong to

internal validity, whose are developers selection and used instrumentation. Developers were carefully involved to be familiar in database and web services development, and in the same time to be diverse in their experience. To avoid fatigue, each participant was only allocated to a single task in a single treatment, and most of them were strongly motivated through their employers to be serious enough, especially in ASAL Technologies [1]. While in the second group, competition spirit was the common attitude to confirm superiority of their own approaches.

For instrumentation, concern is to avoid impacts that may refer to used artifacts while executing the experiment. Therefore, used artifacts, such as tasks and learning documents, were designed to be short and comprehensive as much as adequate. Development environment was optimized over a VMware instance, where the VM encapsulated a similar preconfigured IDE for the first group, and also prepared a database dump for the second group, which avoids structural mistakes and keeps similar data for test procedure. Moreover, "HP CORE i7" Lab PCs were selected to be fast and convenient for most users, but using VMware impacted actual performance, which in terms increased the spent time in first group, and even so some developers suffered from inconvenience of IO peripherals for PCs. For Lab environment, it was selected and accommodated to be suitable as much as possible.

External validity concerns with results generalization to other environments rather than the relevant one of experiment. The most dangerous threat to external validity is to limit the involved developers in a certain sector, or with a specific background experience. And so, this threat is mitigated by selecting diverse developers, without and with background experience in a different fields and technologies, which can shape a comprehensive sample to generalize results for all other environments. On the other hand, experiment's assignment was comprehensively designed to be realistic and to represent most real-world scenarios.

Construct validity concerns to generalize results of experiment to the proposed approach (concept or theory) behind the experiment. The major threat against construct validity is the limits on the scope of environment, which makes the approach applicable for specific technologies only. And so, this threat is avoided by testing a portable and interoperable components during the experiment, where these components can independently run over different platforms with a full compatibility in a loosely coupled and dependency injection mode. For instance, dependency of the tested data-source aspect was injected to certain ORM,

while it can be inversion-ed to any other ORM, or may to object oriented database type, meanwhile tested ORM (Hibernate) can support all the common DB engines. Conceptually, the tested approach is designed to be applied over any different technology or platform.

Finally, conclusion validity concerns with results analysis from statistical perspective, which confirms the relationship between treatment and outcome to draw a correct conclusion. In this experiment, most common techniques are used, such as means, SD, and t-test, which are solid enough against violations of their hypothesis. the main threat in this level of validity is the low number of samples, which may impact the capability to reveal data patterns. Therefor, more samples were taken over different technologies to mitigate this threat, and to draw accurate trends. Meanwhile, participants were enrolled carefully, with precise follow up and supervision to ensure data quality.

Chapter 8

Conclusions

In this chapter, work results have been summarized and concluded. Goals were highlighted with a completed objectives, while difficulties and limitations were explored and addressed for future work.

8.1 Summary

In this research, study focused on the field area of REST API auto generation. In prior work, a 54 studies were the total of results that retrieved by the search process on different online libraries and databases, these results were analyzed and filtered through three major stages. The total final involved articles were 19 studies, these were mapped into the identified classification scheme of this study, and on basis of a predefined criteria to include or exclude prior studies.

The identified classification addressed studies based on the applied approach, which includes five categories: guidance, EMF based, model driven with code generation, RDF ontology model based, and OpenAPI model based approach. In each category, a brief discussion was done around the applied techniques or methods for each study from different perspectives, such as research approach, addressed problem, proposed contribution, hypothesis, employed methodology, and recommended future work.

In prior work, review shows most studies that required more verification by applying validation approach, which will confirm the scientific value for these studies to be considered in the real development environments. Secondly, according to the statistics, this field of research needs more

Table 8.1: Research approach facets.

Research Approach	Number of studies
Evaluation	10
Experience papers	4
Validation	5

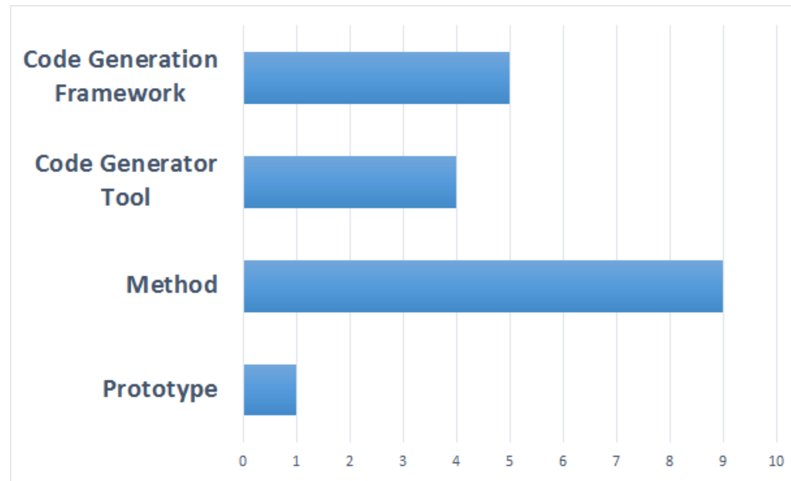


Figure 8.1: Contribution facets.

experience studies by experts. Moreover, most studies used the code generation approach, which cause major impacts on the development process as discussed, and the others imply manual work with a valid directions. Auto code generation was the most adopted approaches, which negatively impacts development process for software products, downsides refer to code rigidity, losing customized code, huge initial efforts, technical complexity, and restrictions for senior developers. Moreover, most studies still requires further empirical verification, where the evaluation of productivity and usability should be confirmed more enough, as shown in table 8.1 and in figures 8.1 and 8.2.

Proposed approach based over a framework that abstract layers for REST web APIs, business logic, data access, and model operations. Architecture is based over using general programming and aspect oriented programming AOP instead of using code generation, where REST APIs can be generalized for any correlated models without making repetitive code. While the dummy code is abstracted behind design patterns, the introduced approach is verified by a solid empirical proof of concept in

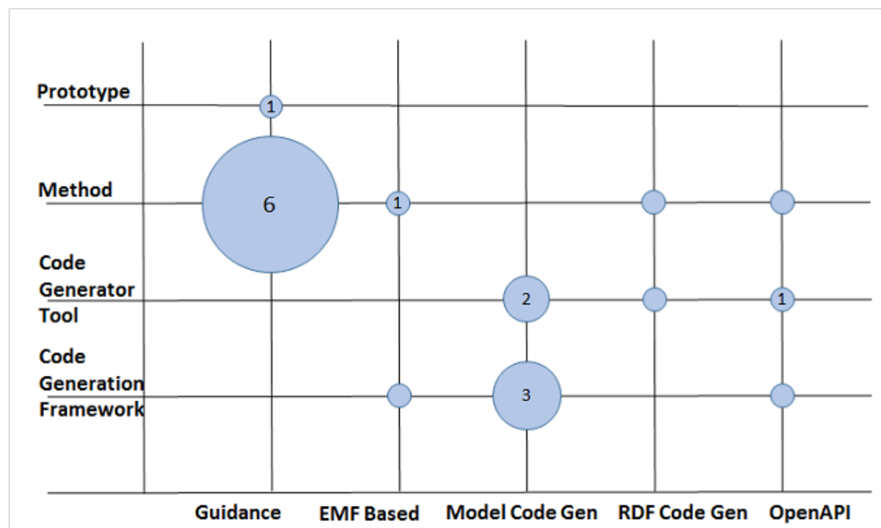


Figure 8.2: Approach categories among contribution facets.

an excrement and survey.

8.2 Goals achieved

Results shows a significant improvements on the productivity and easy to use, although of high prior experience in second group, average of spent time in the second group is more than twice of the first group, while the prior experience in first group is not a prerequisite. Furthermore, results demonstrate easy to learn in first group, where a little impact on the required time versus several years of experience.

In this study, improvements impact development life-cycle for REST web services, while the used approach enables software flexibility, agility without any restriction on developers margin, where research focus on applications Backbone to build backend as REST APIs. Research aims to reduce development time without impacting architecture flexibility and maintainability, reduce time to learn and to understand, and hence it simplifies the required human qualifications, and avoids downsides of auto code generators.

8.3 Recommendations

In this research, developed framework contains most development procedures of backbone into a centralized controllers, which enables developed services to plug and play inside framework. And so, system packages can be developed by several teams in a different places, which positively impacts compliance with Parallel Agile PA approach [14], where PA enables parallel development among teams over sprints, which may impact productivity.

From the perspective of application testing, it is recommended to research around the impact of centralism on building test cases and test automation. Moreover, future studies can investigate this centralism on security and permissions implementations on the applications level.

Maintainability is one of the most critical aspects in software development processes, where future research may study the impact of this approach on software changes and maintainability in real projects as a case studies. Furthermore, practitioners can study impact of developing a dedicated IDE plugin on the proposed approach, such as a dedicated Eclipse Developed Plugins.

On the other hand, it is worth to apply the same methodology over different platform, such .NET, Python, and PHP. Researchers can study impacts of the proposed approach over an other different platform, where impacts can also be compared with results in this research.

Appendix A

Tabular Information

In this appendix, all relevant tabular information will be included as a table with related introductions, so readers can refer to them as per their needs, information were categorized in sections.

A.1 Related work

Following Table A.1 includes all related articles before applying include – exclude criteria, as discussed in Section 3.2.4:

Table A.1: Related articles before filtration.

Draft Section	**Paper**	**CONF**	**Year**	**Relevant**	**Comments**
Automatic Code Generation	Automatic Code Generation with Business Logic	IEEE	2016	2	
	Design of Automatic Source Code Generation	Springer	2018	2	
REST-based Database	Developing a Prototype of REST-based Database	IEEE	2017	8	
Generation of RESTful APIs from Models	Generation of RESTful APIs from Models	ACM	2016	9	
Model-Driven Approach for REST	A model-driven approach for REST compliant services	IEEE	2014	8	
	A Model Driven Approach for the Development of Semantic RESTful	ACM	2013	8	

Continued on next page

Table A.1 – *Continued from previous page*

Draft Section	Paper	CONF	Year	Relevant	Comments
	Model-driven Code Generation for REST APIs - Master's Thesis	Stuttgart	2015	9	
	A Model-Driven Engineering Approach for RESTful Web Services	Springer	2017	6	
	A Model-Driven Tool for the Specification of REST Microservice	ICIST	2017	6	
	Model-Oriented Web Service Implementations Compared to Traditional Web Services	IEEE	2017	3	
	A data-model driven web application development framework	ACM	2014	2	
	Model-Oriented Web Services	IEEE	2016	3	
	Explication and semantic querying of enterprise information systems	Springer	2014	5	
	Example-Driven Web API Specification Discovery	Springer	2017	8	
	Extending OpenAPI 3.0 to Build Web Services from their Specification	WEBIST	2018	8	
	A Model-Driven Method for Fast Building Consistent Web Services from OpenAPI-Compatible Models	Springer	2018	6	First Online: 01 February 2019
	RESTful Web Services Development With a Model-Driven Engineering Approach	igi-global.com	2019	9	
RESTful using UML	Structural and Behavioral modeling of RESTful web service interface using UML	IEEE	2013	7	
Modularizing RESTful with AOP	Modularizing RESTful Web Service Management with Aspect Oriented Programming	IEEE	2015	5	
Rapid Realization of Executable Domain Models	Rapid Realization of Executable Domain Models via Automatic Code Generation	IEEE	2017	9	
	Automatic Code Generation Using Uml To Xml Schema Transformation	Journal	2014	2	
	The parallel agile process: Applying parallel processing techniques to software engineering	Weliy Journal	2018	2	
Linked-Data-Driven REST	Building Flexible and Reusable Semantic Web Interfaces	Springer	2016	4	
	Linked REST APIs: A Middleware for Semantic REST API Integration	IEEE	2017	8	

Continued on next page

Table A.1 – *Continued from previous page*

Draft Section	Paper	CONF	Year	Relevant	Comments
Data-as-a-Service	(Semi)automatic construction of access-controlled web data services	ACM	2018	9	Canada — October, IBM Corp. Riverton, NJ, USA ©2018
	ODaaS: Towards the Model-Driven Engineering of Open Data Applications as Data Services	IEEE	2014	8	
	Model-driven development of OData services: An application to relational databases	IEEE	2018	8	
	Semi-automatic Generation of Data-Intensive APIs	essi.upc.edu	2017	7	
	A Data-driven approach to improve the process of data-intensive API creation and evolution	CAiSE	2017	5	
	APIComposer: Data-Driven Composition of REST APIs	Springer	2018	6	
GraphQL	Generating GraphQL-Wrappers for REST(-like) APIs	Springer	2018	7	Sep, 2018, IBM Research in 2017

A.2 Implemented survey

below figure (Figure A.1) demonstrates the implemented survey to investigate the satisfaction of participants, and also to explore their opinions and expectations about the used framework.

A.3 Results of 1st experiment

below figure (Figure A.2) demonstrates collected results in first experiment.

A.4 Results of 2nd experiment

below figure (Figure A.2) demonstrates collected results in second experiment.

Questionnaire

On a scale between Strongly Agree (5) to Strongly Disagree (1), please rate the following statements:

	Questions	5	4	3	2	1
Training Material (Demo video)	1. The training session (such as demonstrative video and docs) was clear.					
	2. The information in the training video was helpful to complete the tasks and experiment.					
	3. I needed more help to complete the tasks during the experiment.					
Framework quality	4. The required exercise included real-world scenarios.					
	5. I am satisfied with how it was easy to use this framework to build REST API.					
	6. This framework minimized usual efforts to build REST APIs.					
	7. This framework reduced the time needed to create REST APIs.					
	8. Making changes to the generated REST APIs required little time.					
	9. I could fix problems easily and quickly in this framework.					
	10. I believe that proposed framework can improve productivity.					
	11. This framework has all the functions and capabilities that I expect to build REST APIs.					

Notes: ____

Figure A.1: Implemented survey.

Participant	Start Time	End Time	Spent Time	Progress (%)	Experience (Years)	Help Requests for Model	Help Requests for REST	REST Bugs	Model Bugs	Comments
From **ASAL** Technologies Staff	22/12/2019 02:53:00 PM	22/12/2019 04:10:00 PM	01:17:00	100%	0.2	3	1	0	2	He was very week in coding (Front-End Application Development Trainee).
From **ASAL** Technologies Staff	22/12/2019 04:31:00 PM	22/12/2019 05:20:00 PM	00:49:00	100%	0.2	2	1	0	1	She wasn't strong in coding, but she learns faster and thinks logically (Front-End Application Development Trainee).
From **ASAL** Technologies Staff	23/12/2019 11:52:00 AM	23/12/2019 01:00:00 PM	01:08:00	100%	0	4	0	0	3	He was very week in coding (Front-End Application Development Trainee).
From **Harri** Ramallah Staff	24/12/2019 08:00:00 PM	24/12/2019 08:30:00 PM	00:30:00	100%	10	4	0	0	1	He is very mature in java programming, and he is an expert in Oracle DB and Machine Learnining.
From **Jaffa.Net** Software Staff	29/12/2019 06:00:00 PM	29/12/2019 06:50:00 PM	00:50:00	100%	2	1	0	0	1	One year as QA engineer, and one year as Front-End Application Development engineer.
From **Al-Quds Open University ITC** Staff	29/12/2019 02:34:00 PM	29/12/2019 03:05:00 PM	00:31:00	100%	5	5	0	0	0	His experience in Struts Framework, he expects that exercises by Struts FW within 7 hours at least.
From **EXALT** Technologies Staff	01/03/2020 05:15:00 PM	01/03/2020 05:55:00 PM	00:40:00	100%	10	6	0	0	2	He has long experience in application back-end development.

Figure A.2: Results of 1st experiment.

Participant	Start Time	End Time	Spent Time	Progress (%)	Experience (Years)	Help Requests for Model	Help Requests for REST	Bugs #	Comments
From **ASAL** Technologies Staff	22/12/2019 04:27:00 PM	22/12/2019 05:31:00 PM	01:04:00	50%	2				He expected to finish with more one hour. Used Python language with auto generators tools.
From **ASAL** Technologies Staff	23/12/2019 12:01:00 PM	23/12/2019 12:50:00 PM	00:49:00	20%	9				Used MySQL server deployment time + Mysql database import, CherryPy package (Restful API server) deployment, SQLalchamy as an ORM library.
From **Freightos** Ramallah Staff	29/12/2019 08:30:00 PM	29/12/2019 09:23:00 PM	00:53:00	25%	1	1			Used Apache Camel. Whole code of model was given to help him.
From **EXALT** Technologies Staff	01/03/2020 05:20:00 PM	01/03/2020 06:00:00 PM	00:40:00	35%	10				Used Spring boot Framework.

Figure A.3: Results of 2nd experiment.

Appendix B

Code Snippets

In this appendix, all relevant code snippets will be included as a figures, so readers can refer to them as per their needs, snippets have been categorized in sections.

B.1 Model driven software development

Partial example (Figure B.1) of a RAML API specification file, as discussed in Section 2.3.7:

```
#%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"
```

Figure B.1: RAML API specification file Example.

OpenAPI 3.0, YAML standard example - 1 (Figure B.2), as discussed in Section 2.6:

```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
```


Figure B.2: YAML standard example – 1.

OpenAPI 3.0, YAML standard example - 2 (Figure B.3), as discussed in Section 2.6:


```yaml
    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"
```


Figure B.3: YAML standard example – 2.

B.2 Resource description framework

Description syntax example of RDF (Figure B.4), as discussed in Section 2.4:

```
<?xml:namespace ns = “http://www.w3.org/RDF/RDF/” prefix
  = “RDF” ?>
<?xml:namespace ns = “http://purl.oclc.org/DC/” prefix =
  “DC” ?>
<?xml:namespace ns = “http://person.org/BusinessCard/”
  prefix = “CARD” ?>
<RDF:RDF>
    <RDF:Description RDF:HREF = “http://uri-of-Document-1”>
        <DC:Creator RDF:HREF = “#Creator_001”/>
    </RDF:Description>
    <RDF:Description ID=“Creator_001”>
        <CARD:Name>John Smith</CARD:Name>
        <CARD:Email>smith@home.net</CARD:Email>
        <CARD:Affiliation>Home, Inc.</CARD:Affiliation>
    </RDF:Description>
</RDF:RDF>
```

Figure B.4: RDF description syntax [46].

Schema syntax example (Figure B.5) of RDF description, as discussed in Section 2.4:

```
<?xml:namespace ns = “http://www.w3.org/RDF/RDF/” prefix
  =“RDF” ?>
<?xml:namespace ns = “http://purl.oclc.org/DC/” prefix =
  “DC” ?><RDF:RDF>
    <RDF:Description RDF:HREF = “http://uri-of-Document-1”>
        <DC:Creator>John Smith</DC:Creator>
    </RDF:Description>
</RDF:RDF>
```

Figure B.5: RDF schema syntax [46].

Appendix C

Source Code

In this appendix, source code of training example and experiment assignment were included, so developers can refer to them for more clarification.

C.1 Training example

Training example was prepared to learn participants of experiment, as discussed in section 6.1.

C.1.1 Model

This section lists model source code for all related entities, as shown in the following figures: C.1, C.2, C.3, C.5, C.4, and C.6.

C.1.2 Services

This section lists source code of web services for all the related entities, as shown in the following figures: C.7 and C.8.

C.2 Experiment assignment

Used as assignment to execute for the experiment to measure time and bugs, as discussed in section 6.2.

```
1  package model;
2
3⊕ import java.io.Serializable;[]
10
11
12⊖ /**
13   * The persistent class for the city database table.
14   *
15   */
16 @Entity
17 @Table(name="city")
18 public class City implements Serializable, com.es.fw.ov.IEntity {
19     private static final long serialVersionUID = 1L;
20
21⊖    @Id
22     @Column(unique=true, nullable=false)
23     private int id;
24
25⊖    @Column(length=45)
26     private String name;
27
28     //bi-directional many-to-one association to Country
29⊖    @ManyToOne
30     @JsonBackReference
31     @JoinColumn(name="countryId")
32     private Country country;
33
34⊖    public City() {
35     }
36
37⊕    public Country getCountry() {[]
40
41⊕    public void setCountry(Country country) {[]
44
45⊖    public IEntity clone() {
46         return null;
47     }
48
49⊖    @Override
50     public Serializable PK() {
51         return this.getId();
```

Figure C.1: City entity model.

C.2.1 Model

This section lists model source code for all related entities, as shown in the following figures: C.1, C.2, C.9, C.10, C.11, and C.12.

```
package model;

import java.io.Serializable;

/**
 * The persistent class for the country database table.
 */
@Entity
@Table(name="country")
public class Country implements Serializable, com.es.fw.ov.IEntity {
    private static final long serialVersionUID = 1L;

    @Id
    @Column(unique=true, nullable=false)
    private int id;

    @Column(length=45)
    private String name;

    //bi-directional many-to-one association to City
    @JsonManagedReference
    @OneToMany(mappedBy="country", fetch = FetchType.EAGER)
    private List<City> cities;

    public Country() {
    }

    public IEntity clone() {
        return null;
    }

    @Override
    public Serializable PK() {
        return this.getId();
    }

    @Override
    public boolean equals(Object obj) {
        return this.PK().equals(((IEntity) obj).PK());
    }
```

Figure C.2: Country entity model.

C.2.2 Services

This section lists source code of web services for all the related entities, as shown in the following figures: C.13 and C.14.

```
1  package model;
2
3⊕ import java.io.Serializable;[.]
7
8⊖ /**
9   * The persistent class for the factory database table.
10  */
11 @Entity
12 @Table(name="factory")
13 public class Factory implements Serializable, com.es.fw.ov.IEntity {
14     private static final long serialVersionUID = 1L;
15
16⊖    @Id
17     @Column(unique=true, nullable=false)
18     private int id;
19
20⊖    @Column(length=45)
21     private String description;
22
23⊖    @Column(length=45)
24     private String region;
25
26⊖    @Column(length=45)
27     private String title;
28
29     //bi-directional many-to-one association to City
30⊖    @ManyToOne
31     @JoinColumn(name="cityId")
32     private City city;
33
34⊖    public Factory() {
35     }
36
37⊕    public City getCity() {[.]
40
41⊕    public void setCity(City city) {[.]
44
45⊕    public int getId() {[.]
48
49⊕    public void setId(int id) {[.]
```

Figure C.3: Factory entity model.

```
1  package model;
2
3⊕ import java.io.Serializable;
10
11⊖ /**
12  * The persistent class for the car database table.
13  */
14 @Entity
15 @Table(name="car")
16 @DiscriminatorValue("1")
17 @PrimaryKeyJoinColumn(name = "Id", referencedColumnName = "id")
18 public class Car extends Vehicle implements Serializable, com.es.fw.ov.IEntity {
19     private static final long serialVersionUID = 1L;
20
21⊖     @Column(length=45)
22     private String bodyStyle;
23
24⊖     @Column(length=45)
25     private String seats;
26
27     //bi-directional many-to-many association to Part
28⊖     @ManyToMany(cascade=CascadeType.MERGE, fetch = FetchType.EAGER)
29     @JoinTable(name="car_has_part", joinColumns={@JoinColumn(name="carId")}
30                                 , inverseJoinColumns={@JoinColumn(name="partId")})
31     private Set<Part> parts;
32
33⊖     public Car() {
34     }
35
36⊕     public String getBodyStyle() {
39
40⊕     public void setBodyStyle(String bodyStyle) {
43
44⊕     public String getSeats() {
47
48⊕     public void setSeats(String seats) {
51
52⊕     public Set<Part> getParts() {
55
56⊕     public void setParts(Set<Part> parts) {
```

Figure C.4: Car entity model.

```
1  package model;
2
3⊕ import java.io.Serializable;
11
12⊖ /**
13  * The persistent class for the part database table.
14  *
15  */
16 @Entity
17 @Table(name="part")
18 public class Part implements Serializable, com.es.fw.ov.IEntity {
19     private static final long serialVersionUID = 1L;
20
21⊖    @Id
22     @Column(unique=true, nullable=false)
23     private int id;
24
25     private float price;
26
27⊖    @Column(nullable=false, length=45)
28     private String serialNo;
29
30     //bi-directional many-to-many association to Car
31⊖    @JsonBackReference
32     @ManyToMany(mappedBy="parts", cascade = CascadeType.MERGE, fetch=FetchType.EAGER)
33     private Set<Car> cars;
34
35⊕    public Part() {
37
38⊖    public IEntity clone() {
39         return null;
40     }
41
42⊖    @Override
43     public Serializable PK() {
44         return this.getId();
45     }
46
47⊖    @Override
48     public boolean equals(Object obj) {
49         return this.PK().equals(((IEntity) obj).PK());
```

Figure C.5: Part entity model.

```
1  package model;
2
3⊕ import java.io.Serializable;
9
10⊖ /**
11  * The persistent class for the vehicle database table.
12  */
13 @Entity
14 @Table(name="vehicle")
15 @Inheritance( strategy = InheritanceType.JOINED)
16 @DiscriminatorColumn(
17         discriminatorType = DiscriminatorType.INTEGER,
18         name = "typeId",
19         columnDefinition = "INT")
20 public class Vehicle implements Serializable, com.es.fw.ov.IEntity {
21     private static final long serialVersionUID = 1L;
22
23⊖     @Id
24     @Column(unique=true, nullable=false)
25     private int id;
26
27⊖     @Column(length=45)
28     private String color;
29
30⊖     @Temporal(TemporalType.DATE)
31     @Column(nullable=true)
32     private Date dateOfMade;
33
34⊖     @Column(length=45)
35     private String description;
36
37⊖     @Column(length=45)
38     private String status;
39
40⊖     @Column(length=45)
41     private String vinNo;
42
43     //bi-directional one-to-one association to Factory
44⊖     @JoinColumn(name="id")
45     @OneToOne(cascade=CascadeType.ALL, fetch = FetchType.EAGER)
46     private Factory factory;
```

Figure C.6: Vehicle entity model.

```
1⊕ /**□
4  package service;
5
6⊕ import javax.servlet.http.HttpServletRequest;□
18
19⊖ /**
20  * @author Salah.Hussein
21  *
22  */
23 @Controller
24 @RequestMapping("/car")
25 public class CarAPI extends SpringWebAPI<Car> {
26
27⊖     @Override
28     public Car newEntity() {
29         return new Car();//super.
30     }
31
32⊖     @RequestMapping(value = "/test" , method = RequestMethod.GET, produces = MediaType.TEXT_PLAIN_VALUE)
33     public @ResponseBody String test() {
34
35         return "Good Test.";
36     }
37
38⊖     @RequestMapping(value = "/empty" , method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
39     public @ResponseBody Car empty() {
40         return new Car();
41     }
42
43⊖     @RequestMapping(value = "/test1" , method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
44     public @ResponseBody Car test(HttpServletRequest req) {
45         IWorkFlow<Car> workFlow = getEntityWF(req);
46
47         return workFlow.select("Car").get(0);
48     }
49 }
50
```

Figure C.7: Car web APIs.

```
1⊕ /**□
4  package service;
5
6⊕ import javax.servlet.http.HttpServletRequest;□
18
19⊖ /**
20  * @author Salah.Hussein
21  *
22  */
23 @Controller
24 @RequestMapping("/part")
25 public class PartAPIs extends SpringWebAPI<Part> {
26
27⊖     @Override
28     public Part newEntity() {
29         return null;
30     }
31
32⊖     @RequestMapping(value = "/test1" , method = RequestMethod.GET, produces = MediaType.APPLICATION_JSON_VALUE)
33     public @ResponseBody Part test(HttpServletRequest req) {
34         IWorkFlow<Part> workFlow = getEntityWF(req);
35
36         return workFlow.select("Part").get(0);
37     }
38 }
39
```

Figure C.8: Part web APIs.

```
1  package model;
2
3⊕ import java.io.Serializable;[..]
9
10⊖ /**
11  * The persistent class for the address database table.
12  */
13 @Entity
14 public class Address implements Serializable, IEntity {
15     private static final long serialVersionUID = 1L;
16
17⊖     @Id
18     private int id;
19
20     private String region;
21
22     private String streetName;
23
24     private String telephone;
25
26     private String title;
27
28     //bi-directional many-to-one association to City
29⊖     @ManyToOne
30     @JoinColumn(name="cityId")
31     private City city;
32
33⊖     public Address() {
34     }
35
36⊕     public int getId() {[..]
39
40⊕     public void setId(int id) {[..]
43
44⊕     public City getCity() {[..]
47
48⊕     public void setCity(City city) {[..]
51
52⊕     public String getRegion() {[..]
```

Figure C.9: Address entity model.

```
1  package model;
2
3⊕ import java.io.Serializable;[]
10
11⊖ /**
12  * The persistent class for the student database table.
13  */
14 @Entity
15 @Table(name="Student")
16 @PrimaryKeyJoinColumn(name = "Id")
17 @DiscriminatorValue("1")
18 public class Student extends Person implements Serializable, IEntity {
19     private static final long serialVersionUID = 1L;
20
21⊖    public Student() {
22         setTypeId(1);
23     }
24
25     //bi-directional many-to-many association to Parent
26⊖    @JsonManagedReference
27     @ManyToMany(cascade=CascadeType.ALL, fetch = FetchType.EAGER)
28     @JoinTable(name="parent_has_student", joinColumns={@JoinColumn(name="studentId")}
29                                      , inverseJoinColumns={@JoinColumn(name="parentId")})
30     private List<model.Parent> parents;
31
32⊕    public List<model.Parent> getParents() {[]
35
36⊕    public void setParents(List<Parent> parents) {[]
39
40⊕    public Parent addParent(Parent parent) {[]
45
46⊕    public Parent removeParent(Parent parent) {[]
51
52⊖    public IEntity clone() {
53         return null;
54     }
55
56⊖    @Override
57     public Serializable PK() {
58         return this.getId();
59     }
```

Figure C.10: Student entity model.

```
1   package model;
2
3⊕  import java.io.Serializable;
11
12⊖ /**
13   * The persistent class for the parent database table.
14   */
15  @Entity
16  @Table(name="Parent")
17  @PrimaryKeyJoinColumn(name = "Id")
18  @DiscriminatorValue("2")
19  public class Parent extends Person implements Serializable, IEntity {
20      private static final long serialVersionUID = 1L;
21      private String mobileNumber;
22      private String name;
23      private String phoneNumber;
24
25⊕     public Parent() {
28
29⊕     public IEntity clone() {
32
34⊕     public Serializable PK() {
37
38⊕     public String getMobileNumber() {
41
42⊕     public void setMobileNumber(String mobileNumber) {
45
46⊕     public String getName() {
49
50⊕     public void setName(String name) {
53
54⊕     public String getPhoneNumber() {
57
58⊕     public void setPhoneNumber(String phoneNumber) {
61
62⊖     @JsonBackReference
63      @ManyToMany(mappedBy="parents", fetch = FetchType.EAGER)
64      private Set<Student> students;
65
66⊕     public Set<model.Student> getStudents() {
```

Figure C.11: Parent entity model.

```
 1 package model;
 2
 3⊕import java.io.Serializable;[.]
12
13⊖/**
14  * The persistent class for the person database table.
15  *
16  */
17
18 @Entity
19 @Table(name = "Person")
20 @Inheritance( strategy = InheritanceType.JOINED)
21 @DiscriminatorColumn(
22         discriminatorType = DiscriminatorType.INTEGER,
23         name = "typeId",
24         columnDefinition = "INT")
25 public class Person implements Serializable, IEntity {
26     private static final long serialVersionUID = 1L;
27
28⊖    @Id
29     private int id;
30     private int typeId;
31     private String arabicName;
32⊖    @Temporal(TemporalType.DATE)
33     private Date dateOfBirth;
34     private String email;
35     private String englishName;
36     private String gender;
37     private String identityNum;
38     private String nationality;
39     private String placeOfBirth;
40     private String status;
41
42     //bi-directional many-to-one association to Address
43⊖    @ManyToOne
44     @JoinColumn(name="addressId")
45     private Address address;
46
47⊕    public Person() {[.]
49
50⊕    public int getId() {[.]
```

Figure C.12: Person entity model.

```
1  package com.es.srm.service.student;
2
3⊕ import java.util.Iterator;
28 @Controller
29 @RequestMapping("/student")
30 public class StudentAPI extends SpringWebAPI<Student> {
31
32⊖     @Override
33     public Student newEntity() {
34         // TODO Auto-generated method stub
35         return new Student();
36     }
37 }
38
```

Figure C.13: Student web APIs.

```
1⊕ /**
4  package com.es.srm.service.parent;
5
6⊕ import javax.servlet.http.HttpServletRequest;
19
20 @Controller
21 @RequestMapping("/parent")
22
23 public class ParentAPI extends SpringWebAPI<Parent> {
24
25⊖     @Override
26     public Parent newEntity() {
27         return new Parent();
28     }
29
30 }
```

Figure C.14: Parent web APIs.

Bibliography

[1] Asal technologies. https://www.asaltech.com/. (Accessed on 01/06/2020).

[2] Eclipse data tools platform (dtp) project — the eclipse foundation. https://www.eclipse.org/datatools/. (Accessed on 12/31/2019).

[3] Eclipse tools for spring - dzone - refcardz. https://dzone.com/refcardz/eclipse-tools-spring?chapter=1. (Accessed on 12/31/2019).

[4] Hibernate getting started guide. https://docs.jboss.org/hibernate/orm/5.4/quickstart/html_single/#preface. (Accessed on 12/31/2019).

[5] Home. http://www.es-pal.com/es/. (Accessed on 01/06/2020).

[6] How to calculate p value in excel (step-by-step tutorial). https://spreadsheeto.com/p-value-excel/. (Accessed on 01/10/2020).

[7] Intro to the jackson objectmapper — baeldung. https://www.baeldung.com/jackson-object-mapper-tutorial. (Accessed on 12/31/2019).

[8] Jackson objectmapper. http://tutorials.jenkov.com/java-json/jackson-objectmapper.html. (Accessed on 12/31/2019).

[9] Jboss tools - hibernate tools. https://tools.jboss.org/features/hibernate.html. (Accessed on 01/01/2020).

[10] Likert scale questions: Definitions, examples + how to use them — typeform. https://www.typeform.com/surveys/likert-scale-questionnaires/. (Accessed on 01/04/2020).

[11] Maven – introduction. https://maven.apache.org/what-is-maven.html. (Accessed on 12/31/2019).

[12] Rest apis automatic generation - youtube. https://www.youtube.com/watch?fbclid=IwAR132OaqQbjIYgJUDuY9ZXjn7e-EybycvqYpb65hBxIU0gqmNx_nyLxDjJI&edufilter=NULL&feature=youtu.be&v=Fdq15c3T4vk. (Accessed on 01/04/2020).

[13] Djamel Amar Bensaber and Mimoun Malki. Development of semantic web services: model driven approach. In *Proceedings of the 8th international conference on New technologies in distributed systems*, page 40. ACM, 2008.

[14] Ali Asfour, Samer Zain, Norsaremah Salleh, and John Grundy. Exploring agile mobile app development in industrial contexts: A qualitative study. *International Journal of Technology in Education and Science*, 3(1):29–46, 2019.

[15] John Bailey, David Budgen, Mark Turner, Barbara Kitchenham, Pearl Brereton, and Stephen Linkman. Evidence relating to object-oriented software design: A survey. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 482–484. IEEE, 2007.

[16] Bernd Bruegge and Allen H Dutoit. Object–oriented software engineering. using uml, patterns, and java. *Learning*, 5(6):7, 2009.

[17] Frank Budinsky, David Steinberg, Raymond Ellersick, Timothy J Grose, and Ed Merks. *Eclipse modeling framework: a developer's guide*. Addison-Wesley Professional, 2004.

[18] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on software engineering*, 20(6):476–493, 1994.

[19] Dolors Costal, Carles Farré, Cristina Gómez, Petar Jovanovic, Oscar Romero, and Jovan Varga. Semi-automatic generation of data-intensive apis, 2017.

[20] Rafael Corveira da Cruz Gonçalves and Isabel Azevedo. *RESTful Web Services Development With a Model-Driven Engineering Approach*, pages 191–228. IGI Global, 2019.

[21] Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Apicomposer: Data-driven composition of rest apis. In *European Conference on Service-Oriented and Cloud Computing*, pages 161–169. Springer.

[22] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Example-driven web api specification discovery. In *European Conference on Modelling Foundations and Applications*, pages 267–284. Springer.

[23] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Model-driven development of odata services: An application to relational databases. In *2018 12th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–12. IEEE.

[24] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. Emf-rest: generation of restful apis from models. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1446–1453. ACM.

[25] Kalvin Eng, Diego Serrano, Eleni Stroulia, and Jacob Jaremko. (semi) automatic construction of access-controlled web data services. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, pages 72–80. IBM Corp.

[26] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1, 1999.

[27] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[28] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.

[29] Markus Fischer. *Model-driven code generation for REST APIs*. Thesis, 2015.

[30] Ned Freed and Nathaniel Borenstein. Multipurpose internet mail extensions (mime) part two: Media types. Technical report, rfc 2046, November, 1996.

[31] Hilary Glasman-Deal. *Science research writing for non-native speakers of English*. World Scientific, 2010.

[32] Chenkai Guo, Jing Xu, Hongji Yang, Ying Zeng, and Shuang Xing. An automated testing approach for inter-application security in android. In *Proceedings of the 9th international workshop on automation of software test*, pages 8–14. ACM, 2014.

[33] Florian Haupt, Dimka Karastoyanova, Frank Leymann, and Benjamin Schroth. A model-driven approach for rest compliant services. In *2014 IEEE International Conference on Web Services*, pages 129–136. IEEE.

[34] Florian Haupt, Frank Leymann, Anton Scherer, and Karolina Vukojevic-Haupt. A framework for the structural analysis of rest apis. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 55–58. IEEE, 2017.

[35] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd international conference on software engineering*, pages 471–480. ACM, 2011.

[36] Jeong-cheol Jeon and Jaehwa Chung. Developing a prototype of rest-based database application for shipbuilding industry: A case study. In *2017 International Conference on Platform Technology and Service (PlatCon)*, pages 1–6. IEEE.

[37] Jill K Jesson and Fiona Lacey. How to do (or not to do) a critical literature review. 2006.

[38] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Alef Arendsen, Darren Davison, Dmitriy Kopylenko, Mark Pollack, et al. The spring framework–reference documentation. *interface*, 21:27, 2004.

[39] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. 2007.

[40] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[41] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[42] Zhifang Liu, Xiaopeng Gao, and Xiang Long. Adaptive random testing of mobile application. In *2010 2nd International Conference on Computer Engineering and Technology*, volume 2, pages V2–297. IEEE, 2010.

[43] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. Investigating web apis on the world wide web. In *2010 Eighth IEEE European Conference on Web Services*, pages 107–114. IEEE, 2010.

[44] Robert C Martin. The principles of ood. url: http://butunclebob.com/articles. unclebob. *PrinciplesOfOod (visited on 01/09/2017)*, 2003.

[45] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri): Generic syntax. 2005.

[46] Eric J Miller. An introduction to the resource description framework. *Journal of library administration*, 34(3-4):245–255, 2001.

[47] Shahid Mujtaba, Kai Petersen, Robert Feldt, and Michael Mattsson. Software product line variability: A systematic mapping study. *School of Engineering, Blekinge Inst. of Technology*, 2008.

[48] Shahid Mujtaba, Kai Petersen, Robert Feldt, and Michael Mattsson. Software product line variability: A systematic mapping study. *School of Engineering, Blekinge Inst. of Technology*, 2008.

[49] Gary W Oehlert. *A first course in design and analysis of experiments*. 2010.

[50] Elizabeth J O'Neil. Object/relational mapping 2008: hibernate and the entity data model (edm). In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1351–1356. ACM, 2008.

[51] Noel Pérez et al. Research methodology: An example in a real project. *Retrieved on April*, 10:2014, 2009.

[52] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Odata version 4.0 part 1: protocol. *OASIS Standard. Available online: http://docs. oasis-open. org/odata/odata/v4. 0/os/part1-protocol/odata-v4. 0-os-part1-protocol. pdf (accessed on 10 December 2017)*, 2014.

[53] Michael Pizzo, Ralf Handl, and Martin Zurmuehl. Odata version 4.0 part 2: Url conventions. *OASIS, Tech. Rep.*, 2014.

[54] ProgrammableWeb.com. Apis show faster growth rate in 2019 than previous years, 2019. https://www.programmableweb.com/news/research-shows-interest-providing-apis-still-high/research/2018/02/23, Last accessed on June 2019.

[55] Xhevi Qafmolla and Viet Cuong Nguyen. Automation of web services development using model driven techniques. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 3, pages 190–194. IEEE, 2010.

[56] Alek Radjenovic and Richard F Paige. Behavioural interoperability to support model-driven systems integration. In *Proceedings of the First International Workshop on Model-Driven Interoperability*, pages 98–107. ACM, 2010.

[57] Digvijaysinh M Rathod, Satyen M Parikh, and BV Buddhadev. Structural and behavioral modeling of restful web service interface using uml. In *2013 International conference on intelligent systems and signal processing (ISSP)*, pages 28–33. IEEE.

[58] Dominik Renzel, Patrick Schlebusch, and Ralf Klamma. Today's top "restful" services and why they are not restful. In *International Conference on Web Information Systems Engineering*, pages 354–367. Springer, 2012.

[59] Diaeddin Rimawi and Samer Zein. A model based approach for android design patterns detection. In *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 1–10. IEEE, 2019.

[60] Guy M Robinson. *Methods and techniques in human geography*. John Wiley & Son Ltd, 1998.

[61] Ángel Mora Segura, Jesús Sánchez Cuadrado, and Juan de Lara. Odaas: Towards the model-driven engineering of open data applications as data services. In *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations*, pages 335–339. IEEE.

[62] Diego Serrano, Eleni Stroulia, Diana Lau, and Tinny Ng. Linked rest apis: a middleware for semantic rest api integration. In *2017*

IEEE International Conference on Web Services (ICWS), pages 138–145. IEEE.

[63] David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, and Arnaud Lanoix. Extending openapi 3.0 to build web services from their specification. In *International Conference on Web Information Systems and Technologies*.

[64] David Sferruzza, Jérôme Rocheteau, Christian Attiogbé, and Arnaud Lanoix. A model-driven method for fast building consistent web services from openapi-compatible models. In *International Conference on Model-Driven Engineering and Software Development*, pages 9–33. Springer.

[65] Harmeet Singh and Syed Imtiyaz Hassan. Effect of solid design principles on quality of software: An empirical assessment. *International Journal of Scientific and Engineering Research*, 6(4), 2015.

[66] Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras. Model driven engineering with ontology technologies. In *Reasoning Web International Summer School*, pages 62–98. Springer, 2010.

[67] Nírondes AC Tavares and Samyr Vale. A model driven approach for the development of semantic restful web services. In *Proceedings of International Conference on Information Integration and Web-based Applications and Services*, page 290. ACM.

[68] Branko Terzić, Vladimir Dimitrieski, Slavica Kordić, Gordana Milosavljević, and Ivan Luković. Development and evaluation of microbuilder: a model-driven tool for the specification of rest microservice software architectures. *Enterprise Information Systems*, 12(8-9):1034–1057, 2018.

[69] Bo Wang, Doug Rosenberg, and Barry W Boehm. Rapid realization of executable domain models via automatic code generation. In *2017 IEEE 28th Annual Software Technology Conference (STC)*, pages 1–6. IEEE.

[70] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[71] Christoforos Zolotas, Themistoklis Diamantopoulos, Kyriakos C Chatzidimitriou, and Andreas L Symeonidis. From requirements to source code: a model-driven engineering approach for restful web services. *Automated Software Engineering*, 24(4):791–838, 2017.