University of Economics, Prague

Faculty of Informatics and Statistics

GRAPHQL API TEST AUTOMATION

MASTER THESIS

Study programme: Applied Informatics

Field of study: Information Technologies

Author: Bc. Alexandra Kolpakova

Supervisor: doc. Ing. Alena Buchalcevová, Ph.D.

Prague, December 2019

Declaration

I hereby declare that I am the sole author of the thesis entitled “GraphQL API test automation“. I duly marked out all quotations. The used literature and sources are stated in the attached list of references.

In Prague on .....

Bc. Alexandra Kolpakova

Acknowledgement

I hereby wish to express my appreciation and gratitude to the supervisor of my master's thesis doc. Ing. Alena Buchalcevová, Ph.D., for professional management, providing valuable advices and comments during the thesis processing.

I would also like to thank my family and friends who have always supported and encouraged me throughout work preparation.

Abstract

Master thesis work is specializing on problematics of implementation of GraphQL Application Programming Interface (API) test automation. The main goal is to compare existing GraphQL API test automation solutions to recommend one that can be used in a real project in a company that experienced a transition from Representational State Transfer (REST) to GraphQL API implementation and is searching for a new test automation approach.

The first part of this paper focuses on explaining the important terms. The API role in client-server Web applications architecture style is defined with the HTTP characteristics that enables the communication between the client-side and the server-side. Further chapters give an overview on the REST architecture style and GraphQL technology for API implementation. The next part describes the API testing process, with its main testing activities and challenges.

In the following parts of this thesis, the author introduces GraphQL API test automation solutions compatible with the Java programming language. These are then being compared, using the Multiple Criteria Decision Analysis, and the most beneficial solution is recommended to the concrete company that experienced the transition from REST to GraphQL API implementation.

Keywords

GraphQL API, API testing, GraphQL test automation, REST-assured

JEL Classification

L150, L860

Content

Introduction 12
Goals 13
Target group 13
Research methods 14
Assumptions and limitations 14
Expected benefits 15
Work structure 15
1 Literature review 17
1.1 Professional publications 17
1.2 Academic works 18
1.3 Internet resources 19
2 Application Programming Interface 20
2.1 Client-server architecture 20
2.1.1 HTTP communication 24
2.2 REST architecture style 25
2.2.1 Principles and constraints 26
2.2.2 Limitations 27
3 GraphQL 28
3.1 History of GraphQL 29
3.2 Features 31
3.2.1 GraphQL server description 32
3.2.2 Type system and schemas 32
3.2.3 Query Language 33
4 API testing 36
4.1 Characteristics of API testing 36
4.1.1 Testing activities 38
4.1.2 Challenges 41
4.2 API test automation 42
5 GraphQL API test automation using Java 45
5.1 Test automation solutions research 45
5.2 Solution overview 46
5.2.1 Test GraphQL Java 46
5.2.2 REST-assured 47

5.2.3 Karate 49
6 Evaluation of GraphQL API solutions for concrete company 51
6.1 Context definition 51
6.1.1 Company overview 52
6.1.2 GraphQL API 54
6.1.3 Requirements for API testing 55
6.2 Alternatives introduction 57
6.3 Criteria definition 57
6.3.1 Criteria evaluation values 61
6.4 Weights specification 62
6.5 Measuring the alternatives 63
6.5.1 Test GraphQL Java 63
6.5.2 REST-assured 71
6.6 Alternatives evaluation 79
Conclusions 83
List of references 85
Annexes 91
Annex A: GraphQL custom server implementation 91
Annex B: GraphQL requests used in solutions evaluation 92
Annex C: CRUD Test Cases 94
Annex D: MCDA questionnaires 96
Annex E: CRUD test suite implementation 100

List of Figures

Figure 1 Client-Server network model (source: https://techdifferences.com/difference-between-client-server-and-peer-to-peer-network.html) 21
Figure 2 API in client-server architecture of Web application (source: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/) 22
Figure 3 Client adapter code model implemented by Netflix (source: https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d) 28
Figure 4 Google Trends of searching "GraphQL" topic in past 5 years worldwide (source: https://trends.google.com/) 31
Figure 5 Custom GraphQL server schema part (source: author) 33
Figure 6 Example of query validation (source: author) 33
Figure 7 Example of query variable usage (source: author) 35
Figure 8 Example of similar data shapes in multiple queries (source: author) 35
Figure 9 Example of using fragment in GraphQL queries (source: author) 35
Figure 10 Types of testing in client-server architecture style for Web applications (source: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/, modified by: author) 36
Figure 11 V-model in software testing (source: https://www.testbytes.net/blog/v-model-and-w-model-software-testing/) 37
Figure 12 Mike Cohn's Test Automation Pyramid (source: https://www.360logica.com/blog/sneak-peek-test-framework-test-pyramid-testing-pyramid/) 43
Figure 13 Sending request method in Test GraphQL Java example Test Class (source: https://github.com/vimalrajselvam/test-graphql-java) 47
Figure 14 Test GraphQL Java library methods (source: https://github.com/vimalrajselvam/test-graphql-java) 47
Figure 15 REST-assured test example (source: https://www.baeldung.com/rest-assured-response) 48
Figure 16 Basic user authentication example code with REST-assured (source: https://www.baeldung.com/rest-assured-authentication) 48
Figure 17 Karate test example (source: https://www.baeldung.com/karate-rest-api-testing) 50
Figure 18 Karate GraphQL test example (source: https://github.com/intuit/karate) 50
Figure 19 Dependency installed in Test GraphQL Java library example project (source: author) 64
Figure 20 The ease of understanding solution A's methods - questionnaire results (source: author) 68
Figure 21 CRUD suite test run time using Test GraphQL Java library (source: author) 69
Figure 22 Test GraphQL Java error message example (source: author) 70
Figure 23 YouTube search results using keyword "rest-assured" (source: https://www.youtube.com/results?search_query=rest-assued) 72

Figure 24 Installed dependencies in Initial project setup with REST-assured (source: author) 72
Figure 25 Extended installed dependencies in Initial project setup with REST-assured (source: author) 73
Figure 26 The ease of understanding solution B methods - questionnaire results (source: author) 76
Figure 27 CRUD suite test run time using REST-assured (source: author) 77
Figure 28 REST-assured error message example (source: author) 78
Figure 29 Normalised criterion matrix formula (source: [89]) 80
Figure 30 Formula for calculating alternatives performance value (source: [89]) 80
Figure 31 The ease of understanding solution methods questionnaire template (source: author) 97
Figure 32 Error messages and logs informational content questionnaire template (source: author) 98
Figure 33 Criteria for choosing GraphQL API test automation solution (source: author) 99
Figure 34 Error messages and logs informational content questionnaire result (source: author) 100

List of tables

Table 1 Description of HTTP/1.1 methods (source: [34], [37]) 25
Table 2 HTTP/1.1 response status codes (source: [34]) 25
Table 3 GraphQL server operations for Data object (source: author) 32
Table 4 HTTP methods used in REST architectural style with custom GraphQL server operations (source: author) 34
Table 5 ISO 9126-1 software quality characteristics relation to API testing activities (source: [69], [70], [71]) 39
Table 6 Test scenarios templates for API functionality verification (source: author) 40
Table 7 API test automation challenges (source: [78]) 44
Table 8 GraphQL test automation approaches using Java search results evaluation (source: [79]) 46
Table 9 List of requirements for the integration testing approach from company X (source: author) 56
Table 10 List of criteria for the MCDA method (source: author) 58
Table 11 Get GraphQL structure Test Case (source: author) 60
Table 12 Criteria evaluation results with weight calculated (source: author) 63
Table 13 Calculating of average test suite run time for Test GraphQL Java (source: author) 69
Table 14 Calculating of average test suite run time for REST-assured (source: author) 77
Table 15 Alternatives evaluation criteria and weights (source: author) 80
Table 16 Minimising to maximising criteria transformation example (source: author) 80
Table 17 Application of transformation formula for normalising criterion matrix (source: author) 80
Table 18 Criteria values evaluation (source: author) 81
Table 19 CRUD suite Test Cases (source: author) 96

List of program´s codes

Code 1 "Data" object example (source: author)..... 32
Code 2 Request payload body for GraphQL API (source: author)..... 34
Code 3 Google search query for GraphQL test automation approaches using Java (source: author)..... 45
Code 4 BaseTest class implementation with Test GraphQL Java library (source: author) 65
Code 5 GetServerSchema Test Case implementation with Test GraphQL Java library (source: author)..... 66
Code 6 OkHttp request builder example using authorization header (source: author) 66
Code 7 Edit existing Data test implementation with Test GraphQL Java library (source: author)..... 67
Code 8 Invalid request example (source: author)..... 70
Code 9 BaseTest class implementation with REST-assured (source: author)..... 73
Code 10 GetServerSchema test case implementation with REST-assured (source: author) 74
Code 11 REST-assured request builder example using authorization header (source: author) 74
Code 12 Edit existing Data test implementation with REST-assured (source: author)..... 75
Code 13 REST-assured request with enabled logging (source: author)..... 78
Code 14 GraphQL server schema (source: author) 91
Code 15 Data object class (source: author)..... 92
Code 16 Query and mutation implementation example (source: author) 92
Code 17 "getServerSchema" query (source: author) 93
Code 18 "getListOfData" query (source: author)..... 93
Code 19 "getData" query (source: author)..... 93
Code 20 "createData" mutation (source: author)..... 94
Code 21 "dataTypeFragment" fragment (source: author)..... 94
Code 22 "editData" mutation (source: author) 94
Code 23 "deleteData" mutation (source: author)..... 94
Code 24 CRUDSuite implementation using Test GraphQL Java library (source: author)103
Code 25 CRUDSuite implementation using REST-assured library (source: author) 105

List of abbreviations

API	Application Programming Interface
TC	Test Case
GraphQL	Graph Query Language
REST	Representational state transfer
QA	Quality Assurance
FIS	Faculty of Informatics and Statistics
MCDA	Multiple Criteria Decision Analysis
HTTP	Hypertext Transfer Protocol
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
CSS	Cascading Style Sheets
SOAP	Simple Object Access Protocol
RPC	Remote Procedure Call
XML	Extensible Markup Language
JSON	JavaScript Object Notation
JPEG	Joint Photographic Experts Group
MIME	Multipurpose Internet Mail Extensions
URL	Unified Resource Locators
UNI	Uniform Resource Identifiers
WSDL	Web Service Definition Language
OSFA	One-Size-Fits-All
RAD	Rapid Application Development
BDD	Behaviour Driven Development
WSA	Weighted Sum Approach

Introduction

Since World Wide Web (Web) technology was developed in 1990 as a system that runs on the Internet infrastructure, its use and role in people's everyday lives have changed increasingly [1]. Today, the Internet has become a necessity not only to individual lives but also in economic and political areas [2]. The growth of the Web usage influenced the popularity of e-commerce, this allows customers to purchase products and services through the Internet [3], making it more and more difficult for companies to grab their target audience's attention.

For a company to succeed in their field, it is important not only to understand the consumer's needs for a product or a service but also the expectations they have for Internet applications where they get the first impressions on the product. This leads to having higher requirements from Web applications and putting more emphasis on them while verifying if the final solution is according to the user acceptance criteria in terms of quality.

Common Web applications are designed according to the client-server architecture that allows the Internet user to communicate with Web application server-side through a Web page. This communication is provided by the Application Programming Interface (API) that sends information from the client to the server-side and back. All mentioned elements of Web applications are making the testing process more challenging as the system contains many dependent elements that cannot be controlled [4].

The process of testing Web applications can be handled in many ways, from testing the architecture components separately to testing their integration by handling API testing. API testing requires more technical knowledge and systems integration understanding to cover all possible Test Cases (TC) and prevent system failure. API tests belong to the integration tests category, in which the execution process is being highly automated to maintain a constant cost and support the continuous improvement of the Web application development process [5]. The test automation process is supported using tools, frameworks, and libraries that allow to set up the automated process of test activities such as test execution or result checking [6]. Some of the existing solutions for the API test automation suit the needs of different API implementation approaches, others are specific only to certain API types.

This leads to the problem that not all tools are compatible with the new technologies of API implementation like Graph Query Language (GraphQL). GraphQL is a rather new technology that was introduced in 2015 and not all approaches suited for most APIs support GraphQL and its specific features. Some of them start with the implementation and provision of trial versions for users. Moreover, the open-source solutions for GraphQL API testing that can be found on the Internet are mainly used by developers for writing unit tests but not for integration tests. The automation process for GraphQL API testing is becoming longer as the testing team should provide the proof-of-concept for the existing solutions to evaluate their usability on the project and find the solution that will suit their needs. The

longer process of test automation implementation has an impact on companies developing their solutions using new technologies.

These problems served as a reason for this paper to provide the reader with information about the existing approaches to GraphQL API test automation. As GraphQL technology supports multiple programming languages, this leads to the existence of variety of testing approaches for each language. Because of the author's interest and previous experience with test automation using Java, in this thesis, only GraphQL API test automation solutions implemented using Java are described. In the beginning, the author defines the API role in the Web application architecture, describes the GraphQL technology and specifies the testing approaches for the API quality verification. Then the author introduces multiple solutions that can be used for the API automation testing and, using the example of a real company that has implemented GraphQL API, evaluates the application of the existing solutions on the project based on a list of given requirements.

Goals

The main goal of this thesis is to compare the existing GraphQL API test automation solutions to recommend the one that can be used on a real project in a company that experienced a transition from Representational State Transfer (REST) to GraphQL API implementation and is searching for a new test automation approach.

The main goal is divided into the following sub-goals:

- Define the role of the Application Programming Interface in Web applications
- Introduce the GraphQL technology
- Characterize the process of API testing
- Describe the existing solutions for the GraphQL API test automation using Java
- Evaluate the application of the existing solutions on a real-life project

Target group

First and foremost, this work is aimed at Quality Assurance (QA) engineers and testers specialized on integration testing, especially those working for a company such as the one described in the practical part of the work, as this thesis describes the existing solutions that can be used for testing GraphQL API. Moreover, the methods and principles introduced can be adjusted to the already existing projects, independently of the API implementation approach.

Students of the Software Quality Assurance specialization of Faculty of Informatics and Statistics, as well as other students interested in the software quality verification process or API principles, will find this work useful, as it gives a basic understanding of the API principles and describes methods for verifying API quality. For the same reason, this work can be useful for testers specialised in other testing areas that are planning to change their

field of work or just want to gain some basic understanding of the verification process in other testing types and levels.

This thesis can also be helpful to developers working with GraphQL API technology and give them an overview of the implementation of the test automation process for their solution.

Research methods

To accomplish the main and sub-goals, several methodologies were used during the process of writing this thesis. To understand the subject of the thesis, the author used the method of searching resources by keywords, to research the available existing materials concerning the topic. The results of the research, including the definition of keys and searched sources, are listed in a separate chapter — *1 Literature review*.

The author handled the research for forming a list of the existing GraphQL API test automation solution using Java by searching for information on relevant Web pages and repositories by defined keywords. The full process of the research is described in *5 GraphQL API test automation using Java*.

Comparing the process and evaluation of the solutions was handled according to the Multiple Criteria Decision Analysis (MCDA) method, its stages and application in the thesis are described in chapter *6 Evaluation of GraphQL API solutions for concrete company*. For specification of the company X needs the interview with the QA lead was handled to receive more information about the company's products and use cases and specify the requirements for test automation solutions.

During the evaluation of the existing automation, the GraphQL API solutions in the example project implementation the author uses are object-oriented programming methods. In the practical part of the work, several criteria were evaluated thanks to handling qualitative research by giving a questionnaire to the target group that was represented by the QA community of company X.

Assumptions and limitations

This thesis assumes from the reader minimal knowledge of the software development process and the role of testing in the software life cycle. Even though the theoretical part of the work presents the principles of API functionality, reader knowledge in this topic is an advantage.

The thesis is concentrated on the integration of the test automation process so the reader's previous experience with any other automation framework or library as well as previous experience with manual API testing will be a big advantage as he can better understand the use and methods of solutions described in the work.

During the evaluation process of test automation solutions applicability author uses Java programming language concepts, design patterns and libraries for implementation of

example test projects using different approaches. To understand this part of the work, basic knowledge of object-oriented language programming is an advantage, but not the condition. For those who are not interested in the technical part of the work chapter *6.5 Measuring the alternatives* can be omitted.

Expected benefits

The main goal of the thesis is to choose the best solution that can be used on a real-life project for GraphQL API test automation process. The output of the practical part of the work during which existing solutions using Java language are evaluated and the best possible solution is chosen is the first main benefit of the thesis. During the evaluation process author set up a testing project using different solutions from scratch that is also counted as a further benefit of the work as it gives a real example of the existing solutions in use. Besides recommending the solution for the real-life project author gives further recommendations of other methods and design patterns that can be implemented for the project purposes. Furthermore, the example of building a test project for GraphQL API tesing using different solutions can be adjusted to other real-life projects, as author brings the general idea of the use of automation tools for API quality verification.

Another output of the work is the description of the existing GraphQL API approaches to test automation using Java. In the compact description, the author gives an overview of the libraries and framework found including the introduction of their features and specifics and gives links to official documentation and useful online resources.

One of the other expected outputs is the description of API testing, its necessity, and challenges. Moreover, GraphQL is a new technology and there are not so many resources from which testers can easily get a basic understanding of the testing process. The thesis covers the principles of API testing explaining from the beginning the role of API in Web application architecture and explains the principles and ways of testing API functionality by given Test Cases templates and possible tests description.

Work structure

This thesis is structured in a way to lead the reader from the theoretical basis to the practical part in which existing GraphQL API solutions are compared. Chapter *1 - Literature review* describes the process of literature research that relates to the thesis problem.

In the theoretical part of the thesis in chapter *2 - Application Programming Interface* the author describes the principles of the Application Programming Interface, its use, and role in client-server architecture style used for Web application implementation, gives an overview of one of the well-known and highly used architecture styles – REST. In the following chapter – *3 GraphQL,* GraphQL query language principles and advantages are defined with the description of its features on the example of implemented custom GraphQL server.

In chapter *4 API testing,* the author focuses on the testing process of APIs, describes its basics and challenges, and specifies the benefits of test automation and challenges that can occur during API test automation.

Chapter *5 GraphQL API test automation using Java* provides an overview of the existing test automation GraphQL solutions that can be used for testing on the integration level.

6 Evaluation of GraphQL API solutions for concrete company describes the step-by-step process of comparison of the list of solutions found to choose the best possible solution for the real project. In the chapter, the author describes the company X itself its products and use cases, used technologies and previous approaches to API test automation. Then the author defines the criteria based on requirements received from company development and QA communities. Existing solutions are evaluated according to given criteria and the best possible solution is recommended with the introduction of features that can be added to the existing solution for its improvement.

1 Literature review

In order to achieve defined main goal of the master thesis literature research was carried out to find the most relevant resources for the topic of the work. Literature research was handled by searching according to chosen keywords in multiple resource portals. This chapter introduces the results of the research and each part of it describes resources found according to their type: professional publications, academic works or Internet resources.

1.1 Professional publications

Professional publications were searched using Internet portal https://scholar.google.com and digital libraries like Association for Computing Machinery (ACM) and ProQuest. Resources were searched by following keywords: “Client-server architecture”, “API”, “GraphQL vs REST”, “GraphQL API”, “GraphQL testing”, “API testing”, “Automation testing frameworks”.

1. **APIs: A Strategy Guide** published by Daniel Jacobson, Greg Brail and Dan Woods in 2011 [7]. The book is focused on describing the principles of Application Programming Interface. It talks about API business strategies and models, including API values chains, ways how to create them and design principles. Despite of that authors are also describing the role of security and privacy policies for APIs, shows how to operate an API and measure its success.
2. **Client-Server Web Apps with JavaScript and Java: Rich, Scalable, and RESTful** written and published by Casimit Saternos in 2014 [8]. This book is describing the principles of Client-server Web architecture, the use of Hypertext Transfer Protocol (HTTP) protocol in it, REST architecture style and JSON formats. Most parts of this book are concentrated on describing principles for developing Web Applications and technologies that are used for that like Java tools or JavaScript libraries. Another chapter of the book is focused on testing activities and describes types of testing including testing frameworks that can be used for API testing.
3. **Learning GraphQL and Relay** from Samer Buna, published in 2016. The book describes new GraphQL technology [9]. Starting with the reasons for using it and process of setting GraphQL server author than describes the principles of query language itself and defines different types of schema. The book is giving the understanding of the GraphQL structure and basic principles of working.
4. **Automating and Testing a REST API: A Case Study in API testing using: Java, REST Assured, Postman, Tracks, cURL and HTTP Proxies**, published by Alan John Richardson in 2017 [10]. Alan Richardson has several books that are focused on testing processes and the application of programming languages and frameworks in test automation process. This book is focused especial on testing REST API. Author describes the principles of API, HTTP and REST and then introduces different tools that can be used for automation testing with the source

code for several of them. Book gives the reader the basics of API testing process with practical examples of different tools used.

5. **Deviation Testing: A Test Case Generation. Technique for GraphQL APIs**, published by Daniela Meneses Vargas, Andreina Cota Vidaurre in 2018 [11]. Authors in their work the cover the topic of testing GraphQL testing and describing deviation testing technique that can be used for it and which can help increasing test coverage and finding bugs in GraphQL APIs.
6. **Study on GRAPHQL and Automation Testing,** written by in 2018. Authors describe the new GraphQL technology its properties and benefits [12]. Then authors are concentrating on automation testing techniques and existing tools that can be used for test automation. In the end of the work authors give an overview of tools that can be used for automating GraphQL API testing. Topics raised in publication are the closet one for the topic of master thesis.

1.2 Academic works

Academic works were searched in CIKS VŠE catalog as well as in the portal https://theses.cz/. Not so many academic works yet exist on chosen topic of GraphQL API testing. Academic works were searched by keyword "GraphQL API". Some of the found works are focused on GraphQL API implementation but not on testing their results. Following thesis are the most relevant for the chosen problematic.

1. **REST and GraphQL API Implementations Comparison** written by Šimon Podlipský in 2019 [13]. Master thesis is focused on implementation process of Web application using REST and GraphQL technology. In his master thesis author firstly describes the techniques of REST and GraphQL implementation and then compares their use by developing Web application using two different approaches. In his work author also evaluate the complexity of working with two different technologies.
2. **Performance of frameworks for declarative data fetching: An evaluation of Falcor and Relay+GraphQL** written by Mattias Cederlund in 2016 [14]. In his thesis author focused on performance of data fetching process using different technologies like Falcor, Relay, GraphQL Author handle the experiment with comparing named frameworks performance level by running tests for measuring their latency, data volume and sending different number of requests. At the end of his work author evaluate the results and gives the recommendation for frameworks and describes the difference between them considering specified metrics.
3. **API Design in Distributed Systems: A Comparison between GraphQL and REST** written by Thomas Eizinger in 2017 [15]. In his master thesis author introduces GraphQL and REST technologies in order to compare them according to chosen criteria. API technologies were compared in operation reusability, discoverability, component responsibility, simplicity, performance, interaction visibility and customizability. At the end of thesis author summarizes and evaluates comparison results and recommends which technology is better for what cases.

1.3 Internet resources

Big number of resources and information about GraphQL technology and test automation solutions for server-side can be found in the Internet by searching through Google engine. Relevant information was searched using following keywords: “GraphQL testing”, “GraphQL test automation framework”, “How to test GraphQL”, “GraphQL vs. REST”. The list of the most useful and relevant Internet portals where these topics were discussed is the following:

1. **GraphQL** – https://graphql.org/learn/
   GraphQL online documentation describes the principles of working with this technology [16]. It introduces step by step how to use it features and gives example of the code for implementing GraphQL in different programming languages. Moreover, this page gives the useful links to existing communities and blogs that are focused on GraphQL topic.
2. **Medium** – https://medium.com/
   Medium is an online publishing platform in which different point of views of developers about usage of GraphQL can be found.
3. **GitHub** – https://github.com/
   GitHub is a Web service for software development process based on git technology. In terms of the master thesis topic in GitHub’s accounts were found many different library examples connected to GraphQL API implementation or the use of automation libraries for server-side testing.

2 Application Programming Interface

This chapter introduces Web application client-server architecture and defines the role of API in the model. In following parts API on different layers of architecture are described with the examples and brief introduction of HTTP that enables frontend-to-backend interaction. In the following subchapter well-known REST architecture style principles and limitations are specified. Chapter fulfills the sub goal of the master thesis "*Define the role of the Application Programming Interface in Web applications*".

With the rapidly changing world the use of APIs has been changing as well and while at 1994 application programming interface was used for getting domain and page, era of Web in 2010 is trying to get rid of the pages and connect APIs and people straight together [7]. This means that through decades API stopped being taken just as interface for system-to-system communication and now can be sold as an independent online product. Today is the era of API economy - business models that enables secure access and exchange of data [17], when APIs can be used for integration of people, places, data, systems, algorithms, transactions and enable to turn business into platforms with ecosystems inside and outside of the enterprise.

Even though the use and approaches to API are being changed over decades, main purpose for the API remains the same – enable the communication between different systems. According to Gartner IT Glossary [18] API is:

„An interface that provides programmatic access to service functionality and data within an application or a database. It can be used as a building block for the development of new interactions with humans, other applications or smart devices. Companies use APIs to serve the needs of a digital transformation or an ecosystem, and start a platform business model. "

2.1 Client-server architecture

Web application is a software package that performs in a specified way and is situated on a remote server that can be accessed by a user through Internet [19]. A client-server architecture for building Web applications is a popular approach not only because it is the closest to the Web architecture itself, but also because of the number of advantages [20], some of which are the following:

- Manageable code thanks to separation between client and servers' tiers.
- Data can be delivered in Json or XML formats that is more readable format.
- Developing different parts of application in isolation. API, mobile devices on client side, version of new system can be developed separately with no related impact. This also allows to develop new features in parallel nor waiting on another task to be finished which are not related to each other.

- Ability to prototype helps to test and verify new ideas, clarify vague approached and provides clear communication about given requirements.
- Better application performance as client-side engines allow performance of comprehensive calculations in a way that server workload won't be visible for a client.

Client-server architecture is displayed on *Figure 1* below.

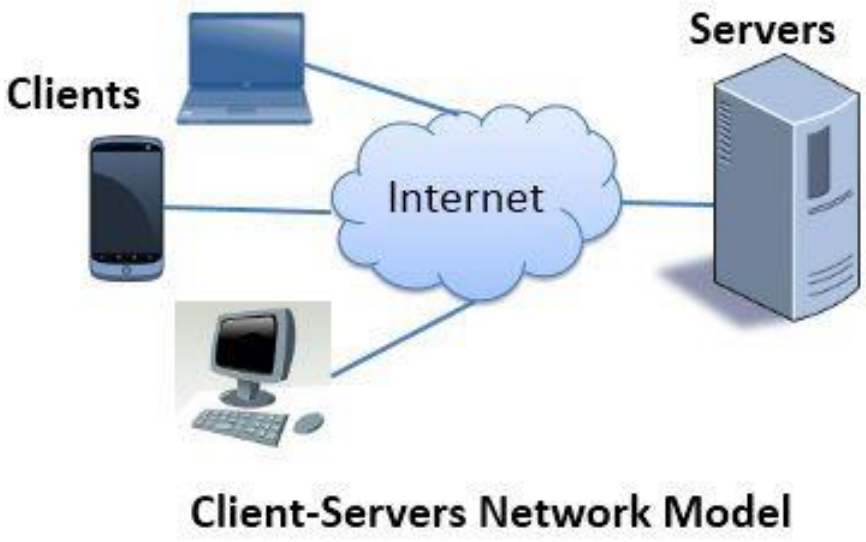


Figure 1 Client-Server network model (source: https://techdifferences.com/difference-between-client-server-and-peer-to-peer-network.html)

The model describes the way how server provides resources and services from its side to multiple clients (laptops, mobiles, personal computer). End user is using the client-side only, the rest part of the model is used by systems themselves. When end user provides any actions from the Graphical User Interface (GUI) on client-side his requests are sent to the server via Internet network. Server-side is used for hosting, delivering and managing resources and services that are used by client [21]. When the request is received from the client-side, server process it and sending the response through the Internet back to the client. In "Client-Servers Network Model" one server can handle multiple client requests at the same time, client can as well be connected to a several servers that provide different types of services. API can be implemented on both sides of Client-Server architecture.

Client-side APIs

On client-side APIs can be written with JavaScipt language and divided into Browsers APIs and Third part APIs.

Browser APIs are built into Web browser and can support front-end implementation by exposing data from the browser and surrounding computer environment. The most common browser APIs [22] are:

- Web Audio API – API for manipulating audio in the browser like altering the volume or applying effects.

- Manipulating with Document Object Model – API allows to manipulate with HTML and CSS files by applying new styles to page.
- Notifications APIs – API for retrieving data from device in a way useful for Web application like control and display system notifications to the user.
- Client-side storage – API that allows to save data between page loads that helps to work with the application with no Internet connection.

Third-Party API are not built into the browser so the code should be taken from somewhere else from the Web – from the Third-Party. Some of the most popular Third-Party APIs [23] are:

- Google API – API that allows to integrate Google Ads with another website
- Twitter API – API for displaying the latest tweets on the website
- YouTube API – API for searching and playing YouTube videos from another website
- Facebook API – API for user authorization, payment acceptance, targeted advertising and marketing campaigns.

Server-side APIs

In comparison to client-side APIs that are working on client layer only, server-side APIs is working like access point to the server and databases for the Web application [24], as displayed on *Figure 2*. Its main goal is getting requests send by client-side in the right format and place of the server-side, for example to the correct database, get the response from server and send it back to the client-side.

MuleSoft company – one of the leaders of the Gartner Magic Quadrant 2018 for Full Life

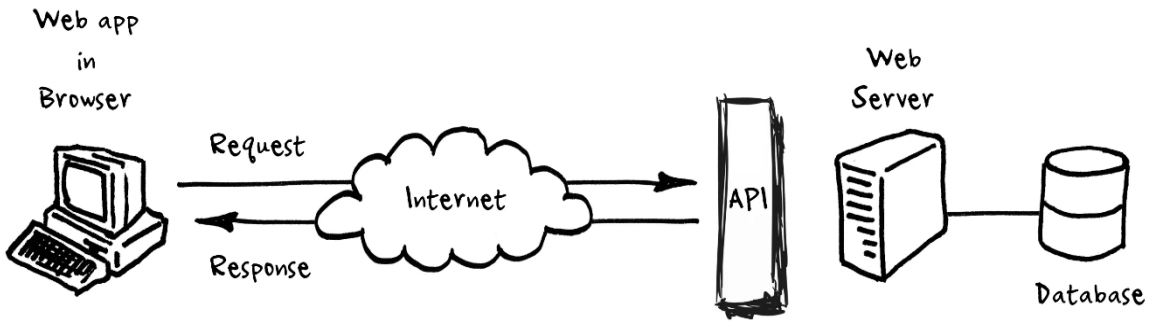


Figure 2 API in client-server architecture of Web application (source: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/)

Cycle API Management[1] [25], in their YouTube channel posted a video [27] in which they compare API to the waiter in the restaurant. When client decides which meal to choose, the

[1] API management consists of the activities connected to the full life cycle of application programming interface like planning, design, implementation, maintaining, versioning, testing, publication, consumption. Without full life cycle API management is impossible to build ecosystem and run effective API program [26].

waiter gets its order and bring it directly to the kitchen, where client requests and orders are handled by cooks that prepare a meal according to given specification. When meal is ready waiter takes it carefully from the kitchen and bring as a result to the client.

APIs are working the same way: client from its side using the Web application in the browser defines the requirements and send them with a "waiter" API to the kitchen, kitchen is a server that performs action based on given requests and after finalizing the result prepare the response and send it back with API to the client.

Here are couple of real examples of the use of server-side APIs:

- Login into Facebook – server-side API gets request from user with login credentials, data are sent to the server and if user exists and credentials are valid response is sent back with access token for the session.
- Searching for a flight – user enters flight information to the system; parameters are sent to API that handle the request and make the server start searching for the flight that suits given parameters.

One API can perform multiple tasks and operations as it has documentation and specification, which states how obtained information should be handled [28]. Massive Web applications like Facebook able user to provide different actions from login to storing photos, posting, making bargains and handle payments. For all these needs same API is used, but all requests that are coming used for different needs and should be provided on different server components and databases. Because of that Web application APIs consists of different points – endpoints, that can be accessed with requests for different needs. Endpoint is the entry point of the communication channel of two systems, that accesses server resources it needs to perform the required task [29]. Endpoints are playing important role in each Web API as they have impact on performance and productivity of Web application. Verification of functions of API endpoints can help to control that application provides the actions it is expected to.

Web application interaction APIs can be implemented in different forms using different technologies. For accessing Web services – software system designed to support interoperable machine-to-machine interaction over a network [30], can be used different protocols as Simple Object Access Protocol (SOAP), Remote Procedure Call (RPC) or REST architectural style [31].

- SOAP – protocol based on eXtensible Markup Language (XML) that provides definition for Web services interaction between each other or to the client application [32].
- RPC – protocol that enables data transfer of XML or JavaScript Object Notation (JSON) formats in client-server model [33].
- REST – architecture style with certain principles and requirements that enables simple communication through interfaces and identification and manipulating server resources (REST architecture will be described in subchapter *REST architecture style*).

Earlier named approaches to Web APIs are sending client requests with data to server via XML or JSON file formats. This process is handled by Hypertext Transfer Protocol (HTTP) protocol which characteristics and principals of working are described in the following section.

2.1.1 HTTP communication

The communication between client and server is handled by HTTP which is sending requests and response through Internet and acts as a transport system in client-server architecture. Main characteristics of HTTP protocol [34] are:

- *Connectionless*
  Each time client sends a request through HTTP the new connection between client and server is opened, when the response is successfully sent back the connection closes and client and server sides are not anymore connected to each other.
- *Media independent*
  HTTP protocol allows sending different data formats like text files, Microsoft Word files, JPEG images, HTML files, movies and many others. To distinguish each resource data type from each other HTTP tags them with label Multipurpose Internet Mail Extensions (MIME). When a Web browser receives data object it looks first at MIME tag, so it knows how to handle the object received.
- *Stateless*
  As HTTP according to first characteristic is connectionless, thanks to that neither client nor server of them can store information about each other across Web.

HTTP presents a suitable way of packaging data with useful information over Web using Unified Resource Locator (URL) for defining the resource path and specifying their location on the server-side [35]. HTTP sends requests and response from client to server via messages that are formatted according RFC822[2] message format for transferring the required data. Each message send through HTTP protocol should have following four items:

- Start line
- 0 or more header fields
- Empty line
- Optional body message

HTTP header provides necessary information about the send data and. In case of request with header client-side inform the server about its capabilities and specify method, content-type, charset, version of HTTP protocol used and requested Uniform Resource Identifiers (UNI) it is searching for. The most useful methods that are supported by HTTP/1.1 version request are described in the *Table 1* where HTTP methods are linked to related CRUD operations - basic functions of persistence storage as create, read, update, delete.

[2] RFC822 – Standard for ARPA Internet Text Messages [36]

Method	CRUD	Description
POST	Create	Send client data into a server API and creates a new resource. Data that are send to the server are stored in request body.
GET	Read	Used to retrieve data from specified resource in server.
PUT	Update	Used to update existing resource in the server.
PATCH		Used to provide partial modifications of specified resource.
DELETE	Delete	Used to remove the existing resource from the server.
HEADER		Same as GET but sends only status line and header.
CONNECT		Open a tunnel identified by given URI.
OPTIONS		Describes the communication options.
TRACE		Method used for debugging as it returns the input from client-side [37].

Table 1 Description of HTTP/1.1 methods (source: [34], [37])

In response for sent HTTP request server responds with response message that has the same RFC822 structure. In status line HTTP returns status code from server for received request. Header of HTTP response is used to inform the client about the result of the request that was received and processed [38]. Status codes consists 3-digit numbers and are categorized into 5 groups that are according to their first digit [34]. Status codes categories are described in the *Table 2*.

Code	Status	Description
1xx	Informal	Request was received by server, connection is open, process is continuing.
2xx	Success	Request was received and accepted by server.
3xx	Redirection	Further actions are taken in order to complete the request.
4xx	Client Error	Error occurred on the client side – incorrect syntax of HTTP request or request cannot be fulfilled.
5xx	Server Error	Error occurred on server side – valid request was received, but server fails to perform required operations.

Table 2 HTTP/1.1 response status codes (source: [34])

A knowledge of client-server architecture, API place in it and HTTP that is used for transmitting data on the Web is a benefit for designing better APIs. Following subchapter is introducing one the approaches of server-side API – REST architecture style.

2.2 REST architecture style

Representational State Transfer term was firstly publicised in doctoral dissertation "Architectural Styles and the Design of Network-based Software Architectures" of Roy Thomas Fielding American computer scientist in 2000. The idea of REST architecture style

that brought by him and his colleagues to the world is still used in the development of Web services.

REST architecture style in comparison to other protocols like SOAP or RPC used for APIs implementation doesn't describe the specific of API implementation but brings the set of principles, constraints and properties [39].

2.2.1 Principles and constraints

Before REST architecture style developers had to deal with SOAP for API integration because of that REST is compared to SOAP principles. One of the differences of the use of HTTP protocol that came with REST is the use of the URIs for defining resource paths. SOAP protocol is working with operations defined in Web Service Definition Language (WSDL) schema to get resources from the server-side, REST instead is using nouns to describe the endpoint with a noun to specify the resource location and use methods of HTTP protocol like GET, PUT, POST, DELETE to apply CRUD operation over chosen resources [40].

Another change is the use of JSON file format that allows to send readable data format that is consistent with client-side languages like JavaScript.

According to Roy T. Fielding dissertation [41] there are 6 constraints in REST architecture style:

1. Client-Server
   The use of client-server architecture style in which both sides have different sets of concerns. Functionality should be properly separated between client and server to improve scalability and make both independent. In this constraint client is displayed as a triggering process that displays information and perform requests and server is a reactive process that manipulates with given information.
2. Stateless
   No station state on the server side brings the idea that each request from client-side should contain all the necessary information for server to process the request.
3. Cacheable
   Cache performs as a mediator in client-server interaction in a way that requests can be considered cacheable and be reused in response to later requests.
4. Uniform Interface
   One common interface for API components must be defined and applied to each resource.
5. Layered System
   Layered system is adding proxy and gateways that make only the layer with interacting component visible and others "inner-layers" hidden.
6. Code-On-Demand
   Access of the resources is given to client component so its functionality can be extended to a deployed client.

2.2.2 Limitations

REST architecture style is highly used but described principles and constraints have their own limitations that are discussed by developers in their blogs and online portals like publishing platform Medium or blog on Good API [42]. Despite the common comments about REST being not descriptive enough without any guidelines and being challenging to keep consistence from client and server side, there are some other implementation limitations that are commonly being discussed [43]. Here are some of them:

- Multiple round-trips
  Under round-trip is meant a request that the client performs to the server and the response that the server sends back to the client [44]. In case when information that must be collected from a different URI paths should be displayed to user, multiple requests to different endpoints are be sent to get all the necessary information. For example, getting information about user's activity log and the list of user's followers on social network requires sending requests to multiple endpoints. In order to escape from this situation "One-Size-Fits-All" (OSFA) approach is used for endpoint implementation, when several resources are grouped together in order not to locate resources on multiple URI paths and send separate requests for them [45]. Such approach makes it hard and messy to maintain and handle the code that maps all URIs.
- Over/under fetching
  REST API implementation has fixed endpoints structures, that results in getting in some case some unnecessary information in the response message (over fetching) or opposite when specific endpoint doesn't provide enough information (under fetching) and here comes the need of multiple round trips [46]. Fixed structure of endpoints also leads to the problem of adjusting frontend on client side to the changes made on server side. In case that endpoint structure was changed it must be changed at each place on the frontend when the request is sent to that endpoint.

3 GraphQL

Chapter presents the GraphQL technology by describing it's history, used concepts, principles and specific features. Then the main differences of GraphQL and REST APIs implementation are showed. It helps to fulfil the sub goal of the master thesis "*Introduce the GraphQL technology*".

Even though endpoint based APIs like REST has its advantages and they were already well known among developers and were simple for implementation, its limitations like multiple round trips, over- and under-fetching, implementation of "One-Size-Fits-All" endpoints, that were named in previous chapter *2.2 REST architecture style*, made companies to start searching for better solutions that will surpass existing disadvantages. To overcome existing REST limitations several world-known companies introduced their approaches for their specific product cases.

Netflix, American media-service provider, to get rid of using OSFA REST endpoint for different devices came up with their own solution that they have successfully patented at 2013 [45]. They have implemented a custom layer, that is with sent request to a specific endpoint gather client information data and then parse, format and adapt gathered data of request before sending it to the server side. *Figure 3* below represents implemented custom layer.

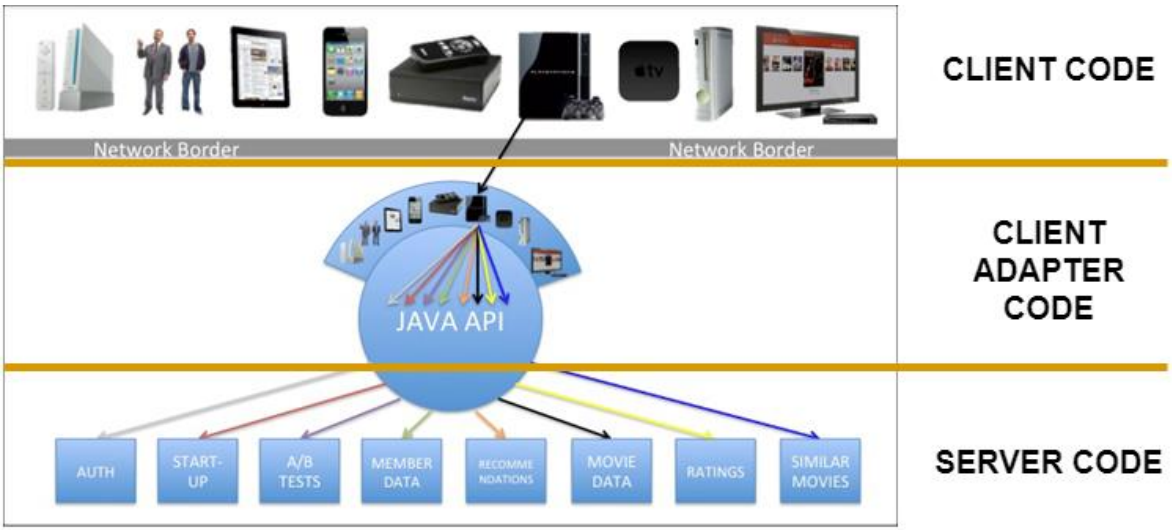


Figure 3 Client adapter code model implemented by Netflix (source: https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d)

When a specific device sends single request from client to server side, adapter receives it and parse into many different requests that are then addressed as multiple calls of Java API, each of those are related to dependent service [47]. Successful transactions are returned to adapter which prepares the content, prunes unwanted elements, handle errors and format the response before sending it back to the client.

On the other hand, in 2015 Soundcloud company – developer of an online audio distribution platform, was trying to solve the problem with long process of new features implementation that has an impact on already implemented functionality for their platforms [45]. Because of this issue, they have decided to change the architecture logic and started to implement separate API server for each use case. This pattern was named as "Backend for Frontend" and helped in maintaining ecosystem of microservice in a way that each use case can be optimized and modified without the worries of influencing another one [48]. Another approach was GraphQL technology introduced by Facebook, Inc. company.

3.1 History of GraphQL

Developing process of the GraphQL began at early 2012 when Facebook was dealing with implementation of "News Feed" for iPhone Operation System (iOS) mobile platform. On February 1, 2012 in Securities and Exchange Commission (SEC) S-1 form [49] – initial registration form for new securities required by the SEC for public companies that are based in the U.S, Facebook, Inc. stated [50] that one of the risks is connected to people using more mobile devices.

"Growth in use of Facebook through our mobile products, where we do not currently display ads, as a substitute for use on personal computers may negatively affect our revenue and financial results;"

At that time Facebook mobile applications on iOS and Android platforms were becoming more complex and required feeding more data to the user which made them to suffer from poor performance and unexpected crashes [51]. In terms of that Facebook decided to redesign and develop the new concept of "News Feed" API for mobile apps. Implementation of the "News Feed" required getting information about posts not only like post author, it's content, comments, number of given likes but also some nested data that were interconnected and recursive [52]. Old concept of used REST API required getting data from different endpoints and APIs which requires making multiple round-trips to get needed resources, what exactly had an impact on performance of applications. The concept of having a new technology that will allow developers to use better data-fetching capabilities and reduce the network usage caught several developers from different teams and made them start creating the new solution for "News Feed" [53].

Before implementing a new solution for "News Feed" Lee Byron, GraphQL co-creator and part of the service team at that time in Facebook, starts with raising the questions about why "News Feed" is needed, how people would use it and what for. All these led to understanding that "News Feed" API are complicated for a typical API solution, so he started to search for new ideas. At the same time another GraphQL co-creator Nick Schrock was already dealing with big amount of data on the server side and had an idea of making the API implementation simpler. He made the first GraphQL prototype at that time named as "SuperGraph" [53].

When two of them get together with Dan Schafer "News Feed" engineer from server side they started working on the first version of GraphQL that took them several months to meet the iOS "News Feed" needs. Dan Schafer describes GraphQL [54] in the following way:

"GraphQL is a query language for your API that shifts the contract between clients and servers that allows the server to say 'these are the capabilities that I exposed' and allows the clients to describe their requirements in a way that ultimately empowers product developers to build the products they want to create."

The implemented GraphQL solution was based on several principles that are described by Lee Byron [55]:

1. Efficient and predictable
   Data shape is known and can be derived from sent query that will help to eliminate the problem of assuming the data shape.
2. Static type system
   When GraphQL server is being built with its type system, data types are specified for all fields and arguments. This helps to describe from the very beginning what is possible and what is not to do and determine while preparing the request if it is valid or not. GraphQL server with its types fields and structure can be explored by using Browser integrated development environment (IDE) like GraphiQL. With the change of GraphQL server structure the documentation will also be changed.
3. Power to clients
   New product features can be easily added or deprecated on server side with leaving client side unaffected.

The launch of iOS mobile apps using new technology had a success and other teams wanted to transit using the GraphQL in their cases. First "Photos", then "Profiles" and "Groups" teams and later the rest of Facebook development started using it.

At that point on January 2015 at first React Conference Dan Schafer presented GraphQL technology [56]. The solution at this moment wasn't yet open source, moreover Facebook team only wanted to give their view on working with complicated APIs, but as the public was excited by their solution and wanted to use on their projects, Facebook decided to prepare and re-design a little bit their GraphQL solution so it can be applied to different cases. In 2015 Facebook announced the release of open-source GraphQL technology written in JavaScript language.

When the GraphQL was made public a huge community originated and since then many versions of GraphQL were built in number of languages such as Java, .NET, Scala, Python [52]. As the interest of new technology is still increasing – viz. *Figure 4*, companies around the world started to change their Web APIs to newly born GraphQL technology. Now GraphQL is used by GitHub, PayPal, Universe, FileJet, Atlassian and many others

companies a list of which can be found on GraphQL official pages https://graphql.org/users/.

Figure 4 Google Trends of searching "GraphQL" topic in past 5 years worldwide (source: https://trends.google.com/)

3.2 Features

GraphQL implementation was based on special principles described earlier that help to overcame existing limitations of other endpoint-based APIs like REST. These principles appear in query language as its features. Brian Kimotoki in his book *Beginning GraphQL: Fetch data faster and more efficiently whilst improving the overall performance of your web application* [57] specifies four GraphQL main features:

1. Hierarchical queries
2. Introspective
3. Strongly typed
4. Client-specified queries

This subchapter introduces each named feature with the code example of real GraphQL API. "Introspective" and "Strongly typed" features are covered in *3.2.2 Type system and schemas,* another two features are described in *3.2.3 Query Language*.

Custom simple GraphQL server was implemented based on tutorial *"GraphQL Java Example for Beginners [Spring Boot]"* in Swathi Prasad blog [58]. Even though description of implementation process of GraphQL server is not connected to the master thesis goals the whole process with code examples is described and attached in the *Annex A: GraphQL custom server implementation*. All attached screenshots are made from Playground –

graphical, interactive, in-browser GraphQL IDE, created by Prisma and based on GraphiQL [59].

3.2.1 GraphQL server description

Custom GraphQL server is implementing basic CRUD operations for manipulating with Data object. Each Data object has its own unique ID as well as characteristics Type and Example. Data object represents a data type with an example. Simple Data object example can be represented in JSON format as displayed on *Code 1*.

```
{
   "data":{
      "id":"1",
      "type":"int",
      "example":"1"
   }
}
```

Code 1 "Data" object example (source: author)

Operations that can be applied on Data object are described in *Table 3*.

Operation	CRUD	Description
createData	CREATE	Allows to add new Data with specified Type and Example. ID is automatically assigned to new instance of object.
getData	READ	Returns Data instance characteristics Type and Example for specified Data ID.
getListOfData	READ	Returns the list of all existence Data objects with all characteristics.
editData	UPDATE	Allows to update existent instance of Data. Only Type or Example characteristics of instance can be changed, ID remains the same.
deleteData	DELETE	Specified Data by ID is removed permanently from server.

Table 3 GraphQL server operations for Data object (source: author)

3.2.2 Type system and schemas

GraphQL is a server-side runtime that executes queries by using type system that user defined for their data [16]. In Playground custom GraphQL schema looks in a way that is displayed on *Figure 5*. Basic components of schema are object types – object that client can fetch from service, and the fields that are specified for that object. In custom GraphQL server DataType is object type and ID, Type and Example are its fields.

Figure 5 Custom GraphQL server schema part (source: author)

Type system helps to validate syntactic query validation and provide server responses with appropriate error messages. As GraphQL is being introspective it allows the client to inspect fields, types and queries. This allows the supportive IDE/editors like Playground or GraphiQL perform autocompletion and validation and throws error messages when the defined type system is violated [60]. When field that is not defined in GraphQL schema is used in a query IDEs display validation error as displayed on *Figure 6*.

Moreover, introspection of GraphQL allows querying "__schema" and "__type" fields that give information about schema object its type, input fields, enumeration values, kinds of fields, description, interfaces and others.

Figure 6 Example of query validation (source: author)

3.2.3 Query Language

Hierarchical query GraphQL features states that the query sent by a client has the same shape as the result that is returned by server. Server gives client its capability by type system and schema according to which client build its query.

Apart from being hierarchical queries are also client-specified, that means that client can choose what fields he wants to get back from server. This feature enables to overcome over-/under-fetching limitation of REST architectural style [57].

Beside from sending queries that are used only for data fetching GraphQL also allows sending mutations. In comparison to queries mutations are used to modify the data on server-side. Server can specify what action should certain operation provide. If custom GraphQL server had REST architectural style its operations will be sent with different REST methods as shown on the table *Table 4*.

Operation	HTTP method
createData	POST
getData, getListOfData	GET
editData	PUT
deleteData	DELETE

Table 4 HTTP methods used in REST architectural style with custom GraphQL server operations (source: author)

With GraphQL all requests are using POST method. With POST request queries are sent to GraphQL API having in body message operation name, query itself (can be both query and mutation) and list of variables specified in JSON format. Even though queries and mutations that are send are written in graphql format that is not JSON, in HTTP request they are sent in a request payload body which sends data as JSON object. Queries and mutations in the body should be parsed to JSON as visible in *Code 2*. Responses from GraphQL API are returned in JSON format.

```
{
   "operationName":"getListOfData",
   "query":"query getData {\r\n    getListOfData {\r\n        id\r\n        type\r\n
example\r\n    }\r\n}\r\n",
   "variables":null
}
```

Code 2 Request payload body for GraphQL API (source: author)

Following chapters introduces several useful GraphQL features that are used in practical part of master thesis.

Variables

In GraphQL queries client can declare a variable using “$” symbol of either scalar, enum or input object type. Variables can be optional or obligatory that can be specified in declaration part using “!” [16]. List of used variables is send in body via HTTP request or can be filled in Browser IDEs modal window like displayed on the following *Figure 7*.

Figure 7 Example of query variable usage (source: author)

Fragments

In both queries and mutations user can specify what fields he want to get back from server. In case of providing CRUD operation over a certain object fields that can be returned have the similar shape as it can be seen in query getData and mutation createData on *Figure 8*.

```
query getData($id: ID!) {
  getData(id: $id) {
    id
    type
    example
  }
}

mutation createData($typeValue: String!, $exampleValue: String!) {
  createData(type: $typeValue, example: $exampleValue) {
    id
    type
    example
  }
}
```

Figure 8 Example of similar data shapes in multiple queries (source: author)

The fields specified in each query are of the same type – DataType and can be brought together in a separate unit called a fragment. Fragments allow user to combine set of fields and reuse them in different queries [16] as displayed on *Figure 9*.

```
query getData($id: ID!) {
  getData(id: $id) {
    ...dataTypeFragment
  }
}

mutation createData($typeValue: String!, $exampleValue: String!) {
  createData(type: $typeValue, example: $exampleValue) {
    ...dataTypeFragment
  }
}

fragment dataTypeFragment on Data {
  id
  type
  example
}
```

Figure 9 Example of using fragment in GraphQL queries (source: author)

4 API testing

This chapter introduces the API testing, specifies relation between integration and API testing, defines API testing activities and the biggest challenges of API testing process. Following part of the chapter focuses on describing API test automation process including reasons and problems. Chapter fulfills one of the defined sub-goals of master thesis "*Characterize the process of API testing*".

Applications based on client-server architecture requires specific form of testing to prevent and predict system failures [61]. Software testing is the process consisting of activities for evaluation whether they satisfy defined requirements or not and reveal defects that may cause the product failure [6]. While verifying quality of Web applications different types of testing should be taken into consideration in order to ensure the better quality of the final product.

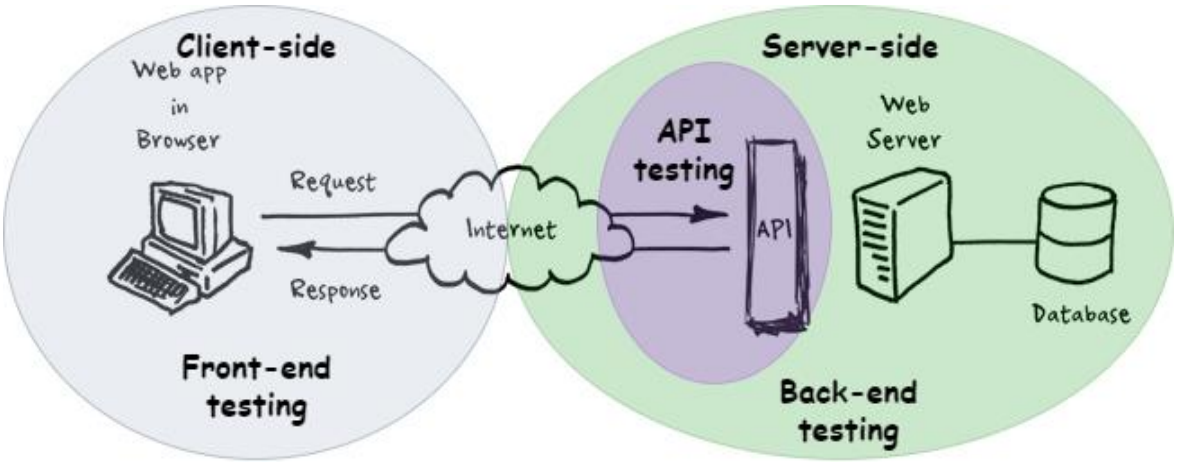


Figure 10 Types of testing in client-server architecture style for Web applications (source: http://www.robert-drummond.com/2013/05/08/how-to-build-a-restful-web-api-on-a-raspberry-pi-in-javascript-2/, modified by: author)

On *Figure 10* different testing types that can be applied on both sides of Web application architecture are displayed. Two bigger circles separate client and server sides from each other. Client-side verification is known as front-end testing process that verifies the Graphical User Interface by checking its functionality, usability, reliability, performance. On the server-side the backend-testing process is handled by verifying Web servers, database operations, business rule implementations, security and performance [62]. A part of server-side is an API that can be tested in both ways as a part of back-end testing and as a separate unit.

4.1 Characteristics of API testing

Front-end and back-end testing types are unofficial terms and more used in practice while specifying the type of the product part under test. According to International Software Testing Qualifications Board (ISTQB) there are four levels of testing: component,

integration, system and acceptance. Each of the test level is performed according to the related stage of development lifecycle that is visible on *Figure 11*Figure 11 V-model in software testing (source: https://www.testbytes.net/blog/v-model-and-w-model-software-testing/).

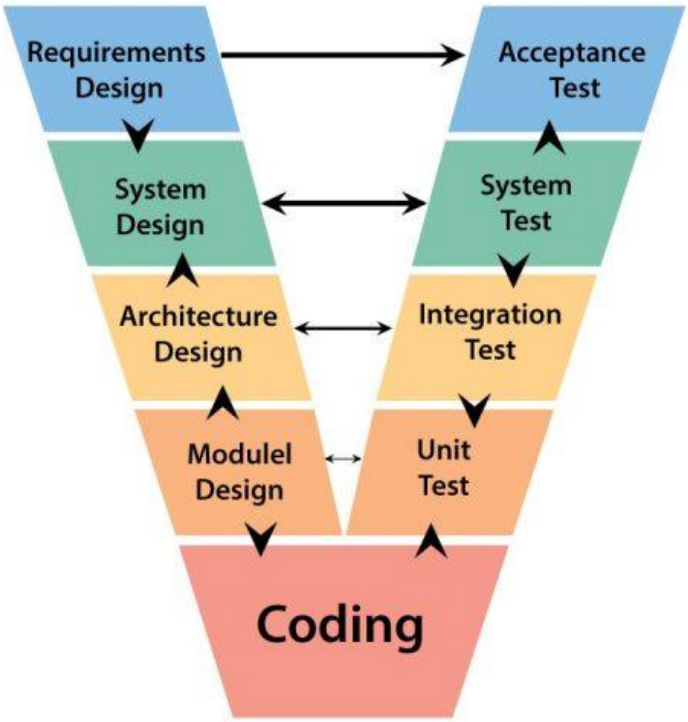


Figure 11 V-model in software testing (source: https://www.testbytes.net/blog/v-model-and-w-model-software-testing/)

Component testing

Component or unit testing focuses on testing separate units like software modules, objects, classes, methods that can be independently testable [63]. Unit testing is done by developer who wrote the code and defects are fixed with the highest priority. Unit tests are automated and are written in the same language as the code itself. One of the possible approaches to component testing is test-driven development when automated Test Cases are prepared before programming itself.

Integration testing

In ISTQB Glossary [6] integration testing is defined in the following way:

"Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems."

Integration testing test objects describe what should be tested on that level. these could be either subsystems, databases, infrastructures, microservices, interfaces and APIs [63]. Two different levels of integration testing exist depending on the test objects: component integration testing, system integration testing. Component integration testing is used to verify interactions between system components while system is checking integration n a

higher level like interfaces integration. Besides functional verification integration tests are also used for non-functional verification characteristics.

System testing

System testing verifies capabilities and behaviour of the whole product [63]. During system testing both functional and non-functional requirements are checked on the environment similar as much as possible to the production.

Acceptance testing

Higher level of testing is acceptance testing, that is provided on the final stage of development and its goal to validate that the system is complete and work as expected [63].

This leads to a conclusion that API testing is the type of integration testing level. API quality is tested by submitting commands to software or Web application interface under test [64]. API is an important part of client-server architecture, which allows sending data from client to server, which means that logic situated on API level should be carefully verified in terms of specified requirements [65]. API testing objectives are the same as for integration testing level:

- Reduce risk
- Verify functional and non-functional behavior is met according to design
- Build confidence in product quality
- Find bugs within interface
- Prevent defects from transition from server to client side

API testing can be done with incomplete system, it means that each endpoint can be tested separately without the dependency on other parts of the system to be implemented [66]. Because of that running API tests on Web application has a list of benefits [67]:

- Allows providing earlier independent testing activities of implemented logic
- Specifies the exact place of system failure on server side that helps to reduce time for fixing defects
- Ensures that application modules like third part APIs or multiple Web servers can work together

4.1.1 Testing activities

Apart from defined earlier test levels also different types of tests can be applied to system under test. According to ISO 9126 – international standard for the evaluation of software, there are 6 quality characteristics for product verification: functionality, reliability, usability, efficiency, maintainability, portability [68]. Relation of each quality characteristic to API testing activities is visible in *Table 5*.

Characteristic	Description	API testing activities
Functionality	Functional behaviour according to specification	Verification of API endpoints [69]: • Used data types • Returned status codes • Calling of another API/event
Reliability	Ability of system to maintain its services under defined conditions	• Handling network failures between client and server sides [69]
Usability	Ease of use in regard of system functional specification	• Usage of authentication methods • Correspondence of HTTP error code and message to system failure [70]
Efficiency	The use of system resources	• Measuring uptime and rate limiting (rate of received or sent traffic across network [71]) according to usage policies • Measuring server response time
Maintainability	Ability to identify and fix a system component	• Providing documentation of existing API endpoints with their usage • Verification of endpoint URLs logic scheme • Correspondence of HTTP error code and message to system failure [70]
Portability	Adoption to changes	• Verification of endpoint URLs logic scheme

Table 5 ISO 9126-1 software quality characteristics relation to API testing activities (source: [69], [70], [71])

For each quality characteristic several API testing activities can be handled. Some of testing activities can even relate to multiple quality characteristic. In order to give a practical understanding and describe principles of API testing next section gives test examples for functional verification of GraphQL server quality implemented in *3.2.1 GraphQL server description*.

Example of functional verification

Functional requirements on endpoint-based APIs can be checked by sending requests through HTTP protocol and verifying data in responses for the same request or sending other requests.

GraphQL operation's functionality do not distinguish HTTP methods and use only POST for sending requests, but still functional tests logic differs depending on type of request sent and the implemented logic on server side. General TC examples that can be applied on custom GraphQL server can be divided into verification of operations that allows retrieving

data from server (“getData”, “getListOfData”) and operations for modifying data (“createData”, “editData”, “deleteData”).

Test scenarios templates for verifying different types of operations are represented in *Table 6* [72]. Those test templates can be adjusted to any endpoint-based API like REST.

Retrieve data – positive scenario			
Test template description: template with positive test scenario for verifying endpoints functionality that enable fetching data from server side.			
Step 1	Prepare and send valid request that retrieves data from server without modifying them to the appropriate endpoint	**Expected result**	Request is sent to defined endpoint
Step 2	Verify status code returned from server	**Expected result**	Status code is 200
Step 3	Verify response data	**Expected result**	Data returned from server are correct and contain expected and relevant resources
Modify data – positive scenario			
Test template description: template with negative test scenario for verifying endpoints functionality that enable data modification on server side.			
Step 1	Prepare and send valid request with appropriate HTTP method	**Expected result**	Request is sent to defined endpoint
Step 2	Verify status code and error message returned	**Expected result**	Status code is 200
Step 3	Prepare and send request to retrieve modified data from server	**Expected result**	Request is sent to defined endpoint
Step 4	Verify status code and response data	**Expected result**	Status code is 200, all previous modifications were successfully saved and can be returned to the client
Retrieve and modify data - negative scenario			
Test template description: template with negative test scenario for verifying endpoints functionality. Tests checks API reflection on receiving invalid request like wrong data in request defined, use of wrong data type, incomplete request.			
Step 1	Prepare and send invalid request	**Expected result**	Request is sent to defined endpoint
Step 2	Verify status code and error message returned	**Expected result**	4xx status code is returned with expected error message

Table 6 Test scenarios templates for API functionality verification (source: author)

4.1.2 Challenges

Even though API testing has its own benefits, there are also several challenges while testing server-side interface of Web applications. API is holding the programming logic of server side through which client is communicated to the server. API is also responsible for better user experience as it enables interactions between each module, application and system on the server side. Moreover, modern applications are now more often composed of multiple services connecting to each other at runtime, which is making API testing even more challenging [73]. API testing challenges are being highly discussed on many Internet portals, books and personal blogs, some of them are giving recommendations and solutions regarding defined challenges. The list of challenges below is based on information from "The Art of Software testing" book written by Glenford J.Myers, Tom Badgett and Corey Sandler [4], CA technologies presentation of new approach to API testing "API Testing Guide" [73] and "Challenges Of API Testing" article posted in Resourcology blog [74].

Required technical skills

API has no UI, which means that tester require more technical skills for understanding the principles of its functionality and way of working with API calls and endpoints. After finding a defect in API layer the process of testing is not ended, each endpoint communicates to different database or server or even third part application, that's why for complete analysis of defect causes tester require to have a basic knowledge of working with databases and SQL language, understand XML and JSON file formats for evaluation error message returned from server side.

In "Automated software testing" book written by Elfreide Dustin, Jeff Rashka and John Paul authors raise the idea of running some of integration tests that are used for checking integrated components on unit testing levels [75]. Close to this idea was also a statement raised in "Introduction to Software testing" book by Paul Ammann and Jeff Offutt that members of development team are usually responsible for integration tests but not testers [66].

Business environment

Depending on business strategy and subject of Web application, API represents the logic hidden in server side that should be also verified [4]. Complex products have more complex requirements and business rules that should be verified that makes the process of analyzing and test design more difficult.

Infrastructure setup

Before testing API layer, infrastructure include databases, server's configuration and preparing, sometimes simulating Third-Party applications should be done according to given specification and in a closest way to real end user environment conditions.

Complexity of API structures

API structures are becoming more complex as they are often using multiple services. A single API call may trigger a parallel or serial action in different modules. It influences testing activities as more scenarios should be covered.

Dependent systems under construction

Even though one of the named benefits of API testing is earlier testing, it's not always a clear benefit. Some dependent systems of tested API part (endpoint) can be under construction. In this case for covering all TCs dependent system's functionality and behaviour are simulated and the tests are run against the mocked unit. Unfortunately, this may influence the reliability of tests result as such tests types like performance or loading. Mocked services cannot simulate real rate conditions like real production environment.

Maintenance of API schema

When server-side is under construction or new features are being implemented, API schema may be changed quite regular. In certain approaches to interface implementation like REST architecture style for example, even small change on server side require changes on API level and changes of prepared test data like JSON or XML files for making calls to endpoints.

API versioning

With the releasing of new product versions, the logic of API implementation may change multiple times and leads to existing of multiple API versions. In some cases, while certain functionality is being depreciated, the API still requires handling old version calls as some dependent system may not know about the change and still be using previous versions of the system. API should in these cases recognize missing values or depreciated function calls and assign default values to them. Such situations lead to creating more test scenarios.

Named in this section challenges occurs not only during test analysis, test design and manual test execution processes but also can take a place in stage of test automation process. Next subchapter describes the reason and advantages of API test automation.

4.2 API test automation

The trend of using test automation in development process was growing with the popularity of agile methodologies and Rapid Application Development (RAD), methodology that is focusing on adaptive processes instead of planning and scheduling activities in order to provide more frequent, incremental software build [75]. Defined methodologies include regular and frequent test activities as each small change of software could have an impact on already implemented features and functionalities. Automation of manual and repeated testing activities after each code change is being beneficial for organization in different perspectives. The finalized list of test automation benefits formed in "*Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey*" [76] is presented below:

- Improved product quality
- Higher test coverage
- Reduced testing time
- Reliability of test execution result
- Increase in quality confidence
- Reusability of tests
- Less human effort
- Reduction in cost
- Increased fault detection

Companies are trying to include test automation into development process as soon as possible to prevent defects on earlier stages and reduce the cost of fixing occurred system failures. However, not all test activities are needed to be automated, according to *Figure 12*.

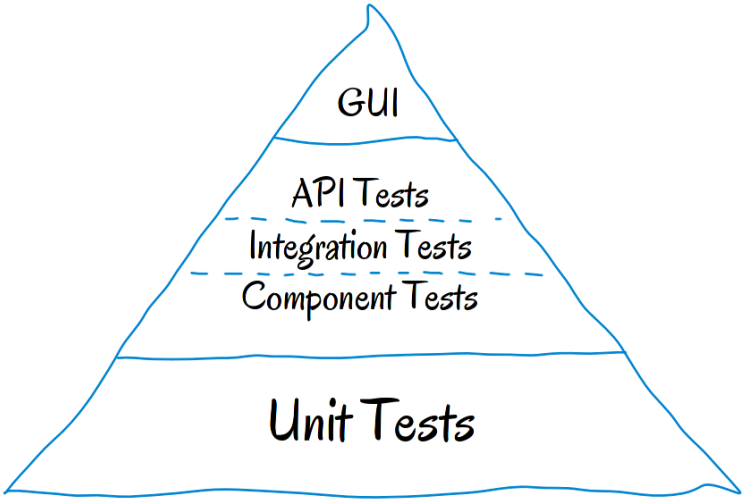


Figure 12 Mike Cohn's Test Automation Pyramid (source: https://www.360logica.com/blog/sneak-peek-test-framework-test-pyramid-testing-pyramid/)

First level of Test Automation Pyramid is covered with Unit tests that are written by developers. On the higher-level Component, Integration and API tests are located. These tests are implemented and maintained by testers and require more test automation effort than front-end tests that are situated on the last pyramid level.

Although, API test automation is being thought as one of the most vital, easiest and quickest testing to be done, it has some challenges directly related to challenges described in *4.1.2 API* testing *Challenges* [77]. *Table 7* below describes API test automation challenges and their relation to general API testing challenges.

API test automation challenges	Description	Related API testing challenges
Initial project setup	In some projects depending on infrastructure initial automation project setup may be a challenging activity that may last for a long time. This challenge may also be related to integration with existing eco system	• Infrastructure setup

Maintenance	Change of a product require continuous maintenance and update of already written Test Cases and suites. In API test automation testers are preparing requests that are being sent to the endpoints, change on server-side require updating created test data	• API versioning • Maintenance of API schema • Complexity of API structures
Skilled resources	Test automation solutions require knowledge of programming languages and libraries to be used	• Required technical skills
Integration with existing eco system	Integration of API test automation in already existent software build process in a company	

Table 7 API test automation challenges (source: [78])

Table above presents general API challenges that may occur on a project, they may vary depending on a product, technologies and logic used in implementation. Wide range of tools being used for test automation may help in evading some of named challenges and make the process of test automation implementation and maintenance easier for testers.

5 GraphQL API test automation using Java

This chapter focuses on describing possible solutions on GraphQL API test automation using Java. The chapter fulfils one of the set sub-goals of the thesis – "*Describe the existing solutions for the GraphQL API test automation using Java*".

In the first part of the chapter, a process of searching for possible solutions is described step by step. After finalising the list of found approaches for GraphQL test automation author gives a reader an overview of their specific features and characteristics.

5.1 Test automation solutions research

First, the author searched through Google Scholar and ACM digital library for the possible solutions for GraphQL test automation by the "GraphQL testing" keyword. ACM digital library gave no result for the searched keyword, Google Scholar returned only one article "Deviation Testing: A Test Case Generation. The technique for GraphQL APIs" in which no test automation tools are introduced.

Because of that reason the author searched Google portal with the same defined keyword, however, the results count was up to 2640 and required a more specific search. As in terms of thesis, the author is searching for approaches using Java programming language, keyword "java" was added to the search condition. GraphQL technology was from the beginning implemented using JavaScript and only then was adjusted to other languages like Java. The number of test automation approaches in JavaScript because of that is quite high and although adding "java" keyword helped to shorten but didn't remove all the results with GraphQL testing solutions using JavaScript and other JavaScript libraries. The list of keywords was expanded with the condition ".js" for excluding JavaScript file types from the search results. With the given condition, the number of returned results was 10 with only one of them related to the searched topic. In API test automation with Java, the usage of open-source frameworks and libraries is a common approach and most of them are described in GitHub Wiki pages, topics or README files as it is a popular software development platform [79]. That's why the search result was narrowed with the condition to search through the GitHub site and excluding the found GitHub topics. The final search rule is displayed on the *Code 3 Google search query for GraphQL test automation approaches using Java*.

```
site:github.com "graphql testing" java -.js -"GitHub Topics" -"Topic:"
```

Code 3 Google search query for GraphQL test automation approaches using Java (source: author)

The final count of the search result was 23 different GitHub pages and repositories, however, 11 of them referred to the same repository but to different source files. The rest remaining search results were analysed and evaluated against their relation to the thesis problematic. *Table 8* presents the result of GitHub repositories evaluation according to exclusion criteria like "Application source code with GraphQL server", "JavaScript library

usage", "Repositories under construction, "Other source code". Those repositories fulfilling at least one of the exclusion criteria are not relevant for the thesis. Fulfilled exclusion criterion is marked with "X" on the table.

#	Repository name	Application source code for GraphQL server	JavaScript library usage	Repositories under construction	Other source code
1	test-graphql-java	-	-	-	-
2	Rest-assured	-	-	-	-
3	researchpaper-java	X	-	-	-
4	gorm-graphql	X	-	-	-
5	Sangria-GraphQL-Neo4j-backend	X	-	-	-
6	graphql-spring-boot	-	-	-	X
7	commerce-cif-connector	X	-	-	-
8	my-thai-star	-	-	X	-
9	graphql-integration-test	-	X	-	-
10	karate	-	-	-	-
11	DevCouncilJune2018	-	-	-	X
12	ormoush-graphql-testing	-	-	X	-

Table 8 GraphQL test automation approaches using Java search results evaluation (source: [79])

After analysing all the found GitHub repositories only three of them are relevant and fully satisfy the aim of the research. Solutions presented in the repositories present the source code for testing GraphQL API. Each solution is described in the following subchapter.

5.2 Solution overview

In subchapter found GraphQL API test automation solutions are introduced to the reader by describing their features and principles. The author describes the main methods of found solutions by giving an example of Test Case implementation.

5.2.1 Test GraphQL Java

Test GraphQL Java is a library introduced by Vimal Selvam on 2. June 2019. He implemented a simple GraphQL testing solution that won't need to work with various dependencies, can use any Java testing framework and HTTP client, be able to transform GraphQL file to request payload [80].

In the example project, Vimal Selvam defines the prepareResponse method, displayed on *Figure 13,* for sending a request with already prepared request payload transferred to the method as a String. The request is sent using OkHttp HTTP client and then saved as a Response object.

```
private static final OkHttpClient client = new OkHttpClient();
private final String graphqlUri = "https://graphql-pokemon.now.sh/graphql";

private Response prepareResponse(String graphqlPayload) throws IOException {
    RequestBody body = RequestBody.create(MediaType.get("application/json; charset=utf-8"), graphqlPayload);
    Request request = new Request.Builder().url(graphqlUri).post(body).build();
    return client.newCall(request).execute();
}
```

Figure 13 Sending request method in Test GraphQL Java example Test Class (source: https://github.com/vimalrajselvam/test-graphql-java)

Request payload can be created using the parseGrapql library method from File object or InputStream and map of variables which the author creates as an ObjectNode object. As a variable map null value can be sent. Parse GraphQL method allows converting InputStream or File to String value and created new ObjectNode with query String and variables map for the payload – viz. *Figure 14*.

```
public static String parseGraphql(File file, ObjectNode variables) throws IOException {
    String graphqlFileContent = convertInputStreamToString(new FileInputStream(file));
    return convertToGraphqlString(graphqlFileContent, variables);
}

private static String convertToGraphqlString(String graphql, ObjectNode variables) throws JsonProcessingException {
    ObjectMapper oMapper = new ObjectMapper();
    ObjectNode oNode = oMapper.createObjectNode();
    oNode.put("query", graphql);
    oNode.set("variables", variables);
    return oMapper.writeValueAsString(oNode);
}
```

Figure 14 Test GraphQL Java library methods (source: https://github.com/vimalrajselvam/test-graphql-java)

This simple library for testing GraphQL can be installed by adding maven or Gradle dependency. Library source code is available on the GitHub repository under https://github.com/vimalrajselvam/test-graphql-java.

5.2.2 REST-assured

REST-assured is an open-source Java-based library designed for simplifying the testing REST services using Java [81]. The first version of REST-assured dependency could be downloaded since the 1st of December 2010. By October 2019 during the time of preparing the current thesis, REST-assured has already introduced the 4.0.0. version.

REST-assured is built on top of the HTTP Builder and supports POST, GET, PUT, DELETE, OPTIONS, PATCH, and HEAD requests. REST-assured allows the user to validate and verify responses for saved requests [82].

A simple example of the test implemented with the REST-assured library is displayed in *Figure 15*.

```
@Test
public void givenMovie_whenMakingPostRequestToMovieEndpoint_thenCorrect() {
    Map<String, String> request = new HashMap<>();
    request.put("id", "11");
    request.put("name", "movie1");
    request.put("synopsis", "summary1");

    int movieId = given().contentType("application/json")
      .body(request)
      .when()
      .post(uri + "/movie")
      .then()
      .assertThat()
      .statusCode(HttpStatus.CREATED.value())
      .extract()
      .path("id");
    assertThat(movieId).isEqualTo(11);
}
```

Figure 15 REST-assured test example (source: https://www.baeldung.com/rest-assured-response)

In the example, Test Case basic REST-assured methods are used. The requests to the server are sent using the "Given-When-Then" structure. In the given block, the user sent the request parameters as headers and body. REST-assured also provides special methods for setting request parameters, such as defining the content type or adding authentication headers – viz. *Figure 16*.

```
given().auth()
  .basic("user1", "user1Pass")
  .when()
  .get("http://localhost:8080/spring-security-rest-basic-auth/api/foos/1")
  .then()
  .assertThat()
  .statusCode(HttpStatus.OK.value());
```

Figure 16 Basic user authentication example code with REST-assured (source: https://www.baeldung.com/rest-assured-authentication)

The user authentication with REST-assured allows you to test and validate the API security. The library allows multiple authentication schemes [83], such as:

- Basic Authentication
- Digest Authentication
- Form Authentication
- OAuth[3] 1 and OAuth 2

[3] OAuth – open protocol that allows secure authorization from Web, mobile and desktop applications [84]

When the request parameters are specified in the “Given” block, the user defines what method should be used to send a request like POST, GET, etc in “When” block. After sending the request to the given endpoint, the user can set an additional verification by checking the status code received from the server.

When the request is sent to the server and a response is received, in “Then” block the user specifies the verification rules that can be applied to the response received both in JSON and XML format. The user can apply assertions on the value with the body function, save the value into Response Object in JSON format or extract the value and save it to the defined variable. On the extracted response object basic JUnit assertion functions can be used for response verification.

Apart from all the described features, REST-assured also allows logging the information about the requests sent and the received responses, which significantly helps in finding out the cause of the test failure. The logging functions can be enabled in multiple places [85]:

- Log request details
- Log response details
- Log response on error occurred
- Log response when validation failed

On REST-assured Wiki pages in GitHub repository under https://github.com/rest-assured/rest-assured/wiki can be found the description of all the library features and methods. Multiple tutorials explaining how REST-assured can be used in practice can be found on the Internet, for example, Baeldung [85] and Tutorialspoint [86].

Although REST-assured has a big Internet community of users, in the official documentation there is no information about testing GraphQL APIs. However, this topic is being discussed in other communities and blog pages and based on that is obvious that GraphQL API test automation with REST-assured is also possible. The only difference is that for GraphQL API test automation, the user must prepare the request payload with parsed as a String query value and variables map [87].

5.2.3 Karate

Karate is an open-source Java-based Behaviour Driven Development (BDD) framework for HTTP API testing. Karate is built on top of the Cucumber framework and uses Gherkin plain language parser to describe the tested features. With the Karate framework, different types of tests can be automated, like functional API tests, performance tests and mocks. The Karate framework allows the verification of both JSON and XML response formats [88].

A simple Karate test can be visible on *Figure 17*. Test scenarios themselves are written in the “feature” files and can be run with the test runner with JUnit 4 or called directly from the test class with Karate annotation with JUnit 5.

When the name of the Test Case in the “Given” block is set, other request parameters are being specified, like URL or request body message. After sending a request with the defined method in the “When” block, the user can verify the response status code and the property values in “Then” block using prepared methods and functions.

```
Scenario: Testing a POST endpoint with request body
Given url 'http://localhost:8080/user/create'
And request { id: '1234' , name: 'John Smith'}
When method POST
Then status 200
And match $ contains {id:"#notnull"}
```

Figure 17 Karate test example (source: https://www.baeldung.com/karate-rest-api-testing)

Karate allows saving the request body in separate files and calling them from multiple tests that help maintain the test data in a project. The test data can be saved not only in JSON and XML format but also, for example, in CSV, TXT, or JavaScript format. In the request, the user can specify arguments and then call from the test a request with a given parameter value. Apart from that, they can use features (Test Cases) inside other features, which reduces the number of method implementation.

Karate allows you to use different formats as a request payload, such as files in GraphQL format. To send a request to the GraphQL API request query, it does not need to be parsed to String and saved separately as a request payload, it can be just be called from the file or saved as a text variable as is visible on *Figure 18*.

```
Scenario Outline:
  # note the 'text' keyword instead of 'def'
  * text query =
    """
    {
      hero(name: "<name>") {
        height
        mass
      }
    }
    """
  Given path 'graphql'
  And request { query: '#(query)' }
  And header Accept = 'application/json'
  When method post
  Then status 200

  Examples:
    | name  |
    | John  |
    | Smith |
```

Figure 18 Karate GraphQL test example (source: https://github.com/intuit/karate)

Karate, as well as Cucumber, provides user-friendly test reports and allows storing the logging of the executed test output into XML files. A detailed description of Karate framework features can be found on the GitHub repository under https://github.com/intuit/karate.

6 Evaluation of GraphQL API solutions for concrete company

The following chapter describes the process of test automation solutions evaluation for the real company's needs. This chapter helps to achieve one of the defined sub-goals– "*Evaluate the application of the existing solutions on a real-life project*" and by what helps to achieve the main goal of this thesis.

Multiple Criteria Decision Analysis (MCDA) method was chosen for process evaluation. MCDA method is handled in the following steps [89]:

1. Context definition
2. Introduce available alternatives
3. Select the criteria for evaluation
4. Specify criteria importance (weights)
5. Measure the impact of each alternative on the selected criteria
6. Evaluation of the alternatives

In *6.1 Context definition* company X is introduced by giving its product characteristics and use cases as well as defining the reasons for from REST to GraphQL API implementation. In this part also the requirements received for test automation solutions from company X are described.

The next part of the chapter *6.2 Alternatives introduction* is concentrated on introducing available alternatives – solutions used in the experiment. Evaluation criteria specified in the following subchapter are based on the received requirements from the development team and the QA community of company X. When the criteria are introduced author in *6.4 Weights specification* specifies their importance – weights, that are then used in quantitative evaluation process.

The following parts of *6.5 Measuring the alternatives* are describing the measurement process and the impact of each alternative solution on the given criteria.

At the end of the chapter in *6.6 Alternatives evaluation*, the author evaluates the chosen alternatives using Weighted Sum Approach (WSA) [89] for evaluation MCDA alternatives and recommends a solution for company X, giving additional further recommendations.

6.1 Context definition

In the first step of the MCDA context of the analysis should be defined. The main goal of the thesis is to compare existing GraphQL API test automation solutions to the recommend one that can be used in a real project, the context of MCDA than can be defined in the following way:

Choose the most useful GraphQL API test automation solution for integration testing in company X.

In terms of thesis, a real company X that is now searching for the right solution for integration testing was chosen as they have freshly experienced the transition from the REST API approach to GraphQL. Company X overview and information written in the subchapter was received in the interview handled with the QA community leader in company X. The real company name, as well as its employees, are anonymized at the request of the company.

6.1.1 Company overview

Company X is a Canadian-Czech company operating on the global market for more than 10 years. Company X is developing an AI-driven platform consisting of different modules and allowing its users to manage, monitor, and analyse their data. The clients are mostly big companies that work daily with big data and require assistance for their management. The platform offered by company X allows solving the following cases:

- Data profiling and discovery
  Data profiling allows understanding the data patterns by visualising the results of analysis and representing the relationships between the data used. Apart from that data discovery enables the client to discover not only structured data but also analysing text-based data.
- Data catalog and glossary
  In the data catalog, clients can store and search for metadata using the business-friendly interface. To any data entity stored in the data catalog, the user can assign business terms. The glossary can be created by the organization and contains only term instances relevant to the organization.
- Master and reference data management
  This module includes consolidating multiple data from different sources into one golden record that helps to clean the data and get rid of redundant or outdated information. Clients can also store and manage their reference data in one place.
- Big data processing
  The platform can be used for processing an enormous range of data and provide a detailed data quality analysis before applying the required data transformations on them.
- Data quality management
  Data quality management includes data discovery and profiling, as well as automated AI-powered processes of discovering metadata. All analysis results can be stored and displayed to the end business user on the fully configurable custom dashboard.

Depending on the chosen Use Case, the platform can be bought with one or more modules. To solve each of the cases described earlier, the platform users should, in the beginning, connect to their data source (databases where their data are stored) or load data directly into the platform. This is the first and most important functionality of the platform.

The platform allows loading data of different formats and from different sources like CSV files, Excel spreadsheets, Databases. Getting data from databases requires some more actions to be taken by the user to create a connection to the existing data source. Users of the platform can handle the following actions with the data sources:

1. Create a new data source by defining connection parameters
2. See the details of the created data source connection
3. Edit the details of the data source connection
4. Delete the created data source connection

The actions described respond to basic CRUD database operations, these actions are the first and the most important in the system as without connectivity to the data source the rest of the platform modules features cannot be accessed. To improve product quality and prevent the situations when the basic functionality might be affected by changes made with new features implementation, QA engineers chose described functionality to be one of the automated test suites to be run after each change applied on the server by developers.

Development teams

The development group in company X is formed of almost 100 people that are divided into 11 teams according to platform module implementation. Depending on the team focus number of QAs in the team varies from one to up to three people. QA engineers, as well as back-end and front-end developers, form separate communities that have regular meetings for making decisions regarded all teams. Each approach for the test automation process is firstly being discussed within the team and with the proof-of-concept prepared is introduced to the community after what the final decision is made, and one common approach is chosen.

Used technologies

The desktop part of the platform is implemented using Eclipse IDE. The server-side is written in Java using Spring and Kotlin technologies and on client-side, the Web applications are programmed using TypeScript and React.

The QA community has common tools for different types of test automation implemented across the whole company by different teams. The server-side API testing was handled using the Cucumber framework with Java programming language that is described in the following sections. For the Web application test automation, the QA decided on using a custom framework based on Selenium WebDriver and Java. The desktop platform quality is verified by the Rich Client Platform Testing Tool test automation tool for Eclipse-based applications.

Test automation with REST API

For functionality verification on the server-side, the API testing is being handled by QA engineers using the Cucumber framework that supports BDD. The chosen solution was used for 2 years until the decision regarding changing the API implementation approach was

taken by the developers' community. The test automation using the Cucumber framework according to the QA community has several advantages:

- No Java programming skills are needed as Cucumber provides all the methods for sending requests and verifying values received in response from the server
- The Cucumber framework is using Gherkin syntax that allows automated tests to be self-descriptive and easy understandable

Unfortunately, even with the mentioned advantages of this solution, the QA community was thinking of changing the approach for the API test automation as the Cucumber framework had some limitations and the test automation process had become a challenging activity. These limitations are:

- The cucumber methods are not enough to consistently maintain the gross test automation project through different teams and require the implementation of custom additional methods for which Java programming skills are needed
- Cucumber provides the user only with a prepared class of given methods and functions, custom classes are hardly compatible with framework methods
- Variables from the test cannot be transferred to another test (global variables cannot be created for the whole test suite)

The QA community luckily experienced only half of the defined API test automation challenges mentioned in *4.2 API test automation* like "Maintenance" and "Skilled resources", but even those were enough to start searching for a new better solution.

Even though many discussions about changing the framework used for test automation were being handled for almost 3 months, only the transition from REST to GraphQL API implementation made the QA community stop using the Cucumber framework and search for other solutions that can be used for GraphQL API test automation.

6.1.2 GraphQL API

With the REST API implementation used earlier, company X faced several complications that made them start thinking about a new approach to API. Among these complications are:

- Aggregated and composed requests
  When on the Web page the information about the user with the list of his created data sources needed to be displayed, multiple requests were send from client-side to different URL paths in order to fetch all necessary information.
  When updating the information about an existing data source, the client-side needed to send several requests, first with the PUT method and second with the GET method, to fetch the updated information from the server to the user.
- Inconsistent endpoints
  In a growing company, multiple teams implemented API endpoints in their own way with no strict design model. The inconsistency of the API endpoints implementation

led to over-fetching redundant information from different resources while integrating the platform modules.

After various discussions and considerations of the number of existing alternatives that will help evade the defined complications, the development community decided in favour of the transition from REST to GraphQL API implementation. The new approach allows developers to define the required fields in the requests as well as the response fields to be returned from the server after modifying the data. Moreover, the GraphQL type schema helps to frame the back-end developers in multiple teams as it gives a sort of a design module they are following while implementing new endpoints.

The implementation of the GraphQL API technology led to the depreciation of all the API tests created earlier, and now the QA community of company X is searching for a new test automation approach.

6.1.3 Requirements for API testing

Based on previous experience with the REST API test automation QA community in conjunction with the development team specified their own list of requirements for a new integration testing approach. Each requirement is described in *Table 9*.

ID	Name	Description	Obligatory
REQ_001	Tests are written in Java programming language	Developers team by agreement with the QA community require API tests to be a part of the server project as it will help to earlier defect detection and preventing system failure. This requirement helps to overcome one of the mentioned API challenges – initial project setup.	Yes
REQ_002	Gherkin free approach	Because of the previous experience with Cucumber, the QA community decided not to use next time tools or frameworks that are using Gherkin syntax. This requirement helps to overcome one of the mentioned API challenges – skilled resources.	Yes
REQ_003	Solution library or framework can be installed with Gradle system	As API tests are a part of server project test module settings must be identical with the main project settings. The server project is using Gradle build-automation system that making it also obligatory for new solutions to use the same build system. This requirement helps to overcome one of the mentioned API challenge – integration with the existing ecosystem.	Yes
REQ_004	Resource files can be reused in multiple test suites	Because of the dynamic schema implementation on the server-side, the QA community is expecting a high number of	-

		resource files used in the test project, that's why the solution should allow reusing of resource files in different Test Cases and suites. This requirement helps to overcome one of the mentioned API challenges – maintenance.	
REQ_005	Ability to add user authorization header in the request	Web application can be accessed only by authorised users so to the requests send to API	Yes
REQ_006	Tests can be parametrised	With a high number of different cases (different data sources for example) that must be handled for product quality verification, the QA community needs to have a solution that allows test parametrization or can use other Java frameworks that have defined functionality. This requirement helps to overcome one of the mentioned API challenges – maintenance.	-
REQ_007	Ability to log the test run with giving informational error messages on test failure	Logs and error messages on run of automated TCs should provide tester all necessary information for understanding the cause of test failure.	-
REQ_008	Requests are using GraphQL variables and fragments features	GraphQL has implemented useful features like variables and fragments that could be beneficial to use in automated TCs.	-
REQ_009	Test project architecture	QA community describe their own principles that should be used in the test project. All test classes should be extended from one base class - BaseTest, that implements methods for sending requests and receiving a response with status code 200 validation. Send request method should return a response if given condition for status code was satisfied. Returned response is then being used in all tests for verification of response structure and data from the server. This class should provide a variable map for GraphQL requests variables that will be sent each time with the request to the API endpoint. Tester can use variable map in all test suites that are extended from BaseTest. This requirement helps to overcome one of the mentioned API challenges – initial project setup.	Yes

Table 9 List of requirements for the integration testing approach from company X (source: author)

In column Obligatory marked requirements their implementation in a new test automation approach is compulsory. Not marked as obligatory requirements are also important but their lack is not crucial to company X and QA engineers can handle and implement all necessary additional functionality.

6.2 Alternatives introduction

In *5.2 Solution overview,* three different solutions to GraphQL API test automation are introduced, each of those could be the alternative for the evaluation process and all of them fulfill the first given requirement – *REQ_001 - Automated tests should be written in Java*. However, development and QA teams from company X specified another requirement that influences the list of possible alternatives. One of the given requirements – *REQ_002 - Gherkin free approach,* sets the condition for the solution's implemented features.

Involving solutions that don't fulfill defined requirements into the evaluation process is unnecessary, as in the end these solutions won't be taken into consideration for the final decision. Because of that reason, the list of possible alternatives in MCDA is shortened according to the stated requirement.

Only two solutions from the given list of possible three fulfil given conditions and can be announced as possible alternatives for defined context. That's why only *Test GraphQL Java* and *Rest assured* solutions are evaluated with the selected criteria in the following steps of MCDA analysis.

6.3 Criteria definition

MCDA criteria are defined based on the specified requirements in *6.1.3 Requirements for API testing*. Not all of the requirements can be easily transformed into criteria. For example, *REQ_001 - Automated tests should be written in Java* and *REQ_002 - Gherkin free approach* criteria were already applied earlier on the list of alternatives.

In addition to them, the author specifies another set of criteria that will help in the solutions evaluation process. The final list of evaluation criteria with their short description is represented in *Table 10* below.

ID	Name	Description	REQ_ID
CR_001	Documentation, community support	The existence of Internet user's community (discussions in Stack Overflow online community), documentation provided on the Internet (official documentation, GitHub README file, tutorials in the Internet, YouTube tutorial videos), online support (discussions in Stack Overflow online community), resolution level of issues (number of resolved issues is higher than number of opened).	-

CR_002*	Initial project setup time	Time spent by tester on initial project setup with the pre-defined conditions.	REQ_009
CR_003	Usage of Gradle	The ability to install the solution with the build-automation Gradle system.	REQ_003
CR_004	Compatibility with other Java frameworks and libraries	Compatibility of the solution with other Java frameworks and libraries, like Junit4, Junit 5, Jackson, Json[4].	-
CR_005	User authorization	Ability to set basic authorization header in the request.	REQ_005
CR_006*	CRUD tests implementation time	Time spent by tester on CRUD tests implementation with the pre-defined conditions.	-
CR_007	Reusable resources	Ability to use resources in different test classes. Requests can be saved in the project separately under the resource directory and called from test classes.	REQ_004
CR_008	Ability to use GraphQL features	Solution ability to use GraphQL features in test cases and prepared requests like variables and fragments.	REQ_008
CR_009	Test parametrization	Ability to configure the test's parameterization with multiple pre-defined parameters.	REQ_006
CR_010	The ease of understanding solution methods	The level of understanding of methods and logic of solution implementation by the target group of users.	-
CR_011	Test reports	Framework ability to generate test reports or allow the usage of another framework, library, system for generating test results reports.	REQ_007
CR_012	Tests run time	Calculated test run time for the whole test suite.	-
CR_013	Error messages and logs informational content	The informational content of error messages and logs in the used solution for sending an invalid query to the server.	-
CR_014	Custom methods implementation	Potential methods that should be additionally implemented in the test project for initial project setup criteria or CRUD tests implementation.	-

Table 10 List of criteria for the MCDA method (source: author)

Some of the given criteria from the table above (marked with star symbol in the table) require more complicated actions to be taken for their evaluation. The process of measuring these criteria impact on alternatives cannot be described in the table and requires specific

[4] A list of libraries and frameworks were defined on request by company X.

introduction. The following sections introduces the process and evaluation conditions for measuring the impact of some of the criteria on defined alternatives.

CR_002 Initial project setup time

REQ_009 - Test project architecture – is a practical requirement for the test project structure and used principles and design patterns in it. Requirement description assumes about using Object Oriented Programming concepts that are directly connected to *REQ_001* and solution language. This requirement cannot be fully transformed to the criteria as its implementation depends on QA specialist knowledge of design patterns and programming skills but not on chosen solution features. But still, despite it, *REQ_009* was linked to *CR_002* which goal is to evaluate the time spent on the initial project setup. Requirement is then added to the initial setup project and the implementation time of the pattern described in *REQ_009* and in a list of activities for *CR_002* will be added to the total time.

Initial project setup means the first setup of the project from scratch with no pre-installed libraries using chosen solutions.

List of preconditions:

1. Initial project setup for both libraries is verified on the same computer
2. IntelliJ IDEA is successfully installed
3. Java 1.8 is installed on the computer
4. Previous experience in integration testing (1-3 years experience)
5. Previous experience in Java and usage of JUnit framework for test automation (1-3 years experience)

The total time of the stage is counted as a total time spent on the following activities, including time spent on dealing with issues that occurred.

List of activities of initial project setup:

1. Import of the solution library
2. Import of all other necessary libraries and project dependencies
3. BaseTest class implementation
    a. Test class is extended from BaseTest class
    b. BaseTest class allows sending requests and receiving responses
    c. Responses from BaseTest can be used in other test classes for verifying returned properties values and response structure
    d. Variables used in GraphQL queries should be initialized in BaseTest
4. First test “Get GraphQL structure” implementation
    a. URI initialization
    b. Sending specified query for getting schema structure
    c. Status code response verification

To check the ability of solution to work with any GraphQL API and sending requests to it simple test for getting GraphQL Server schema of custom GraphQL server implemented and running earlier is added to the Initial project setup process.

First test "Get GraphQL structure" steps are described in *Table 11*. Query for getting schema structure is attached in the *Annex B: GraphQL requests used in solutions evaluation*.

Get GraphQL structure			
Test description: Test is verifying the status of the GraphQL server by getting its schema structure and checking the returned status code.			
Preconditions: Running GraphQL server implemented according to Annex A: GraphQL custom server implementation			
Step 1	Send attached query to get the GraphQL schema to endpoint http://localhost:8080/graphql	**Expected result**	Request is sent to defined endpoint
Step 2	Verify status code returned from the server	**Expected result**	Status code is 200

Table 11 Get GraphQL structure Test Case (source: author)

CR_006 CRUD tests implementation time

In *6.1.1 Company overview* company X Use Cases were described with the given overview of the important platform functionality – manipulating the existing data source. Actions the user can apply to data sources within platform modules are related to basic CRUD operations in the database. Because of that fact, the criterion was set to measure the time needed for implementation of the CRUD test suite that contains tests for verification of each database operation. Test suite does not check the real company GraphQL API but uses the custom GraphQL server implemented earlier that allows providing the same CRUD operations over Data object.

Criterion has a list of preconditions needed for evaluation of the library's impact.

List of preconditions:

1. No previous experience with test automation using chosen alternatives
2. Tests are written after *Initial project setup* stage
3. Resource files with queries and mutations for tests already created under the test resources folder in the project directory
4. Test suite is run on clear custom GraphQL server with no data

Test suite is represented by four tests that are used for basic CRUD functionality verification; each basic database operation is verified by a separate test. In terms of thesis work, only positive test scenarios are implemented. Into the total time of *CRUD test implementation* is included time for the automation process of four TCs in one test suite. Test case steps are described in *Annex C: CRUD Test Cases*.

Test data like GraphQL queries and mutations are also prepared and can be found in *Annex B: GraphQL requests used in solutions evaluation*. Test automation of received TCs should be implemented with respect to prepared requests, which means that given requests shouldn't be modified and variable maps and fragments features would be used in each test if the library allows the use of these features.

6.3.1 Criteria evaluation values

The impact of alternatives on each criterion can be measured by values. All criteria are divided into three different groups by the evaluation method applied to them. Several criteria are grouped as they are measured in time values, another group of criteria is evaluated by the target group from company X and the last group of criteria is formed of the evaluation values assigned to them.

The first group of criteria is represented by *CR_002, CR_006,* and *CR_012* which measures the time of defined processes like implementation stages or time of test run of automated test scenarios. Time for criteria *CR_002* and *CR_006* is defined in minutes, while time for *CR_012* is represented in milliseconds. All values are normalised in *6.6 Alternatives evaluation* subchapter. Criteria from these groups are considered as minimizing criteria as the higher value they have the less benefit they bring. Other criteria from the rest of the groups are maximising criteria.

CR_010 and *CR_013* are grouped as their evaluation values are set according to questionnaires that received a target group for evaluation of solutions abilities. The target group is formed by QA engineers from the company X QA community. Target group is formed of 15 QA engineers working for company X on average for 1,5 years, has experience with previous Cucumber test automation approach and are specialised on back-end testing. The target group received a questionnaire prepared with code examples from both library projects to receive the subjective opinion of different approaches from the primary group of people who are expected to work directly with one of the evaluated solutions. Questionnaires were made with Google Forms and their templates are attached in *Annex D: MCDA questionnaires*. For evaluation of *CR_010* target group received a source code from test project with BaseTest class and Edit existing Data test attached in *Annex D: MCDA questionnaires* and was asked to evaluate the ease of understanding solution methods with the described rates: 1 – "*It is hard to understand solution methods",* 2 – "*I understood solution methods but I have some questions to its usage"*, 3 – "*I completely understood used methods".* For evaluation of *CR_013* to target group were shown logs and errors messages attached in *Annex D: MCDA questionnaires* so they can rate their informational content with 1 – "*Log gives no information about the test failure*", 2 – "*I partly understood what caused test failure*", 3 – "*I completely understood the error and can solve occurred problem*". From values received from target group average[5] value is calculated in each criterion measurement section.

The rest of criteria are forming the biggest group of criteria: *CR_001, CR_003 – CR_005, CR_007 – CR_009, CR_011, CR_014* are rated with numeric values from 1 to 3, where 1 is "*not satisfying*", 2 stands for "*partly satisfying*" and 3 – "*fully satisfying*". *CR_014 Custom methods implementation* criterion is rated with values 1 and 3, where 1 is "*more than 1*

[5] Average value is calculated in the following way: all the numbers are added up and then divided by how many numbers there are

additional methods are needed", 2 stands for "*1 additional method is needed*" and 3 – "*no additional method implementation is needed*".

Alternatives measurement as well as the implementation of Initial project setup and CRUD TCs preparation is handled by the author of the thesis, as she fulfils the specified earlier preconditions for the given criteria.

6.4 Weights specification

For specifying the weights of each criterion the rank ordering method was applied [89]. The target group was asked to evaluate the importance of the defined earlier criteria by answering the questionnaire, attached in *Annex D: MCDA questionnaires*. Target group was asked to rate the given criteria as: 1 – "*least important*", 2 – "*important*", 3 – "*most important*". From target group answers the criterion average value was calculated.

Based on calculated average value to each criterion was assigned rank from the interval <1, 14>, where 14 – "*most important*" and 1 – "*least important*". In case when criteria obtained the same importance rate the average rank was calculated.

Then each criterion weight is calculated according to the formula $v_i = \frac{b_i}{\sum_{i=1}^{k} b_i}$; [89], where b_i is the rank value assigned for each criteria and k represents the number of criteria and the highest rank assigned to the most important criterion. Total rank value was calculated using the formula $\sum_{i=1}^{k} b_i = \frac{k(k+1)}{2}$; [89]. *Table 12* represents calculatuion process of weights with the final result.

Criteria	1 - least important	2 - important	3 - most important	Average value	Rank	Weight
CR_001 Documentation, community support	0	5	10	2.67	14	0.13
CR_002 Initial project setup time	3	10	2	1.93	1.50	0.01
CR_003 Usage of Gradle	1	7	7	2.40	9.50	0.09
CR_004 Compatibility with other Java frameworks and libraries	0	7	8	2.53	12	0.11
CR_005 User authorization	3	6	6	2.20	3.50	0.03

CR_006 CRUD tests implementation time	3	6	6	2.20	3.50	0.03
CR_007 Reusable resources	1	5	9	2.53	12	0.11
CR_008 Ability to use GraphQL features	0	7	8	2.53	12	0.11
CR_009 Test parametrization	1	7	7	2.40	9.50	0.09
CR_010 The ease of understanding solution methods	1	8	6	2.33	7	0.07
CR_011 Test reports	3	4	8	2.33	7	0.07
CR_012 Tests run time	3	10	2	1.93	1.50	0.01
CR_013 Error messages and logs informational content	2	6	7	2.33	7	0.07
CR_014 Custom methods implementation	3	5	7	2.27	5	0.05
Total:					105	

Table 12 Criteria evaluation results with weight calculated (source: author)

6.5 Measuring the alternatives

This subchapter describes the process of solution evaluation according to the given criteria. Based on characteristics given in this chapter later in *6.6 Alternatives evaluation* the evaluation values are assigned to the solution. Each of the solutions alternative's correlation to the criteria is described in separate sections.

6.5.1 Test GraphQL Java

Subchapter concentrates on Test GraphQL Java library measurement for defined criteria. A general overview of the library is given in *5.2 Solution overview*.

CR_001 Documentation, community support

Test GraphQL API library is quite new and was introduced by VimalRaj Selvam in June 2019 [80]. There is no official documentation or Web page with library methods described apart from the GitHub repository.

Prepared library source code is saved in the GitHub repository that also contains a README file with the description of library methods and examples of usage [90]. No issues were yet found and opened nor resolved. By 31 October 2019 GitHub code was used only once and watched by three people in total.

By the same date, no YoutTube tutorials videos yet exist for the introduced approach for GraphQL testing nor discussions on Stack Overflow started about the introduced approach.

As not a good deal of information can be found on the Internet, the library just partly satisfies the given criterion.

CR_002 Initial project setup time

In the first phase of the evaluation process, the project itself was set up. A new Java project that uses Gradle build-automation system was created. Library dependency was imported from Maven Repository into the project, in addition to the Test GraphQL Java library, itself OkHhtp library was also downloaded. *Figure 19* represents the dependency section of the project.

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.4.2'
    compile 'com.vimalselvam:test-graphql-java:1.0.0'
    compile 'com.squareup.okhttp3:okhttp:4.2.2'
}
```

Figure 19 Dependency installed in Test GraphQL Java library example project (source: author)

In GitHub repository, the author demonstrates the library usage with TestNG Java test automation solution, but according to specified requirements from company X JUnit framework must be used in test automation project, that was the reason for installing also Jupiter library that enables usage of JUnit both versions 4 and 5 functionalities.

When all dependencies were installed, the author created the InitialProjectSetup class for automating the getServerSchema Test Case according to TCs example given in GitHub library repositories. In the GitHub project, an example of preparing requests, receiving and saving response was already implemented as well as the methods for reading GraphQL request's format.

According to specified actions for *CR_002*, BaseTest class was added during the project setup. In BaseTest class implementation, the author adjusted the example usage of solution methods to specified earlier requirements.

In BaseTest class, that can be seen in *Code 4, the* following features were incorporated:

1. GraphQL URI variable is used as local in prepareResponse method
2. Variables map of type ObjectNode is created in BaseTest class
3. SendRequest method with resource fileName argument is added
4. FileName argument is used in file path for specifying request file location in resources
5. SendRequest method is used for sending requests
6. Status code in received response in the sendRequest method is verified and if the returned code is 200 sendRequest method returns the response as an object

```
class BaseTest {

    private static final OkHttpClient client = new OkHttpClient();
    public ObjectNode variables = new ObjectMapper().createObjectNode();

    private Response prepareResponse(String graphqlPayload) throws IOException {
        RequestBody body = RequestBody.create(MediaType.get("application/json;
charset=utf-8"), graphqlPayload);
        String graphqlUri = "http://localhost:8080/graphql";
        Request request = new Request.Builder()
                .url(graphqlUri)
                .post(body)
                .build();
        return client.newCall(request).execute();
    }

    Response sendRequest(String fileName) throws IOException {
        // Read a graphql file as an input stream
        InputStream iStream = TestClass.class.getResourceAsStream("/graphql/" + fileName +
".graphql");

        // Now parse the graphql file to a request payload string
        String graphqlPayload = GraphqlTemplate.parseGraphql(iStream, variables);

        Response response = prepareResponse(graphqlPayload);

        assertEquals(response.code(), 200);

        return response;
    }
```

Code 4 BaseTest class implementation with Test GraphQL Java library (source: author)

The first test itself is implemented in the following way – *Code 5*. The test class is extended from the BaseTest class and implemented Test Case uses its methods. For the first test, no

additional assertions are needed that's why only resource file name is specified as an argument and status code verification is handled inside the sendRequest method.

```
public class GetServerSchema extends BaseTest {

    @Test
    public void getServerSchema() throws IOException {
        sendRequest("getServerSchema");
    }
}
```

Code 5 GetServerSchema Test Case implementation with Test GraphQL Java library (source: author)

The whole Initial project setup from creating a new project to BaseTest class implementation took 60 minutes.

CR_003 Usage of Gradle

As was mentioned in the previous section with the measurement of *CR_002,* the project from the beginning was set up using the Gradle system, which means that the solution fully satisfies this criterion.

CR_004 Compatibility with other Java frameworks and libraries

During the Initial project setup author already tried to install and use other Java frameworks and libraries like JUnit 4.12 and Jupiter library for using JUnit 5.4.2. This leads to the conclusion that the solution fully satisfies the criterion.

CR_005 User authorization

OkHttp library that solution uses for sending requests to endpoint allows adding header with authorization parameter. Header can be added to the request builder as displayed in *Code 6*Code 6 OkHttp request builder example using authorization header.

```
        Request request = new Request.Builder()
                .url(graphqlUri)
                .post(body)
                .header("Authorization", "Basic dXNlcjpwYXNzd3ByZA==")
                .build();
```

Code 6 OkHttp request builder example using authorization header (source: author)

According to Google, research OkHttp allows not only Basic authorization but also OAuth. Criterion is then considered to be fully satisfied.

CR_006 CRUD tests implementation time

According to the criterion description, a test suite with 4 Test Cases was implemented during this phase. Tests were automated using given Test Cases in *Annex C: CRUD Test*

Cases. Example Test Case for editing Data instance is displayed in *Code 7*. The whole test suite is attached in *Annex E: CRUD test suite implementation*.

```
@Test
public void editExistingData() throws IOException {
    variables.put("typeValue", "character").put("exampleValue", "A");

    Response response = sendRequest("createDataWIthFragment");

    // Save response as JsonNode
    String jsonData = response.body().string();
    JsonNode jsonNode = new ObjectMapper().readTree(jsonData);

    // Get new data instance id
    String id = jsonNode.get("data").get("createData").get("id").asText();

    // Change example value and save into variables
    variables.put("id", id).put("exampleValue", "B");
    response = sendRequest("editData");

    // Get updated data instance
    response = sendRequest("getDataWithFragment");

    // Save response as JsonNode
    jsonData = response.body().string();
    jsonNode = new ObjectMapper().readTree(jsonData);

    assertEquals("B", jsonNode.get("data").get("getData").get("example").asText());

}
```

Code 7 Edit existing Data test implementation with Test GraphQL Java library (source: author)

First into variables map values of new Data fields were added. The first request sent in the test is the mutation that creates a new Data instance with given values from the map.

In the solution usage example for verifying values in response, the author saves response body message to String variable from which creates a JsonNode object. The same approach was used in the CRUD suite implementation. JsonNode object allows us to work with the response body and save the ID of the Data as a String value. Saved ID is also added to the map with the new example value for verifying modification functionality of the server.

As a next step request for editing Data with given ID is sent. Response is again s saved firstly as String and is later converted to JsonNode. JUnit 5 assertEquals method is used for verification of changed value in Data instance.

Total time spent on CRUD test suite implementation is 35 minutes.

CR_007 Reusable resources

Prepared requests can be saved and used in different classes in the test project. Library provides multiple ways of resource file import like InputStream or File imports, so there was no need for additional methods implementation. *CR_007* is fully satisfied.

CR_008 Ability to use GraphQL features

The library allows users to send GraphQL requests both with and without variables map. However, there is no implementation for using fragments in the query which means that criterion is only partly satisfied.

CR_009 Test parametrization

Library successfully proved working with other Java frameworks like JUnit which means that it can use its functionality. All TCs can be easily parametrised using JUnit, criterion is then considered being fully satisfied.

CR_010 The ease of understanding solution methods

For this criterion evaluation target group received a questionnaire with a code example of BaseTest class and Edit existing Data test to evaluate the ease of a given solution. The questionnaire template is prepared and attached in *Annex D: MCDA questionnaires*. For the questionnaire was received the following result – viz. *Figure 20*. From the target group, 8 people rated Test GraphQL Java library methods with "*I completely understood used methods*", 6 – *"I understood solution methods, but I have some questions to its usage*" and 1 with "*It is hard to understand solution methods*". The calculated average value is 2,4 that is rounded down to 2.

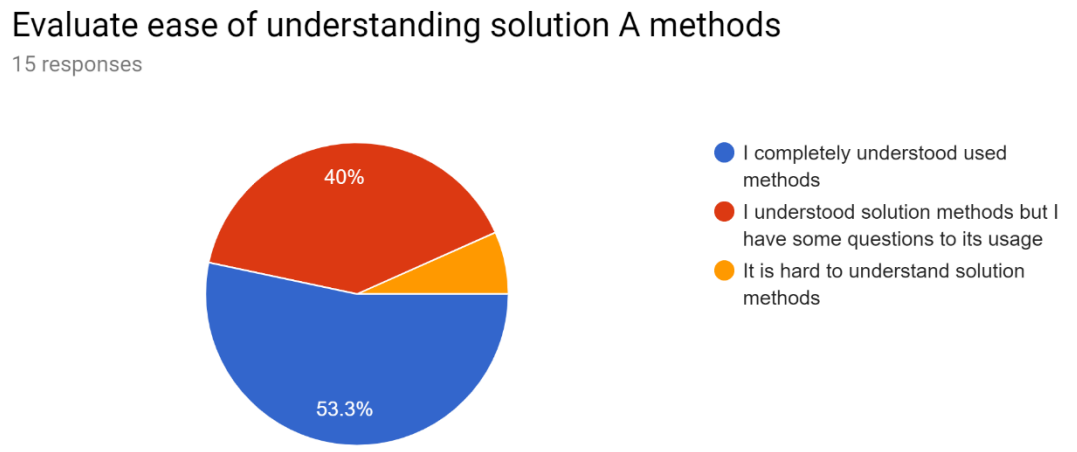


Figure 20 The ease of understanding solution A's methods - questionnaire results (source: author)

CR_011 Test reports

In the library, no special implementation of test reports is added, which means that the format and style of the test report is directly connected to the chosen Java framework for running automated Test Cases. Test reports were successfully generated using with JUnit framework and saved as XML files after each test run. The criterion is fully satisfied.

CR_012 Test run time

CRUD suite is run with Gradle, *Figure 21* represents the tests run time for the whole suite and each test separately.

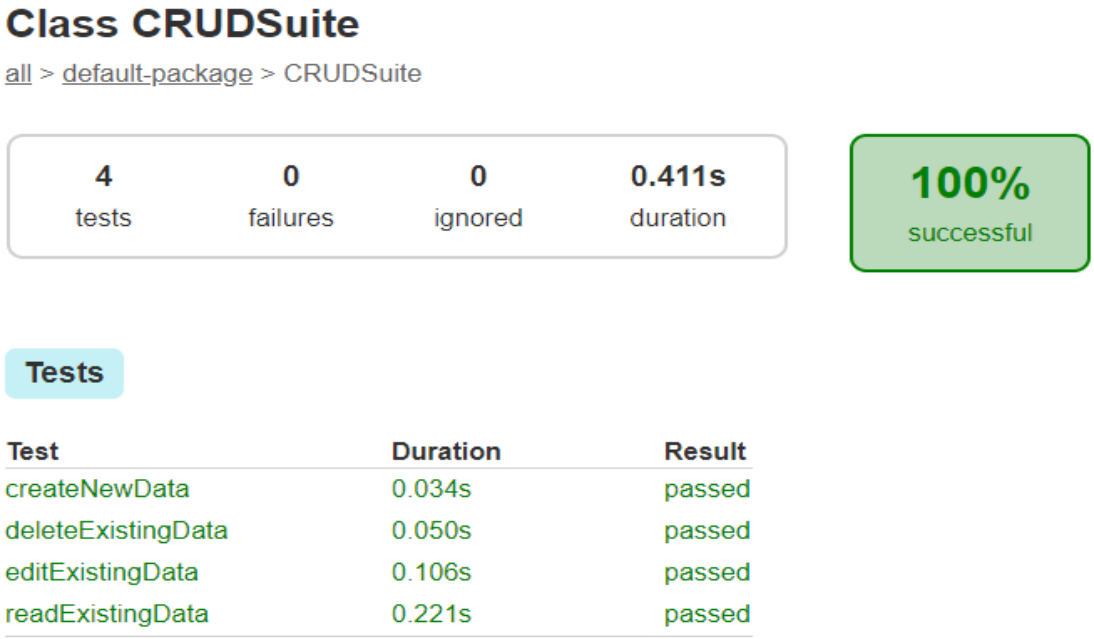


Figure 21 CRUD suite test run time using Test GraphQL Java library (source: author)

The average time for the first 3 test suite runs of the suite is taken into criterion evaluation. *Table 13* represents the time for each run with the average.

	First run	Second run	Third run	Average
Time	1.412s	0.376s	0.411s	0.733s

Table 13 Calculating of average test suite run time for Test GraphQL Java (source: author)

The final test suite run time for criteria evaluation is 733ms.

CR_013 Error messages and logs informational content

As in Test GraphQL Java library author implemented special methods for parsing GraphQL requests author-verified error messages for sending invalid queries displayed in *Code 8*. Query uses a fragment that does not exist that will cause an error for the request sent to the server.

```
query getData($id: ID!){
    getData(id: $id){
```

```
        ...dataTypeFragments
    }
}

fragment dataTypeFragment on Data {
    id
    type
    example
}
```

Code 8 Invalid request example (source: author)

Errors are not handled by the library and the user receives the following output in console displayed in *Figure 22* while executing Test Case that uses invalid query.

```
> Task :cleanTest
> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test

java.lang.NullPointerException
    at CRUDSuite.readData(CRUDSuite.java:63) <43 internal calls>
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) <1 internal call>
    at java.lang.Thread.run(Thread.java:745)

CRUDSuite > readData STARTED
CRUDSuite > readData FAILED
    java.lang.NullPointerException
        at CRUDSuite.readData(CRUDSuite.java:63)
1 test completed, 1 failed
> Task :test FAILED
FAILURE: Build failed with an exception.
* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///C:/development/master-thesis/test-graphql/build/reports/tests/test/index.html
* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output. Run with --scan to get full insights.
* Get more help at https://help.gradle.org
BUILD FAILED in 3s
4 actionable tasks: 4 executed
```

Figure 22 Test GraphQL Java error message example (source: author)

In Test GraphQL Java solution OkHttp library is used for sending requests to the API server, however, there are no examples of enabling logging of the sent requests in the library GitHub repository. OkHttp library itself enables logging but this should be an additional set by the users.

The informational content of the console output with the error message was evaluated by the target group in the questionnaire. The questionnaire template with questionnaire results are attached in *Annex D: MCDA questionnaires*. From target group, 9 people consider Test GraphQL Java solution as "*Log gives no information about the test failure*" and 6 as "*I partly understood what caused test failure*" – viz. *Figure 34*, the criterion received an average value 1.

CR_014 Custom methods implementation

During the measurement of the first alternative author created the list of methods that must be additionally implemented to fully cover the requirements given by company X.

The solution does not allow the full usage of GraphQL features like fragments that may have a big impact on resource and test data size. For example project 3 different requests could reuse the same fragment that can be saved in the project only once. But the solution does not allow adding fragments to the query which means that fragment is saved with each request in 3 files. Changes in the GraphQL schema structure related to the fragment require changing it in multiple places. A method that allows adding a fragment to the requests should be added.

Error messages and logs are not enough for easy understanding of the mistake or cause of the error and require additional method implementation and enabling OkHttp logging in the project.

The total count of additional methods that should be added into the library is 2, criterion is rated with value 1.

6.5.2 REST-assured

REST-assured library overview is given in *5.2 Solution overview*. The following sections are concentrated on the description of another alternative measurement process for defined criteria.

CR_001 Documentation, community support

The REST-assured library has official pages with the links to the useful pages as release notes, news, documentation, legacy rights, and Frequently Asked Questions answered [81].

Source library code can be found in the GitHub repository with the Wiki page that describes step by step the usage of the library with the examples. The REST-assured library is by the 31. October 2019 is watched by 312 users and was marked with star by 4.3 thousand users. In REST-assured README file is attached to the link to the Google support group. REST-assured has already 815 issues resolved and 257 opened. Last closed issue by the 31. October 2019 was resolved on 25. October 2019 that confirms the high support of the library [82]. Not only in GitHub REST-assured repository useful information about the use of solution can be found. On Stack Overflow by the 31 October 2019 500 different discussions by the keyword “rest-assured” are found and on YouTube channel multiple video tutorials can be found using the same keyword, viz. *Figure 23*.

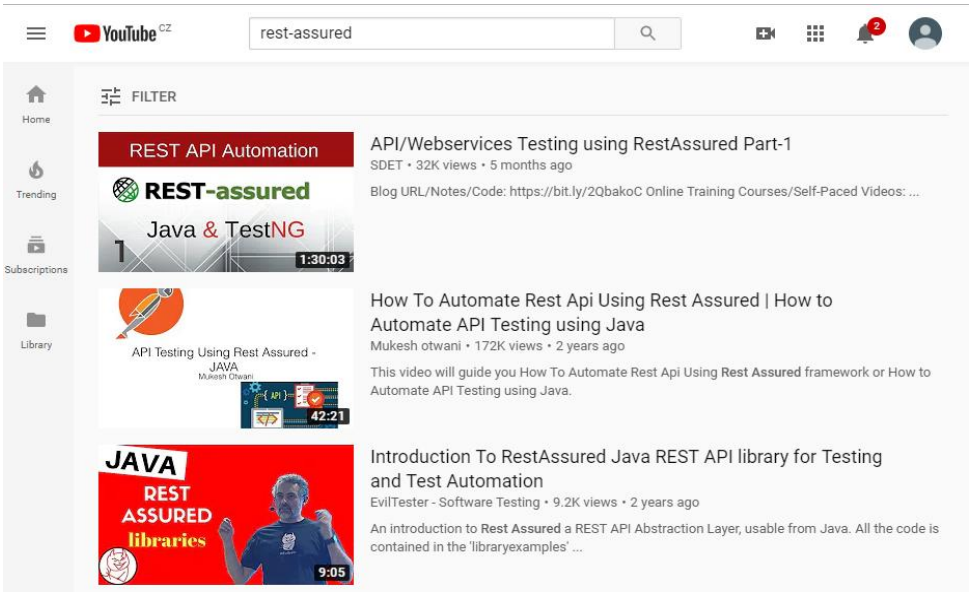


Figure 23 YouTube search results using keyword "rest-assured" (source: https://www.youtube.com/results?search_query=rest-assued)

Described earlier facts of the usage and popularity of REST-assured on the Internet lead to the conclusion that the criterion is fully satisfied.

CR_002 Initial project setup time

The initial project setup was started by the author from scratch with creating a new IntelliJ Java project using the Gradle system. After creating the project REST-assured dependency was installed as well as JUnit versions 4 and 5 – viz. *Figure 24*.

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.4.2'
    testCompile group: 'io.rest-assured', name: 'rest-assured', version: '4.1.2'
}
```

Figure 24 Installed dependencies in Initial project setup with REST-assured (source: author)

After installing the project author started with BaseTest class and first Get GraphQL Schema test implementation. Although the REST-assured library has detailed Wiki pages in GitHub repository official documentation does not provide information about using the library for sending GraphQL requests but one open issue from 27. December 2017 related found [91]. As a result of further research handled several discussions on different Internet portals [87] [92] related to the GraphQL API testing with REST-assured were found. According to the resources found [92], the BaseTest class was implemented in the way shown in *Code 9*.

```
class BaseTest {

    public ObjectNode variablesMap = new ObjectMapper().createObjectNode();

    Response sendRequest(String fileName) {
        // Get string from grphql file, use \A delimiter for scanner to match the
beginning of the String
```

```
        String queryAsString = new Scanner(TestClass.class
                .getResourceAsStream("/graphql/" + fileName + ".graphql"), "UTF-8")
                .useDelimiter("\\A").next();

        // Save query and variables as JSON object
        JSONObject payload = new JSONObject();
        payload.put("query", queryAsString);
        payload.put("variables", variablesMap);

        // Send query with POST method to given URL and verify status code 200
        return given()
                .contentType("application/json")
                .body(payload.toString())
                .post("http://localhost:8080/graphql")
                .then()
                .statusCode(200)
                .extract().response();
    }
}
```

Code 9 BaseTest class implementation with REST-assured (source: author)

Following actions were taken during BaseTest implementation:

1. Variables map of type ObjectNode is created in BaseTest class
2. SendRequest method with String fileName argument is added
3. Resource file is found and saved as String by path with added value of fileName argument
4. Status code of response is verified and if code 200 returned from server response object is returned

As one of the company X requirements was the usage of resource files for saving queries and mutation, the Java Util Scanner class was used as an example. This implementation can be changed to any other solution like Files, FileInputStream or BufferedReader [93]. Saved request as String is saved into payload JSONObject as well as a variables map [92]. Request is sent using the REST-assured library by defining content-type header, body message with payload as a string and POST method to the local GraphQL server. Status code of the received response is checked and if status code received is 200 value response is returned otherwise REST-assured returns an error.

Described above implementation of BaseTest and sendRequest method required additional libraries installation – viz. *Figure 25*.

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'
    testCompile group: 'org.junit.jupiter', name: 'junit-jupiter-api', version: '5.4.2'
    testCompile group: 'io.rest-assured', name: 'rest-assured', version: '4.1.2'
    compile group: 'com.fasterxml.jackson.core', name: 'jackson-databind', version: '2.9.9.3'
    compile group: 'org.json', name: 'json', version: '20180813'
}
```

Figure 25 Extended installed dependencies in Initial project setup with REST-assured (source: author)

Test Get GraphQL Schema is implemented in a way shown in *Code 10*.

```
public class GetServerSchema extends BaseTest {

    @Test
    public void getServerSchema() {
        // Add variables to payload JSON object
        Response response = sendRequest("getServerSchema");
    }
}
```

Code 10 GetServerSchema test case implementation with REST-assured (source: author)

GetSreverSchema class is extended from BaseTest and can use all its functionality like the sendRequest method and variables map. Method sendRequest with file name argument is called from BaseTest class. No additional assertions were added as status code is controlled inside the sendRequest method.

Initial project setup implementation with BaseTest class took in total 95 minutes.

CR_003 Usage of Gradle

During *CR_002 Initial project setup* REST-assured library was installed in the project using the Gradle system, which means that the solution fully satisfies the criterion.

CR_004 Compatibility with other Java frameworks and libraries

In *CR_002 Initial project setup* JUnit versions 4.12 and 5.4.2 were installed in the project using the Gradle system. Criterion is fully satisfied by the REST-assured library.

CR_005 User authorization

REST-assured library allows adding headers while sending requests through HTTP protocol. In the sendRequest method, authorization header can be added to a request in a way shown in *Code 11*.

```
given()
        .contentType("application/json")
        .header("Authorization", "Basic dXNlcjpwYXNzd3ByZA==")
        .body(payload.toString())
        .post("http://localhost:8080/graphql")
        .then()
        .statusCode(200)
        .extract().response();
```

Code 11 REST-assured request builder example using authorization header (source: author)

As the authorization header can be added to the REST-assured builder criterion is then fully satisfied.

CR_006 CRUD tests implementation time

CRUD test suite was implemented according to criterion description. Final suite code with 4 TCs is attached in *Annex E: CRUD test suite implementation*. Each TC in a suite uses requests from *Annex B: GraphQL requests used in solutions evaluation* and were automated according to test steps described in *Annex C: CRUD Test Cases*. *Code 12* is giving an example of the Edit existing Data automated test.

```
@Test
public void editExistingData() {
    variablesMap.put("typeValue", "character").put("exampleValue", "A");

    Response response = sendRequest("createDataWIthFragment");

    // Get new data instance id
    String id = response.path("data.createData.id");

    // Change example value and save into variables
    variablesMap.put("id", id).put("exampleValue", "B");

    response = sendRequest("editData");

    // Get updated data instance
    response = sendRequest("getDataWithFragment");

    assertEquals("B", response.jsonPath().getString("data.getData.example"));

}
```

Code 12 Edit existing Data test implementation with REST-assured (source: author)

In Edit existing Data test first Data fields values were added to the map after what the request for creating new Data instance is sent to GraphQL API. From the received response ID value of created Data is saved into a String variable and added to variables map as it will be used in the following requests for working with Data instance.

Data instance with defined ID is edited by sending editData mutation after what Data instance fields are got from response to the getDataWIthFragment query. REST-assured returns Response object on which jsonPath method ca be applied. JsonPath method in REST-assured allows searching for JSON nodes by the path and returning the value in the format given by the user.

In the last step of the editExistingData test by defined Json path String value for Data Example field is compared with the changed Example field value.

The total time for CRUD suite implementation is 25 minutes.

CR_007 Reusable resources

All used queries and mutations for example project with REST-assured library can be saved to resources. Specified files are searched in the resource directory by the name of the file given as an argument in the sendRequest method. CR_007 is than considered as fully satisfied.

CR_008 Ability to use GraphQL features

REST-assured hasn't got implemented functionality for using GraphQL features like variables or fragments. However, the variables map can be added to the request body payload. For using fragments in requests additional methods implementation is required. The solution just partly satisfies the criterion *CR_008*.

CR_009 Test parametrization

As REST-assured was already proved to be compatible with other Java frameworks as JUnit, all TCs can also be parametrised using the same mentioned earlier framework. The criterion is fully satisfied.

CR_010 The ease of understanding solution methods

Target group formed of QA engineers from company X received a questionnaire with BaseTest class and Edit existing Data test implementation for evaluating the ease of understanding of the solution method. Questionnaire template can be found in *Annex D: MCDA questionnaires*. *Figure 26* represent the results received for the questionnaire.

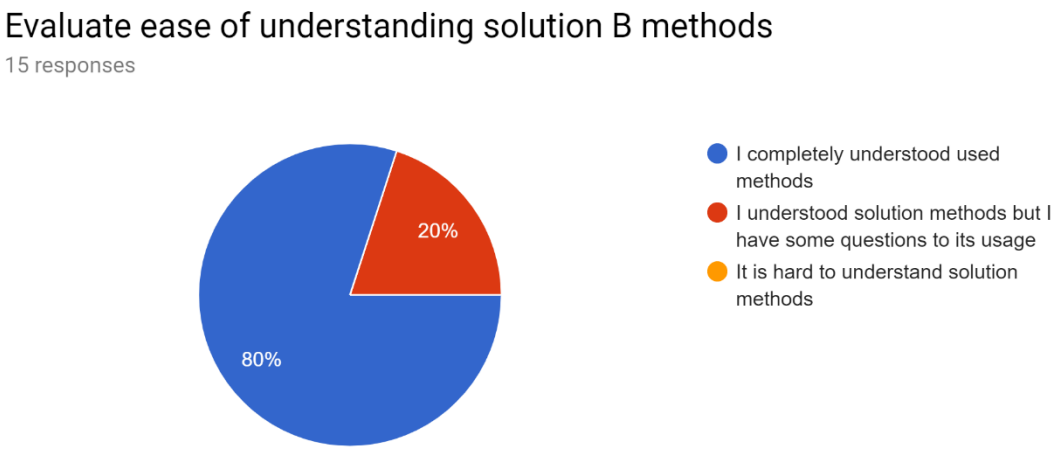


Figure 26 The ease of understanding solution B methods - questionnaire results (source: author)

80% of the target group evaluated the *CR_010* as "*I completely understood used methods*", 20% as "*I understood solution methods but I have some questions to its usage*". This means that criterion receive average value 3.

CR_011 Test reports

REST-assured does not provide reporting options, but test reports can be generated using other libraries and frameworks used in projects like JUnit, TestNG or by installing other report libraries from Maven repository. The solution fully satisfies the criterion.

CR_012 Test run time

Implemented the CRUD test suite run with Gradle on an empty GraphQL server. Test run time is visible in *Figure 27* that represents the time for each TC ran. Evaluation value taken for the criterion is tun time of the whole CRUD suite.

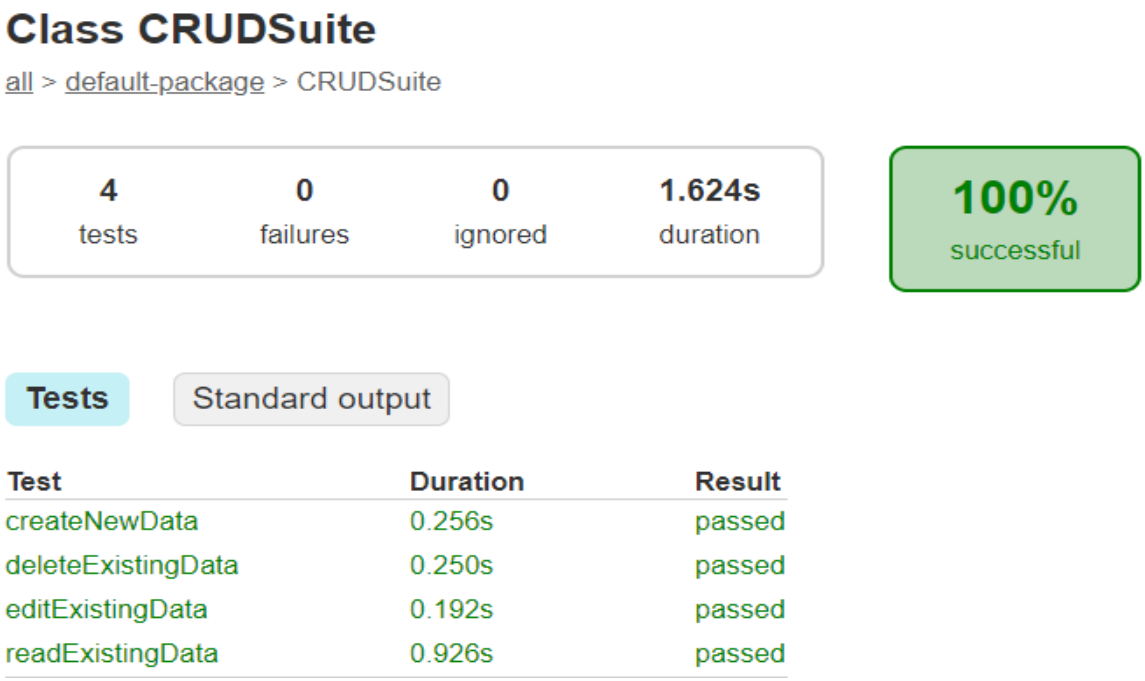


Figure 27 CRUD suite test run time using REST-assured (source: author)

Table 14 shows the time for three test suite runs with the calculated average value.

	First run	Second run	Third run	Average
Time	2.217s	2.420s	1.624s	2.087s

Table 14 Calculating of average test suite run time for REST-assured (source: author)

Final test suite run time for criteria evaluation is 2087ms.

CR_013 Error messages and logs informational content

For measuring of *CR_013* invalid GraphQL request is sent to the endpoint. Invalid GraphQL query was prepared in the same way as described in *6.5.1 Test GraphQL Java CR_013 Error messages and logs informational content*.

REST-assured allows enabling log information for the requests as displayed on *Code 13*. Enabled logging allows the output of sent request information as well as the response received from the server with the error message.

```
given()
        .log().all()
        .contentType("application/json")
        .header("Authorization", "Basic dXNlcjpwYXNzd3ByZA==")
        .body(payload.toString())
        .post("http://localhost:8080/graphql")
        .then()
        .log().all()
        .statusCode(200)
        .extract().response();
```

Code 13 REST-assured request with enabled logging (source: author)

The test was run with an invalid GraphQL query and the log enabled in the sendRequest method. *Figure 28* represents the error in the console output with the log saved closely before the test fails.

```
            {
                "message": "Validation error of type UnusedFragment: Unused fragment dataTypeFragment",
                "locations": [
                    {
                        "line": 7,
                        "column": 1,
                        "sourceName": null
                    }
                ],
                "description": "Unused fragment dataTypeFragment",
                "validationErrorType": "UnusedFragment",
                "queryPath": null,
                "errorType": "ValidationError",
                "path": null,
                "extensions": null
            }
        ]
    }

    expected: <B> but was: <null>
org.opentest4j.AssertionFailedError: expected: <B> but was: <null> <5 internal calls>
        at CRUDSuite.editExistingData(CRUDSuite.java:60) <43 internal calls>
        at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) <1 internal call>
        at java.lang.Thread.run(Thread.java:745)

    CRUDSuite > editExistingData STARTED
    CRUDSuite > editExistingData FAILED
        org.opentest4j.AssertionFailedError: expected: <B> but was: <null> <5 internal calls>
            at CRUDSuite.editExistingData(CRUDSuite.java:60)
    1 test completed, 1 failed
```

Figure 28 REST-assured error message example (source: author)

The log and error messages informational content was evaluated by the target group using the questionnaire template and results are attached in *Annex D: MCDA questionnaires*. REST-assured error messages and logs informational criterion was rated by 11 from the target group as "*I completely understood the error and can solve the occurred problem*" and by 4 as "*I partly understood what caused test failure*". The criterion evaluation value is then 3.

CR_014 Custom methods implementation

While measuring the second alternatives to the specified criteria the author defined only one additional method that should be implemented to cover all company X requirements.

REST-assured does not allow using GraphQL feature as a fragment. Additional method implementation is required if the company wants to use the concept of the mentioned feature in the test automation project.

Only one method needed to be additionally implemented, criterion is rated with value 2.

6.6 Alternatives evaluation

Chapter describes the process of the quantitative alternatives evaluation process using the WSA to choose the most beneficial test automation solution applicable in company X. *Table 15* presents summarised values assigned to defined criteria for each alternative with related weights. However, before calculating the total result the data should be firstly normalised.

Criteria	Test GraphQL Java	REST-assured	Weight
CR_001 Documentation, community support	2	3	0.13
CR_002 Initial project setup time	60	95	0.01
CR_003 Usage of Gradle	3	3	0.09
CR_004 Compatibility with other Java frameworks and libraries	3	3	0.11
CR_005 User authorization	3	3	0.03
CR_006 CRUD tests implementation time	35	25	0.03
CR_007 Reusable resources	3	3	0.11
CR_008 Ability to use GraphQL features	2	2	0.11
CR_009 Test parametrization	3	3	0.09
CR_010 The ease of understanding solution methods	2	3	0.07
CR_011 Test reports	3	3	0.07
CR_012 Tests run time	733	2087	0.01
CR_013 Error messages and logs informational content	1	3	0.07

CR_014 Custom methods implementation	1	2	0.05

Table 15 Alternatives evaluation criteria and weights (source: author)

Data normalisation

CR_002, CR_006, and *CR_012* are minimising criteria as was described before in *6.3.1 Criteria evaluation values*. These criteria should be as a first step of data normalisation transformed into maximising criteria. To do that from the highest of acquired values were subtracted the rest of the values, which helped to transform minimising criteria to maximising. *Table 16* represents the transformation of *CR_002*, 95 value is the highest received from which the other value was subtracted. Now for CR_002 Test GraphQL Java alternative get 35 and REST-assured 0.

Criteria	Test GraphQL Java	REST-assured
CR_002 Initial project setup time	35	0

Table 16 Minimising to maximising criteria transformation example (source: author)

To normalise data on each value of the table was applied the transformation formula displayed in *Figure 29*.

$$r_{ij} = \frac{Y_{ij} - D_j}{H_j - D_j}$$

Figure 29 Normalised criterion matrix formula (source: [89])

In transformation formula Y_{ij} stands for criterion value acquired for each alternative, D_j is the minimal value received for the criterion and H_j is the highest value. For values of *CR_002* represented in *Table 16 Minimising to maximising criteria transformation example*. On the example of CR_002, the application of transformation formula is displayed in *Table 17*. After calculating Test GraphQL Java received value 1 and REST-assured value 0 for criterion CR_002. For criteria that received the same values during the evaluation process, the value 0 is assigned.

Criteria	Test GraphQL Java	REST-assured
CR_002 Initial project setup time	(35-0)/(35-0)=1	(0-0)/(35-0)=0

Table 17 Application of transformation formula for normalising criterion matrix (source: author)

Selecting the best alternative

In *Table 18* is represented the normalised matrix with values for each criterion received as a result of data normalisation. Each criterion acquired value was multiplied by the weight value and after what all criterion values were summarised for each alternative to specify which of them reaches the maximum value (according to formula illustrated on *Figure 30*,

$$u(a_i) = \sum_{j=1}^{k} v_j \cdot r_{ij}$$

Figure 30 Formula for calculating alternatives performance value (source: [89])

where *Vj* is weight value for each criterion and R_{ij} is each alternative value from normalised matrix).

Criteria	Test GraphQL Java	REST-assured	Weight
CR_001 Documentation, community support	0	1	0.13
CR_002 Initial project setup time	1	0	0.01
CR_003 Usage of Gradle	0	0	0.09
CR_004 Compatibility with other Java frameworks and libraries	0	0	0.11
CR_005 User authorization	0	0	0.03
CR_006 CRUD tests implementation time	0	1	0.03
CR_007 Reusable resources	0	0	0.11
CR_008 Ability to use GraphQL features	0	0	0.11
CR_009 Test parametrization	0	0	0.09
CR_010 The ease of understanding solution methods	0	1	0.07
CR_011 Test reports	0	0	0.07
CR_012 Tests run time	1	0	0.01
CR_013 Error messages and logs informational content	0	1	0.07
CR_014 Custom methods implementation	0	1	0.05
Total:	0.02	0.35	-

Table 18 Criteria values evaluation (source: author)

According to the results received during MCDA analysis test automation solution using the REST-assured library is more beneficial for application in company X than solution using Test GraphQL Java library. After the evaluation of the received MCDA analysis result, the author of the thesis can recommend the REST-assured library for company X, even though it does not fully satisfy all their defined in the beginning requirements.

Further recommendations

Author prepared some further recommendations that can be useful for new test automation solution in company X as they can extend the functionality of REST-assured and help to overcome one of the biggest challenges occurred during API test automation process as maintenance.

Among these recommendations are:

- Implementation of fragment feature
  One of the given requirements from company X was the usage of GraphQL features like variables and fragments. The use of variables map can be easily implemented and used with the REST-assured library, however, there is no special function for using fragments in queries. In GraphQL fragments are defined in queries with *"..."* and the fragment name itself. For company X test automation solution would be useful to implement a method that will search through the query before sending it in a body request payload for fragment name and then add to a sent query String the fragment itself loaded from another file. Fragments can be saved in separate GraphQL files under a separate directory in resources. Fragment name in the query or mutation can be searched using the regular expression like "`\...(.*)`", that will help to select only the text from query after *"..."* – name of the fragment.
- Sending requests method
  Implemented sendRequest method always returns Response Object, however not every sent request needs to be saved as an Object because some operations are just auxiliary for achieving the main test purpose. Basic status code verification in sendRequest for those requests is enough. In this case, the author recommends the implementation of a void send request method that does not return any object nor saves the response. This method will only send defined requests, check status code inside and fail the test if received status code from server differs from expected 200 one.

Conclusions

The main goal of the thesis was to compare existing GraphQL API test automation solutions to recommend the one that can be used on a real project in a company that experienced a transition from Representational State Transfer (REST) to GraphQL API implementation and searching for new test automation approach. The main goal was successfully achieved through fulfilling defined smaller sub-goals of the work.

In chapter *2 Application Programming Interface* author achieved sub-goal – define the role of the API in Web applications, by describing the role of APIs in client-server architecture style used for Web applications implementation. Apart from that, the author explains the use of HTTP transfer protocol in relation to API and defines existing solutions for APIs implementation. The author also introduces the REST architecture style for API, characterise it and describes existing limitations.

Defined limitations of the REST architecture style served as a reason for multiple companies to develop new approaches for API. In chapter *3 GraphQL* author introduces one of the existing solutions that evade REST architecture style limitations and along with that achieves another sub-goal of thesis – introduce the GraphQL technology.

In chapter *4 API testing* author concentrates on describing the principles of API testing, defining its affiliation to test levels and specifies the challenges that may occur during manual and automation process of API testing. This chapter fulfills the third defined sub-goal of thesis as – characterize the process of API testing.

With all gained knowledge during the fulfilment of previous sub-goals author achieves next defined smaller goal – describe the existing solutions for the GraphQL API test automation using Java. As an output and benefit of achieving the goal, the author provides an overview of features of three different GraphQL API test automation solutions using Java programming language. Goal output can be found in chapter *5 GraphQL API test automation using Java*.

The fulfilment of the last sub-goal – evaluate the application of the existing solutions on a real-life project, of thesis lead to achieving the main thesis goal. In chapter *6 Evaluation of GraphQL API solutions for concrete company* author using Multiple Criteria Decision Analysis compares different solutions to recommend for introduced in the chapter concrete company. With the gained from the company requirements author specifies the criteria for evaluation of found existing GraphQL API test automation solutions after what in subchapter *6.5 Measuring the alternatives* the process of measuring alternatives impact on given criteria is described in a detailed way. For selecting the most beneficial solution for test automation for the concrete company author applied the Weight Sum Approach to calculate the total value of each alternative.

With the gained result of applied analysis author recommended company to choose REST-assured as a new test automation solution because in comparison to other defined

alternatives REST-assured library is more beneficial and satisfies other requirements received from the QA community. With the recommendation for test automation solution the author also gave some further recommendations for solution improvements that will help in maintaining a big number of API tests through different teams in the company.

List of references

1. LAUDON, Kenneth C. and TRAVER, Carol Guercio. *E-commerce: business, technology, society*. Upper Saddle River : Pearson, 2019. ISBN 978-1292303178
2. NAHAI, Nathalie. *Webs of influence: the psychology of online persuasion: the secret strategies that make us click*. New York : Pearson Education, 2017. ISBN 978-1292134604
3. GARTNER_INC. Digital Commerce. *Gartner* [online]. [Accessed 10 October 2019]. Available from: https://blogs.gartner.com/it-glossary/digital-commerce/
4. MYERS, Glenford J., Corey SANDLER a Tom BADGETT. *The art of software testing*. 3rd ed. Hoboken, N.J.: John Wiley, c2012. ISBN 978-1118031964.
5. AXELROD, Arnon. *Complete guide to test automation: techniques, practices, and patterns for building and maintaining effective software projects*. Berkeley, California : Apress, 2018.ISBN 978-1484238318.
6. ISTQB Glossary. *ISTQB* [online]. [Accessed 10 October 2019]. Available from: https://glossary.istqb.org/
7. JACOBSON, Daniel, Dan WOODS a Gregory BRAIL. *APIs: a strategy guide*. Sebastopol, CA: O'Reilly, c2012. ISBN 978-1449308926.
8. SATERNOS, Casimir, St. Laurent, Simon ST. LAURENT, SIMON, Allyson MACDONALD a Rebecca DEMAREST. *Client-server web apps with JavaScript and Java*. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1449369330.
9. BUNA, Samer. *Learning graphql and relay*. Packt Publishing Limited, 2016. ISBN 978-1786465757
10. RICHARDSON, Alan. *Automating and testing a REST API: a case study in API testing using: Java, REST Assured, Postman, Tracks, cURL and HTTP proxies*. Great Britain: Compendium Developments Ltd, 2017. ISBN 978-0956733290
11. VARGAS, Daniela Meneses, BLANCO, Alison Fernandez, VIDAURRE, Andreina Cota, ALCOCER, Juan P. S., TORRES, Milton Mamani, BERGEL, Alexandre, DUCASSE, Stephane, n.d. *Deviation Testing: A Test Case Generation Technique for GraphQL APIs*. Available from: http://bergel.eu/MyPapers/Mene18a-GraphQL.pdf
12. GAYATHRI, S., MONISHA, M.. *Study on GRAPHQL and Automation Testing*. 2018 Available from: http://ijirt.org/Article?manuscript=145469
13. PODLIPSKÝ, Šimon. *Porovnání implementací REST a GraphQL API*. Prague, 2018. Master thesis. University of Economics, Faculty of Informatics and Statistics. Available from: https://insis.vse.cz/zp/portal_zp.pl?podrobnosti_zp=62368
14. CEDERLUND, Mattias. *Performance of frameworks for declarative data fetching : An evaluation of Falcor and Relay+GraphQL*. 2016. Master thesis. KTH, School of Information and Communication Technology, Computer and Information Sciences Available from: http://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1045900&dswid=-7362
15. EIZINGER, Thomas, *API Design in Distributed Systems: A Comparison between GraphQL and REST*. Vienna, 2017. Master thesis. University of Applied Sciences Technikum Wien, Degree Program Software Engineering. Available from: https://eizinger.io/assets/Master-Thesis.pdf

16. A query language for APIs. *GraphQL* [online]. [Accessed 28 October 2019]. Available from: https://graphql.org/learn
17. Welcome to the API Economy - Smarter With Gartner. *Gartner* [online] [Accessed 28 October 2019]. Available from: https://www.gartner.com/smarterwithgartner/welcome-to-the-api-economy/
18. Application Programming Interface (api). *Gartner* [online] [Accessed 28 October 2019]. Available from: https://www.gartner.com/en/information-technology/glossary/application-programming-interface-api
19. What is Web Application (Web Apps) and its Benefits. *Search Software Quality, TechTarget* [online] [Accessed 28 October 2019]. Available from: https://searchsoftwarequality.techtarget.com/definition/Web-application-Web-app
20. SATERNOS, Casimir, St. Laurent, Simon ST. LAURENT, SIMON, Allyson MACDONALD a Rebecca DEMAREST. Client-server web apps with JavaScript and Java. Sebastopol, CA: O'Reilly Media, 2014. ISBN 978-1449369330.
21. What is Client/Server Architecture? - Definition from Techopedia. *Technopedia* [online] [Accessed 28 October 2019]. Available from: https://www.techopedia.com/definition/438/clientserver-architecture
22. Introduction to web APIs - Learn web development | MDN. *Mozilla and individual contributors* [online] [Accessed 28 October 2019]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction
23. First vs. Third-Party APIs: What You Need to Know. *HubSpot, Inc.* [online] [Accessed 28 October 2019]. Available from: https://blog.hubspot.com/marketing/third-party-api
24. EISING, Perry. What exactly IS an API? *Medium* [online]. 7 December 2017. [Accessed 28 October 2019]. Available from: https://medium.com/@perrysetgo/what-exactly-is-an-api-69f36968a41f
25. GARTNER_INC. Magic Quadrant for Full Life Cycle API Management. *Gartner* [online] [Accessed 28 October 2019]. Available from: https://www.gartner.com/en/documents/3873383
26. GARTNER_INC. Full Life Cycle API Management Software Reviews. *Gartner* [online] [Accessed 28 October 2019]. Available from: https://www.gartner.com/reviews/market/full-life-cycle-api-management
27. MULESOFT VIDEOS. What is an API? *YouTube* [online]. [Accessed 28 October 2019]. Available from: https://www.youtube.com/watch?v=s7wmiS2mSXY
28. What is an API Endpoint? | SmartBear Software. *SmartBear* [online]. [Accessed 28 October 2019]. Available from: https://smartbear.com/learn/performance-monitoring/api-endpoints/
29. What is an API Endpoint?: API Endpoint Definition: RapidAPI. *Last Call - RapidAPI Blog* [online]. [Accessed 28 October 2019]. Available from: https://rapidapi.com/blog/api-glossary/endpoint/
30. Web Services Architecture. *W3C* [online]. [Accessed 28 October 2019]. Available from: https://www.w3.org/TR/ws-arch/#whatis
31. API types. *ffeathers* [online]. [Accessed 28 October 2019]. Available from: https://ffeathers.wordpress.com/2014/02/16/api-types/

32. SOAP Web Services Tutorial: Simple Object Access Protocol EXAMPLE. *Guru99* [online]. [Accessed 28 October 2019]. Available from: https://www.guru99.com/soap-simple-object-access-protocol.html
33. ROUSE, Margaret, MATTURRO, Bree, ROUSE, Margaret and ROUSE, Margaret. What is Remote Procedure Call (RPC)? - Definition from WhatIs.com. *SearchAppArchitecture* [online]. [Accessed 28 October 2019]. Available from: https://searchapparchitecture.techtarget.com/definition/Remote-Procedure-Call-RPC
34. HTTP Tutorial. *Tutorialspoint* [online]. [Accessed 28 October 2019]. Available from: https://www.tutorialspoint.com/http/index.htm
35. GAITATZIS, Tony. *Learn REST APIs: Your guide to how to find, learn, and connect to the REST APIs that powers the Internet of Things revolution*. BackupBrain Press, 2019. ISBN 978-1999381769
36. RFC 822: Standard for the Format of Arpa Internet Text Messages. *W3C* [online]. [Accessed 28 October 2019]. Available from: https://www.w3.org/Protocols/rfc822/ (accessed 10.28.19).
37. What is HTTP TRACE? *CGISecurity.com* [online]. [Accessed 28 October 2019]. Available from: https://www.cgisecurity.com/questions/httptrace.shtml
38. HIGGINBOTHAM, James. The power of HTTP for REST APIs - Part 1. *Tyk API Gateway and API Management* [online]. 15 May 2018. [Accessed 28 October 2019]. Available from: https://tyk.io/power-http-rest-apis-part-1/
39. History of REST APIs. *Mobapi* [online]. 23 April 2018. [Accessed 28 October 2019]. Available from: https://www.mobapi.com/history-of-rest-apis/ (accessed 10.28.19).
40. README. The History of REST APIs. *ReadMe Blog* [online]. 22 February 2017. [Accessed 28 October 2019]. Available from: https://blog.readme.io/the-history-of-rest-apis/
41. Architectural Styles and the Design of Network-based Software Architectures. *Information and Computer Science, University of California, Irvine* [online]. [Accessed 28 October 2019]. Available from: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
42. REST vs. GraphQL: A Critical Review. *Good API Consulting* [online]. [Accessed 28 October 2019]. Available from: https://goodapi.co/blog/rest-vs-graphql
43. ESCHWEILER, Sebastian. REST vs. GraphQL. Medium [online]. [Accessed 28 October 2019]. Available from: https://medium.com/codingthesmartway-com-blog/rest-vs-graphql-418eac2e3083
44. What is a REST API round trip? *Stack Overflow* [online]. [Accessed 28 October 2019]. Available from: https://stackoverflow.com/questions/49093754/what-is-a-rest-api-round-trip
45. GIROUX, Marc-André Where we Come From: An Honest Introduction to GraphQL. *Medium* [online]. 3 March 2019. [Accessed 13 October 2019]. Available from: https://medium.com/@__xuorig__/where-we-come-from-an-honest-introduction-to-graphql-4a2ef6124488
46. GraphQL vs REST - A comparison. *GraphQL community* [online]. [Accessed 28 October 2019]. Available from: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/

47. US20130318154A1 - Api platform that includes server-executed client-based code. *Google Patents* [online]. [Accessed 28 October 2019]. Available from: https://patents.google.com/patent/US20130318154
48. NETFLIX TECHNOLOGY BLOG. Embracing the Differences : Inside the Netflix API Redesign. *Medium* [online]. 18 April 2017. [Accessed 28 October 2019]. Available from: https://medium.com/netflix-techblog/embracing-the-differences-inside-the-netflix-api-redesign-15fd8b3dc49d
49. KENTON, Will. SEC Form S-1. Investopedia [online]. [Accessed 23 October 2019]. Available from: https://www.investopedia.com/terms/s/sec-form-s-1.asp
50. Registration Statement on Form S-1. *U.S. Securities and Exchange Commission* [online]. [Accessed 28 October 2019]. Available from: https://www.sec.gov/Archives/edgar/data/1326801/000119312512034517/d287954ds1.htm#toc287954_2
51. BYRON, Lee. GraphQL: A data query language. *Facebook Engineering* [online]. 26 June 2018. [Accessed 28 October 2019]. Available from: https://engineering.fb.com/core-data/graphql-a-data-query-language/
52. CLARK, Brenda. What is GraphQL: History, Components, and Ecosystem. Medium [online]. 22 August 2019. [Accessed 28 October 2019]. Available from: https://levelup.gitconnected.com/what-is-graphql-87fc7687b042
53. SCOTT, James. Interview With GraphQL Co-Creator Lee Byron: Nordic APIs |. *Nordic APIs* [online]. 19 September 2018. [Accessed 28 October 2019]. Available from: https://nordicapis.com/interview-with-graphql-co-creator-lee-byron/
54. What is GraphQL? | Featuring GraphQL co-creator Dan Schafer - YouTube [Accessed 28 October 2019]. Available from: https://www.youtube.com/watch?v=mRgvbtNuCZY
55. STRANGE LOOP. “GraphQL: Designing a Data Language” by Lee Byron. *YouTube* [online] [Accessed 28 October 2019]. Available from: https://www.youtube.com/watch?v=Oh5oC98ztvI
56. BFF @ SoundCloud. *ThoughtWorks* [online]. 7 December 2015. [Accessed 28 October 2019]. Available from: https://www.thoughtworks.com/insights/blog/bff-soundcloud
57. KIMOKOTI, BRIAN. *BEGINNING GRAPHQL: fetch data faster and more efficiently whilst improving the overall performance of your web application*. PACKT Publishing Limited, 2018. ISBN 978-1789610543
58. PRASAD, Swathi. GraphQL Java Example for Beginners [Spring Boot] - DZone Integration. *dzone.com* [online]. 2 October 2019. [Accessed 28 October 2019]. Available from: https://dzone.com/articles/a-beginners-guide-to-graphql-with-spring-boot
59. GraphQL Playground. *Apollo GraphQL Docs* [online]. [Accessed 28 October 2019]. Available from: https://www.apollographql.com/docs/apollo-server/testing/graphql-playground/
60. WIERUCH, Robin. The road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React.js and Node.js.. 2019 ISBN 978-1730853937
61. SOFTWARE TESTING GENIUS. Client Server Testing. *Software Testing Genius* [online]. 8 September 2018. [Accessed 28 October 2019]. Available from: https://www.softwaretestinggenius.com/client-server-testing/
62. Frontend Testing Vs. Backend Testing: What's the Difference? *Guru99* [online]. [Accessed 28 October 2019]. Available from: https://www.guru99.com/frontend-testing-vs-backend-testing.html

63. Foundation Level Syllabus. *ISTQB® International Software Testing Qualifications Board* [online]. [Accessed 28 October 2019]. Available from: https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html
64. SHARMA, Lakshay. What is Integration Testing and Levels of Integration Testing? *TOOLSQA* [online]. 3 September 2019. [Accessed 28 October 2019]. Available from: https://www.toolsqa.com/software-testing/istqb/integration-testing/
65. NANDITA, DIVYA, GODI, Vijay, MINA, OMAR, DIANA, SAMPADA, LINDA, GUPTA, Rohit, SEELA, Swati, RUPALI and MARKY. How to Perform Backend Testing. *Software Testing Help* [online]. [Accessed 28 October 2019]. Available from: https://www.softwaretestinghelp.com/how-to-perform-backend-testing/
66. AMMANN, Paul and OFFUTT, Jeff. *Introduction to software testing*. Cambridge: Cambridge University Press, 2011. ISBN 978-0521880381
67. What Is API Testing? *SmartBear Solutions* [online]. [Accessed 28 October 2019]. Available from: https://smartbear.com/solutions/api-testing/
68. ISO9126 - Software Quality Characteristics. *sqa.net* [online]. [Accessed 28 October 2019]. Available from: http://www.sqa.net/iso9126.html
69. API Testing - What Should You Really Test? *Resourcology* [online]. [Accessed 28 October 2019]. Available from: https://www.resourcology.com/blog/api-testing-what-should-you-really-test/
70. What is Interface Testing? Types & Example. *Guru99* [online]. [Accessed 28 October 2019]. Available from: https://www.guru99.com/interface-testing.html
71. Rate limiting. *Wikipedia* [online]. [Accessed 28 October 2019]. Available from: https://en.wikipedia.org/wiki/Rate_limiting
72. 7 HTTP methods every web developer should know and how to test them. *Assertible* [online]. [Accessed 28 October 2019]. Available from: https://assertible.com/blog/7-http-methods-every-web-developer-should-know-and-how-to-test-them#patch
73. API Testing Guide An automated approach to API testing transformation. *CA Technologies* [online]. [Accessed 28 October 2019]. Available from: https://docs.broadcom.com/docs/api-testing-guide
74. Key Challenges of API Testing. *Resourcology* [online]. [Accessed 28 October 2019]. Available from: https://www.resourcology.com/blog/api-testing-challenges/
75. DUSTIN, Elfriede, Jeff RASHKA a John PAUL. *Automated software testing: introduction, management, and performance*. Reading, Mass.: Addison-Wesley, 1999. ISBN 978-0201432879.
76. RAFI, D.M., MOSES, K.R.K., PETERSEN, K., MÄNTYLÄ, M.V., *Proceedings of the 7th International Workshop on Automation of Software Test*. Piscataway, NJ : IEEE Press, 2012. ISBN 978-1-4673-1822-8
77. CARMICHAEL, Elise. API Testing Automation Tutorial. *QASymphony* [online]. 7 January 2019. [Accessed 28 October 2019]. Available from: https://www.qasymphony.com/blog/automated-api-testing-tutorial/
78. AGGARWAL, Dheeraj Kumar. Challenges behind automation of REST API Testing. *Medium* [online]. 27 August 2015. [Accessed 13 November 2019]. Available from: https://medium.com/@dheerajaggarwal/challenges-behind-automation-of-rest-api-testing-f12f2eb20687
79. Build software better, together. *GitHub* [online]. [Accessed 28 October 2019]. Available from: https://github.com

80. VIMALRAJ. Introducing Test GraphQL Java. *Vimal Selvam* [online]. 1 June 2019. [Accessed 28 October 2019]. Available from: http://www.vimalselvam.com/2019/06/02/introducing-test-graphql-java/
81. REST Assured. *REST Assured* [online]. [Accessed 28 October 2019]. Available from: http://rest-assured.io/
82. rest-assured/rest-assured. *GitHub* [online] [Accessed 28 October 2019]. Available from: https://github.com/rest-assured/rest-assured
83. ROZA, Ger. REST Assured Authentication. *Baeldung* [online]. 4 May 2019. [Accessed 28 October 2019]. Available from: https://www.baeldung.com/rest-assured-authentication
84. OAuth Community Site. *OAuth Community Site* [online]. [Accessed 28 October 2019]. Available from: https://oauth.net/
85. BAELDUNG. A Guide to REST-assured. *Baeldung* [online]. 9 August 2019. [Accessed 28 October 2019]. Available from: https://www.baeldung.com/rest-assured-tutorial
86. Rest API Automation With Rest Assured. *Tutorialspoint* [online]. [Accessed 28 October 2019]. Available from: https://www.tutorialspoint.com/rest_api_automation_with_rest_assured/index.asp
87. SWAPNIL.S, SWAPNIL.SSWAPNIL.S 8788 BRONZE BADGES, ALEXEY R.ALEXEY R. 8 and BERNHARD NEUDECKERBERNHARD NEUDECKER 1. How to test and automate APIs implemented in GraphQL. *Software Quality Assurance & Testing Stack Exchange* [online]. [Accessed 28 October 2019]. Available from: https://sqa.stackexchange.com/questions/36219/how-to-test-and-automate-apis-implemented-in-graphql
88. intuit/karate. *Github* [online]. [Accessed 28 October 2019]. Available from: https://intuit.github.io/karate/
89. KORVINY, Petr. Teoretické základy vícekriteriálního rozhodovaní. *Petr Korviny | Osobní stránky* [online]. [Accessed 28 October 2019]. Available from: https://korviny.cz/Korviny/soubory/teorie_mca.pdf
90. vimalrajselvam/test-graphql-java. *GitHub* [online]. [Accessed 28 October 2019]. Available from: https://github.com/vimalrajselvam/test-graphql-java/
91. REST-ASSURED. graphql testing · Issue #958 · rest-assured/rest-assured. *GitHub* [online]. [Accessed 28 October 2019]. Available from: https://github.com/rest-assured/rest-assured/issues/958
92. How to use graphql query in restassured api automation directly using java ? *premsvmm.blogspot.com* [online]. [Accessed 28 October 2019]. Available from: https://premsvmm.blogspot.com/2018/09/how-to-use-graphql-query-in-restassured.html
93. How do I create a Java string from the contents of a file? *Stack Overflow* [online]. [Accessed 28 October 2019]. Available from:https://stackoverflow.com/questions/326390/how-do-i-create-a-java-string-from-the-contents-of-a-file

Annexes

Annex A: GraphQL custom server implementation

Annex describes the process of custom GraphQL CRUD server implementation that was implemented according to *GraphQL Java Example for Beginners tutorial* [58]. Server was implemented using Java, GraphQL and Spring technologies. GraphQL schema contains the characteristics of server implementation with all types defined in it. GraphQL schema has Data, DeletResponse, Query and Mutation. Schema representation is displayed in *Code 14*.

```
type Data {
          id: ID!,
          type: String,
          example: String
}

type DeleteResponse {
          status: String
}

type Query {
          getListOfData:[Data]
          getData(id: ID):Data
}

type Mutation {
          createData(type: String!, example: String!):Data
          editData(id: ID!, type: String!, example: String!):Data
          deleteData(id: ID!):DeleteResponse
}
```

Code 14 GraphQL server schema (source: author)

Data object and DeleteResponse object are created using Spring framework. Data object class implementation is displayed in *Code 15*.

```
@lombok.Data
@EqualsAndHashCode
@Entity
public class Data implements Serializable {

    @Id
    @Column(name = "ID", nullable = false)
    @GeneratedValue(strategy = GenerationType.AUTO)
```

```
    private int id;

    @Column(name = "type", nullable = false)
    private String type;

    @Column(name = "example", nullable = false)
    private String example;

}
```
Code 15 Data object class (source: author)

All Data objects can be saved to Data Repository – interface that extends JpaRepository Spring framework methods that allows to get all elements of the Repository.

Queries and mutations functionality are implemented in DataService class that also creates an instance of DataRepository into which all Data objects are saved. Example code of query and mutation is visible in *Code 16*.

```
    @Transactional(readOnly = true)
    public Optional<Data> getData(final int id) {
        return this.dataRepository.findById(id);
    }

    @Transactional
    public DeleteResponse deleteData(final int id) {
        this.dataRepository.findById(id).ifPresent(data -> {
            dataRepository.deleteById(id);
            deleteResponse.setStatus(true);
        });
        return this.deleteResponse;
    }
```
Code 16 Query and mutation implementation example (source: author)

To enable working with queries and mutations GraphQLQueryResolvers and GraphQLMutationResolvers are also implemented. Server is working with H2 embedded database that means that all created Data will be saved only when the GraphQL server is running. When GraphQL server is stopped all created data will be lost. GraphQL server can be run with Spring Boot and its instance is running on http://localhost:8080/graphql with GraphiQL IDE accessed on http://localhost:8080/graphiql.

Annex B: GraphQL requests used in solutions evaluation

This annex contains GraphQL query examples used in tests for checking custom GraphQL server functionality.

Get server schema query

Query is asking for system field "__schema" and returning all object types and their fields names in response.

```
query getServerSchema {
  __schema{
    types{
      name
      fields{
        name
      }
    }
  }
}
```

Code 17 "getServerSchema" query (source: author)

Get list of from server query

Query returns list of all Data objects from the server with fields defined in dataTypeFragment.

```
query getListOfData {
  getListOfData {
    ...dataTypeFragment
  }
}
```

Code 18 "getListOfData" query (source: author)

Get data from server query

Query returns Data instance characteristics with specified ID in the variables map.

```
query getData($id: ID!) {
  getData(id: $id) {
    id
    type
    example
  }
}
```

Code 19 "getData" query (source: author)

Create new data instance mutation

Mutation is used for creating new Data object instance with given by user Type and Example values. Mutation saves new Data object instance with unique ID to the DataRepository.

```
mutation createData($typeValue: String!, $exampleValue: String!) {
  createData(type: $typeValue, example: $exampleValue) {
```

```
    ...dataTypeFragment
  }
}
```
Code 20 "createData" mutation (source: author)

Data type fragment

DataTypeFragment contains fields of Data object, like ID, Type and Example.

```
fragment dataTypeFragment on Data {
  id
  type
  example
}
```
Code 21 "dataTypeFragment" fragment (source: author)

Edit existing data instance mutation

EditData mutation allows changing fields of Data object that is saved in DataRepository with the given ID.

```
mutation editData($id: ID!, $typeValue: String!, $exampleValue: String!) {
  editData(id: $id, type: $typeValue, example: $exampleValue) {
    id
    type
    example
  }
}
```
Code 22 "editData" mutation (source: author)

Delete instance mutation

Mutation deleteData allows removing Data instance with the specified ID. When Data instance is found in the DataRepository and deleted status field of Boolean type with true value is returned. When Data object with ID cannot be found in DataRepository status false is returned.

```
mutation deleteData($id: ID!) {
  deleteData(id: $id) {
    status
  }
}
```
Code 23 "deleteData" mutation (source: author)

Annex C: CRUD Test Cases

Annex contains TCs from CRUD test suite. Each Test Case has a name, description, steps description and expected result value. For all TCs one common precondition is set:

GraphQL server is running with structure defined in *Annex A: GraphQL custom server implementation*. Queries and mutations that can be used for tests are saved in *Annex B: GraphQL requests used in solutions evaluation*.

Create new Data			
Test description: Test is verifying the ability of creating new instance of Data object with defined Type and Example values.			
Step 1	Save test data that are used in query to variables. Set Type value as “character”, Example value as “A”	**Expected result**	Test variables are saved into variables map
Step 2	Send attached mutation for creating new Data on http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 3	Get new Data ID from previous request response	**Expected result**	Data ID is saved into variables
Step 4	Get list of data using prepared query from http://localhost:8080/graphql	**Expected result**	Verify list contains new Data object with ID, Type and Example values
Read existing Data information			
Test description: Test is verifying the ability of reading information(fields) of existing Data object.			
Step 1	Save test data that are used in query to variables. Set Type value as “character”, Example value as “A”. Send attached mutation for creating new Data on http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 2	Get new Data ID from previous request response	**Expected result**	Data ID is saved into variables
Step 3	Send attached query to get Data object information for ID value from previous TC from http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 4	Verify response contains Data object with Type value “character” and Example value “A”	**Expected result**	Data with given ID contains Type value “character” and Example value “A”

Edit existing Data			
Test description: Test is verifying the ability of editing fields of existing Data object.			
Step 1	Save test data that are used in query to variables. Set Type value as “character”, Example value as “A”.	**Expected result**	Request was sent successfully, status code 200 returned from the server

	Send attached mutation for creating new Data on http://localhost:8080/graphql		
Step 2	Get new Data ID from previous request response	**Expected result**	Data ID is saved into variables
Step 3	Save test data that are used in query to variables. Set Example value as "B"	**Expected result**	Test variables are updated
Step 4	Send attached mutation for editing Data object with specified ID on http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 5	Send attached query to get Data object information for ID from http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 6	Verify response contains Data object with Type value "character" and Example value "B"	**Expected result**	Data with given ID contains Type value "character" and Example value "B"
Delete existing Data			
Test Description: Test is verifying the ability of deleting existing Data object.			
Step 1	Save test data that are used in query to variables. Set Type value as "character", Example value as "A". Send attached mutation for creating new Data on http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 2	Get new Data ID from previous request response	**Expected result**	Data ID is saved into variables
Step 3	Send attached mutation for deleting Data object with specified ID on http://localhost:8080/graphql	**Expected result**	Request was sent successfully, status code 200 returned from the server
Step 4	Verify response contains field status which returns true value for deletion operation	**Expected result**	Response contains status field with true value set
Step 5	Get list of data using prepared query from http://localhost:8080/graphql	**Expected result**	Verify list doesn't contain Data ID

Table 19 CRUD suite Test Cases (source: author)

Annex D: MCDA questionnaires

Annex represents questionnaires templates created for master thesis MCDA analysis with Google Forms.

The ease of understanding solution methods

Solution evaluation

* Required

The ease of understanding solution methods

Look at the pictures below with test classes implementation using different libraries and rate the ease of understanding their methods.

Solution A - Base Test class

```
class BaseTest {

    ObjectNode variablesMap = new ObjectMapper().createObjectNode();

    Response sendRequest(String fileName) {
        // Get string from grphql file, use \A delimiter for scanner to match the beginning of the String
        String queryAsString = new Scanner(TestClass.class
                .getResourceAsStream( name: "/graphql/" + fileName + ".graphql"), charsetName: "UTF-8")
                .useDelimiter("\\A").next();

        // Save query and variables as JSON object
        JSONObject payload = new JSONObject();
        payload.put("query", queryAsString);
        payload.put("variables", variablesMap);

        // Send query with POST method to given URL and verify status code 200
        return given() RequestSpecification
                .log().all() RequestSpecification
                .contentType("application/json") RequestSpecification
                .header( headerName: "Authorization", headerValue: "Basic dXNlcjpwYXNzd3ByZA==") RequestSpecification
                .body(payload.toString()) RequestSpecification
                .post( path: "http://localhost:8080/graphql") Response
                .then() ValidatableResponse
                .log().all() ValidatableResponse
                .statusCode(200) ValidatableResponse
                .extract().response();
    }
}
```

Solution A - Edit existing Data test

```
@Test
public void editExistingData() {
    variablesMap.put( fieldName: "typeValue", v: "character").put( fieldName: "exampleValue", v: "A");

    Response response = sendRequest( fileName: "createDataWIthFragment");

    // Get new data instance id
    String id = response.path( path: "data.createData.id");
    variablesMap.put( fieldName: "id", id).put( fieldName: "exampleValue", v: "B");

    response = sendRequest( fileName: "editData");

    // Get updated data instance
    response = sendRequest( fileName: "getDataWithFragment");

    assertEquals( expected: "B", response.jsonPath().getString( path: "data.getData.example"));

}
```

Evaluate ease of understanding solution A methods *

- ◯ I completely understood used methods
- ◯ I understood solution methods but I have some questions to its usage
- ◯ It is hard to understand solution methods

Figure 31 The ease of understanding solution methods questionnaire template (source: author)

Error messages and logs informational content

Solution evaluation

* Required

Error messages and logs informational content

Look at the pictures below with logs and error messages written in the console output by two different solutions. Tests were run using different approaches and failed on the same mistake. Evaluate the informational content of the messages received.

Solution A

```
        {
            "message": "Validation error of type UnusedFragment: Unused fragment dataTypeFragment",
            "locations": [
                {
                    "line": 7,
                    "column": 1,
                    "sourceName": null
                }
            ],
            "description": "Unused fragment dataTypeFragment",
            "validationErrorType": "UnusedFragment",
            "queryPath": null,
            "errorType": "ValidationError",
            "path": null,
            "extensions": null
        }
    ]
}
```

```
expected: <8> but was: <null>
org.opentest4j.AssertionFailedError: expected: <8> but was: <null> <5 internal calls>
    at CRUDSuite.editExistingData(CRUDSuite.java:60) <43 internal calls>
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) <1 internal call>
    at java.lang.Thread.run(Thread.java:745)

CRUDSuite > editExistingData STARTED
CRUDSuite > editExistingData FAILED
    org.opentest4j.AssertionFailedError: expected: <8> but was: <null> <5 internal calls>
        at CRUDSuite.editExistingData(CRUDSuite.java:60)
1 test completed, 1 failed
```

Solution B

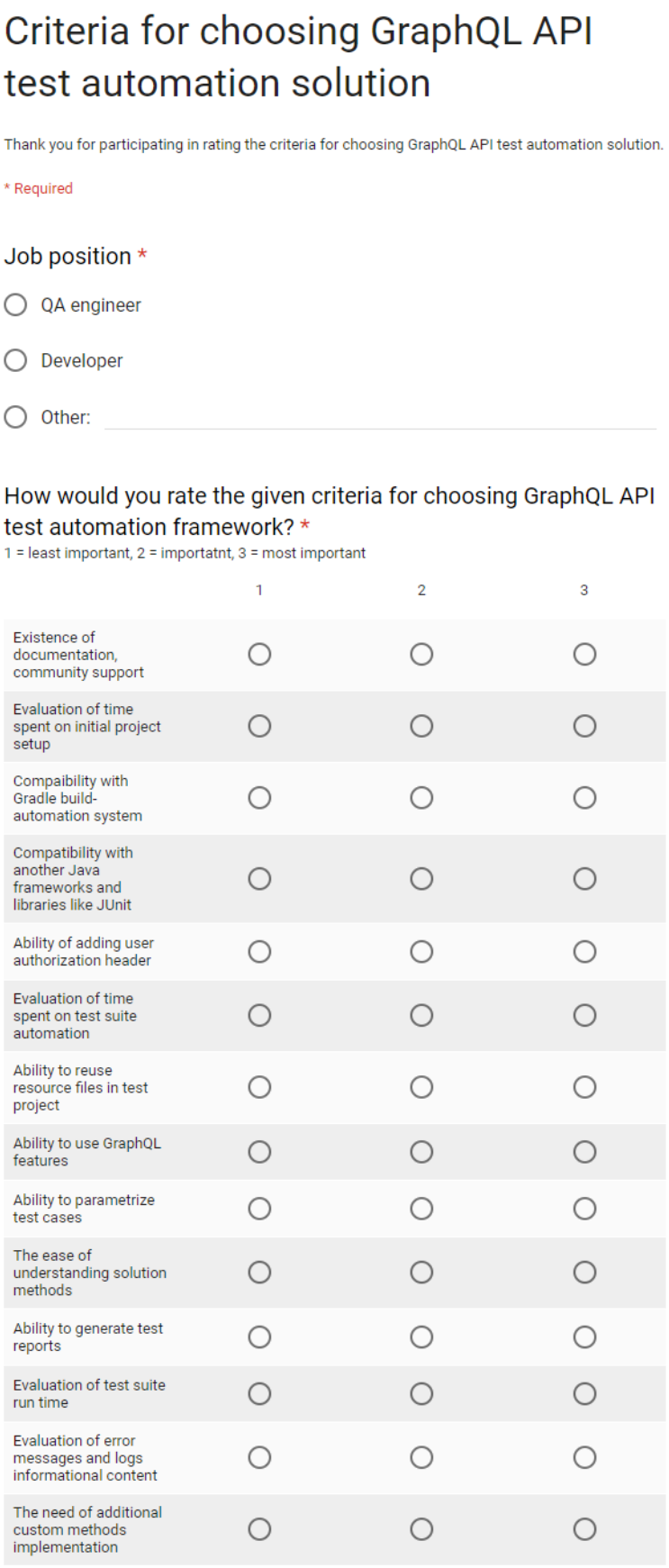
```
Testing started at 13:58 ...
> Task :cleanTest
> Task :compileJava NO-SOURCE
> Task :processResources NO-SOURCE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources
> Task :testClasses
> Task :test

java.lang.NullPointerException
    at CRUDSuite.readData(CRUDSuite.java:63) <43 internal calls>
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1142)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:617) <1 internal call>
    at java.lang.Thread.run(Thread.java:745)

CRUDSuite > readData STARTED
CRUDSuite > readData FAILED
    java.lang.NullPointerException
        at CRUDSuite.readData(CRUDSuite.java:63)
1 test completed, 1 failed
```

Evaluate informational content of different solution's logs *

	Log gives no information about the test failure	I partly understood what caused test failure	I completely understood the error and can solve occured problem
Solution A	○	○	○
Solution B	○	○	○

Figure 32 Error messages and logs informational content questionnaire template (source: author)

Criteria for choosing GraphQL API test automation solution



Criteria for choosing GraphQL API test automation solution

Thank you for participating in rating the criteria for choosing GraphQL API test automation solution.

* Required

Job position *

- QA engineer
- Developer
- Other: _____

How would you rate the given criteria for choosing GraphQL API test automation framework? *

1 = least important, 2 = importatnt, 3 = most important

	1	2	3
Existence of documentation, community support			
Evaluation of time spent on initial project setup			
Compaibility with Gradle build-automation system			
Compatibility with another Java frameworks and libraries like JUnit			
Ability of adding user authorization header			
Evaluation of time spent on test suite automation			
Ability to reuse resource files in test project			
Ability to use GraphQL features			
Ability to parametrize test cases			
The ease of understanding solution methods			
Ability to generate test reports			
Evaluation of test suite run time			
Evaluation of error messages and logs informational content			
The need of additional custom methods implementation			

Figure 33 Criteria for choosing GraphQL API test automation solution (source: author)

Error messages and logs informational content questionnaire result

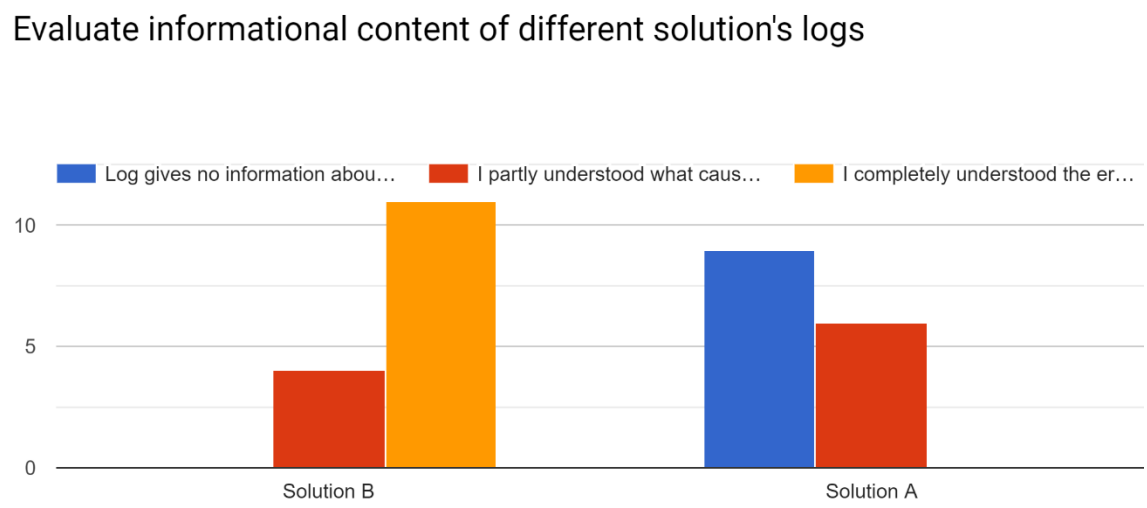


Figure 34 Error messages and logs informational content questionnaire result (source: author)

Annex E: CRUD test suite implementation

Annex contains source code of CRUD suite implementation using different solutions. Test suites represented in *Code 24* and *Code 25* can be edited and some of the methods can be grouped together and used in before and after tests methods that is not however related to any sub-goal of master thesis or described as a requirement for any of the criteria for framework evaluation process handled with MCDA.

```
public class CRUDSuite extends BaseTest {

    @Test
    public void createNewData() throws IOException {
        variables.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Save response as JsonNode
        String jsonData = response.body().string();
        JsonNode jsonNode = new ObjectMapper().readTree(jsonData);

        // Get new Data instance id
        String id = jsonNode.get("data").get("createData").get("id").asText();

        // Get Data from server
        response = sendRequest("getListOfDataWithFragment");
```

```
        // Verify ID is in ID list
        jsonData = response.body().string();
        jsonNode = new ObjectMapper().readTree(jsonData);


Assertions.assertTrue(jsonNode.get("data").get("getListOfData").findValuesAsText("id").con
tains(id));

Assertions.assertTrue(jsonNode.get("data").get("getListOfData").findValuesAsText("type").c
ontains("character"));

Assertions.assertTrue(jsonNode.get("data").get("getListOfData").findValuesAsText("example"
).contains("A"));

    }

    @Test
    public void readExistingData() throws IOException {
        variables.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Save response as JsonNode
        String jsonData = response.body().string();
        JsonNode jsonNode = new ObjectMapper().readTree(jsonData);

        // Get new data instance id
        String id = jsonNode.get("data").get("createData").get("id").asText();
        variables.put("id", id);

        response = sendRequest("getDataWithFragment");

        // Save response as JsonNode
        jsonData = response.body().string();
        jsonNode = new ObjectMapper().readTree(jsonData);

        assertEquals("character",
jsonNode.get("data").get("getData").get("type").asText());
        assertEquals("A", jsonNode.get("data").get("getData").get("example").asText());
    }

    @Test
    public void editExistinData() throws IOException {
        variables.put("typeValue", "character").put("exampleValue", "A");
```

```
        Response response = sendRequest("createDataWIthFragment");

        // Save response as JsonNode
        String jsonData = response.body().string();
        JsonNode jsonNode = new ObjectMapper().readTree(jsonData);

        // Get new data instance id
        String id = jsonNode.get("data").get("createData").get("id").asText();

        // Change example value and save into variables
        variables.put("id", id).put("exampleValue", "B");
        response = sendRequest("editData");

        // Get updated data instance
        response = sendRequest("getDataWithFragment");

        // Save response as JsonNode
        jsonData = response.body().string();
        jsonNode = new ObjectMapper().readTree(jsonData);

        assertEquals("B", jsonNode.get("data").get("getData").get("example").asText());

    }

    @Test
    public void deleteExistingData() throws IOException {
        variables.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Save response as JsonNode
        String jsonData = response.body().string();
        JsonNode jsonNode = new ObjectMapper().readTree(jsonData);

        // Get new data instance id
        String id = jsonNode.get("data").get("createData").get("id").asText();

        // Delete data instance
        variables.put("id", id);
        response = sendRequest("deleteData");

        jsonData = response.body().string();
        jsonNode = new ObjectMapper().readTree(jsonData);
```

```
        assertTrue(jsonNode.get("data").get("deleteData").get("status").asBoolean());

        // Verify instance id s not in the list of data
        response = sendRequest("getListOfDataWithFragment");

        // Verify new data instance id is in id list
        jsonData = response.body().string();
        jsonNode = new ObjectMapper().readTree(jsonData);


Assertions.assertFalse(jsonNode.get("data").get("getListOfData").findValuesAsText("id").co
ntains(id));

    }

}
```

Code 24 CRUDSuite implementation using Test GraphQL Java library (source: author)

```
public class CRUDSuite extends BaseTest {

    @Test
    public void createNewData() {
        variablesMap.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Get new data instance id
        String id = response.path("data.createData.id");

        response = sendRequest("getListOfDataWithFragment");


Assertions.assertTrue(response.jsonPath().getList("data.getListOfData.id").contains(id));

Assertions.assertTrue(response.jsonPath().getList("data.getListOfData.type").contains("cha
racter"));

Assertions.assertTrue(response.jsonPath().getList("data.getListOfData.example").contains("
A"));

    }

    @Test
    public void readExistingData() {
        variablesMap.put("typeValue", "character").put("exampleValue", "A");
```

```
        Response response = sendRequest("createDataWIthFragment");

        // Get new data instance id
        String id = response.path("data.createData.id");
        variablesMap.put("id", id);

        // Get updated data instance
        response = sendRequest("getDataWithFragment");

        assertEquals("character", response.jsonPath().getString("data.getData.type"));
        assertEquals("A", response.jsonPath().getString("data.getData.example"));
    }

    @Test
    public void editExistingData() {
        variablesMap.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Get new data instance id
        String id = response.path("data.createData.id");

        // Change example value and save into variables
        variablesMap.put("id", id).put("exampleValue", "B");
        response = sendRequest("editData");

        // Get updated data instance
        response = sendRequest("getDataWithFragment");

        assertEquals("B", response.jsonPath().getString("data.getData.example"));

    }

    @Test
    public void deleteExistingData() {
        variablesMap.put("typeValue", "character").put("exampleValue", "A");

        Response response = sendRequest("createDataWIthFragment");

        // Get new data instance id
        String id = response.path("data.createData.id");

        // Delete data instance
        variablesMap.put("id", id);
```

```
        response = sendRequest("deleteData");

        assertTrue(response.jsonPath().getBoolean("data.deleteData.status"));

        // Verify instance id s not in the list of data
        response = sendRequest("getListOfDataWithFragment");


Assertions.assertFalse(response.jsonPath().getList("data.getListOfData.id").contains(id));

    }
}
```
Code 25 CRUDSuite implementation using REST-assured library (source: author)