INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

Managing APIs evolution while maintaining System Interoperability

FÁBIO ALEXANDRE CARNEIRO ALVES

Junho de 2023

POLITÉCNICO DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Managing APIs evolution while maintaining System Interoperability

Fábio Alexandre Carneiro Alves

A dissertation submitted in partial fulfilment of the requirements for the degree of Master in Informatics Engineering, Specialisation Area of Software Engineering

Supervisor: Prof. Nuno Ferreira
Co-Supervisor: Eng. Miguel Veiga

Porto, June 30, 2023

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 30, 2023

Fábio Alves

"I knew exactly what to do.
But in a much more real sense, I had no idea what to do."
- Michael Scott

Abstract

Flutter Entertainment, a global sports betting company, has experienced a significant increase in online transactions due to the preference for online solutions and its recent acquisitions. Additionally, with the recurrent improvement of current features and the development of new ones, more components are being created and more updates are being done to existing components.

This growth in both the number of APIs and development features have made it challenging to manage the Web APIs' and their interactions. Additionally, the manual management of API contracts and the necessity to sync the APIs' changes between the clients and the APIs' owners have resulted in both increased development time and incidents. Moreover, due to the high volume of manual operations and the rapid pace of system evolution, the documentation quickly becomes outdated which exacerbates the problem. With all of these issues, the low system interoperability is becoming increasingly apparent which is the problem that this work aims to address.

The main objective of this work is to explore the issue of handling APIs evolution while maintaining system interoperability. For this, some topics must be addressed such as APIs documentation, schema registry and versioning. That said, work is carried out to explore architecture APIs patterns and good practices, system interoperability architectures and current interoperability solutions. Additionally, a proof of concept that satisfies the problem and its evaluation is presented.

After careful consideration of the available market options, the solution with the most benefits for the context of this case study is AsyncAPI, being this specification use as a proof of concept for two specific APIs of the tennis flow, namely TMT and IMG.

To handle versioning, it is used Semantic Versioning. Additionally, specific API lifecycle steps were established and are being used in the documentation of APIs. This documentation is generated automatically using the AsyncAPI specification and is kept up-to-date through the implementation of GitHub workflows. To manage the API contracts, a dedicated GitHub repository is utilized, containing all the necessary schemas referenced within the specification. Lastly, the Microcks tool is used as a solution to handle validation and API testing, enabling contract testing to be performed on the APIs and the implemented work.

In conclusion, the main objective of studying solutions to maintain system interoperability with evolving APIs have been achieved through the successful development of a comprehensive proof of concept. All identified topics are effectively addressed, and the evaluation of the user experience, both before and after the implementation, proves it. Overall, a notable improvement is observed in the system's interoperability and functionality.

Keywords: System Interoperability, Web APIs, Web Systems Evolution, Interoperability Solutions

Resumo

Flutter Entertainment, uma empresa internacional de apostas desportivas, tem registado um aumento significativo de transacções devido à crescente tendencia por soluções online e devido às recentes aquisições do grupo. Além disso, verifica-se um aumento de mudanças dos componentes integrantes do sistema e um crescimento do número de elementos do sistema associado à constante procura por melhores funcionalidades para os seus clientes.

Com este crescimento, tanto no número de APIs como nas atualizações de funcionalidades, começou a ser desafiante gerir os serviços Web e as interacções entre eles. Além disso, a gestão dos contratos das APIs faz-se de forma manual, existindo uma necessidade de sincronização entre os clientes e donos, aumentado o tempo de desenvolvimento e número de incidentes. Devido ao elevado volume de operações manuais e ao rápido ritmo de evolução do sistema, a documentação rapidamente se torna desactualizada. Ao se culminar todos os problemas, tem-se um baixo nível de interoperabilidade, sendo este o problema a resolver.

O principal objectivo deste trabalho é explorar e solucionar os problemas associados a evolução de APIs que dificultam a conservação da interoperabilidade do sistema. Assim sendo, alguns dos tópicos abordados estão relacionados com a documentação, registo dos contratos e versionamento de APIs. Posto isto, realiza-se um trabalho sobre padrões e boas práticas de arquiteturas de APIs, arquitecturas que promovam a interoperabilidade de sistemas e atuais soluções para a conservação de interoperabilidade. Adicionalmente, apresenta-se uma prova de conceito que responde ao problema e a respetiva avaliação da mesma.

Dentro das opções do mercado estudadas, aquela que traz mais benefícios para este caso de estudo é a AsyncAPI sendo implementada uma prova de conceito a duas APIs do fluxo de tênis, TMT e IMG, através do uso desta especificação.

Nesta prova de conceito usa-se como padrão de versionamento o *Semantic Versioning*. Adicionalmente, as etapas de vida da API estão identificadas na documentação. Esta documentação é auto gerada pela especificação AsyncAPI e é automaticamente mantida atualizada através do uso dos fluxos de trabalho do GitHub. Por outro lado, usa-se um repositório do GitHub para conter todas as definições dos contratos das APIs, para serem referenciadas na especificação. Por fim, utiliza-se a ferramenta Microcks para a validação das APIs e do trabalho realizado, uma vez que esta permite realizar testes de contrato às APIs.

Em suma, é possível concluir que o objetivo de estudar soluções para ajudar a manter a interoperabilidade do sistema com a evolução das API cumpre-se, uma vez que a prova de conceito que apresentada abordou todos os tópicos identificados. Em concordância, a avaliação da experiência de utilização por tópico antes e depois do trabalho realizado apresenta uma melhoria genérica.

Palavras-chave: Interoperabilidade de Sistema, Web APIs, Evolução de Sistemas Web, Soluções de Interoperabilidade

Contents

List of Figures xv

List of Tables xvii

List of Acronyms xix

1 Introduction 1
1.1 Context 1
1.2 Problem 2
1.3 Objectives 4
1.4 Code of ethics and professional conduct 6
1.5 Restrictions 6
1.6 Approach Taken 7
1.7 Research methodology 7
1.8 Document Structure 8

2 State of the Art 11
2.1 Web API concepts 11
2.1.1 API lifecycle management 12
2.1.2 Web API types 14
2.1.3 Web API documentation handling 24
2.1.4 Web APIs schema registry 25
2.1.5 Web APIs versioning 26
2.1.6 API testing 27
2.2 System interoperability 30
2.2.1 System interoperability architecture 31
2.2.2 Architecture APIs patterns and good practices 33
2.2.3 Current system APIs interoperability solutions 34

3 Requirements and proposed solution 41
3.1 Requirements engineering 41
3.1.1 System actors 41
3.1.2 Functional requirements 42
3.1.3 Non-functional requirements 43
3.2 Solutions choices for requirements 44
3.2.1 Lifecycle management and versioning 45
3.2.2 Documentation 45
3.2.3 Schema registry 46
3.2.4 API testing 46
3.2.5 Conclusion 46

4 Value analysis **49**
4.1 Terminology 49
4.2 New concept development model 50
4.2.1 Opportunity identification 51
4.2.2 Opportunity analysis 52
4.2.3 Idea generation and enrichment 53
4.2.4 Idea selection 54
4.2.5 Concept definition 59
4.3 Value 59
4.3.1 Perceived value 59
4.3.2 Value proposition 60
4.4 Value analysis conclusions 63

5 Development, Testing and Delivery **65**
5.1 Development 65
5.1.1 APIs design 68
5.1.2 Implementation details 70
5.2 Testing 82
5.3 Delivery 87
5.3.1 Integration in pipelines 87
5.3.2 AsyncAPI how to guide 89

6 Experimentation and Evaluation **91**
6.1 Problem Description 91
6.2 Objectives 91
6.3 Hypotheses 92
6.4 Identification of indicators and sources of information 92
6.4.1 Indicators 92
6.4.2 Sources of information 93
6.5 Description of the evaluation methodology 93
6.5.1 Methodology - questionnaire 93
6.5.2 Methodology - Implementation 93
6.6 Evaluation results 93
6.6.1 Results - questionnaire 94
6.6.2 Results - implementation 96

7 Conclusions **97**
7.1 Achieved objectives 97
7.2 Critical analysis of work carried out 98
7.3 Future work 98

Bibliography **99**

A List of architectural strategies from "Architectural decision-making on interoperability in software-intensive systems" **103**

B AHP - analysis auxiliary calculations **105**
B.1 API lifecycle management 105
B.2 API documentation handling 106
B.3 API schema registry 108

B.4 API versioning 109
B.5 API testing 111
B.6 Pricing 112

C AsyncAPI - Generated documentation 115

D AsyncAPI - how to guide 119

E Questionnaire for the experimentation and evaluation 123

List of Figures

1.1 Design science research process (DSRP) model 7

2.1 Chappell's view of ALM 13
2.2 Remote Procedure Call flow 15
2.3 SOAP Web service model 17
2.4 SOAP message example 17
2.5 RESTful Web API Architecture 19
2.6 GraphQL Schema example 21
2.7 GraphQL Client's query example 22
2.8 GraphQL API interactions 23
2.9 Schema registry system's architecture 25
2.10 API Versioning using gateway 27
2.11 APISIX's dashboard 35
2.12 High-level overview of APISIX's software architecture 36
2.13 AsyncAPI specification document example 37
2.14 AWS API Gateway architecture 38

3.1 Use case diagram 42
3.2 Solution's architecture high level draft diagram 45

4.1 Innovation process stages 50
4.2 New concept development model 51
4.3 Flutter Entertainment plc 2021 growth 52
4.4 AHP Hierarchy Structure 54
4.5 Comparison matrix between criteria and solutions 58
4.6 Value proposition canvas 62

5.1 IMG and TMT - Container Diagram. 69
5.2 IMG and TMT - Deployment Diagram. 70
5.3 AsyncAPI version. 71
5.4 AsyncAPI identifier. 72
5.5 AsyncAPI default content type. 73
5.6 AsyncAPI information. 73
5.7 AsyncAPI tags. 74
5.8 AsyncAPI external documentation. 74
5.9 AsyncAPI servers - dev. 75
5.10 AsyncAPI servers - prd. 76
5.11 AsyncAPI channels. 78
5.12 AsyncAPI compoenents messages definition. 79
5.13 AsyncAPI components bindings and traits definition. 80
5.14 AsyncAPI components schemas definition. 80
5.15 GitHub schema registry repository - flutter-schema-registry. 81

5.16 AsyncAPI TMT's output payload. 82
5.17 AsyncAPI makefile. 82
5.18 Microcks docker containers for AsyncAPI. 83
5.19 Microcks APIs. 83
5.20 Microcks operations details. 84
5.21 Microcks AsyncAPI changes. 85
5.22 Microcks test generation. 85
5.23 Microcks test conformance score. 86
5.24 Microcks test history. 86
5.25 GitHub workflow - AsyncAPI documentation generation. 88
5.26 GitHub workflow - Generated documentation commit example. 89
5.27 GitHub - Generated documentation location. 89

6.1 Questionnaire - section before AsyncAPI results. 94
6.2 Questionnaire - section after AsyncAPI results. 95
6.3 Questionnaire - section for solution usability results. 95

A.1 Complete list of identified architectural strategies from the literature review 104

C.1 AsyncAPI generated documentation - overview. 115
C.2 AsyncAPI generated documentation - operations. 116
C.3 AsyncAPI generated documentation - servers. 117
C.4 AsyncAPI generated documentation - sidebar. 118

D.1 AsyncAPI guide - part 1. 119
D.2 AsyncAPI guide - part 2. 120
D.3 AsyncAPI guide - part 3. 121
D.4 AsyncAPI guide - part 4. 122

E.1 Questionnaire - initial page. 123
E.2 Questionnaire - section before AsyncAPI. 124
E.3 Questionnaire - section after AsyncAPI. 125
E.4 Questionnaire - section for solution usability. 126

List of Tables

2.1 Pros and cons of RPC-based APIs 16
2.2 Pros and cons of SOAP APIs 18
2.3 Pros and cons of RESTful APIs 20
2.4 Pros and cons of APIs using GraphQL 24
2.5 Architectural strategies suggested by "Architectural decision-making on interoperability in software-intensive systems" for each interoperability level . 33
2.6 Interoperability solutions comparison 39

3.1 Functional requirements by stakeholders 43
3.2 Non-Functional requirements by category 44
3.3 Solutions choices per requirement 47

4.1 Saaty fundamental scale (Thomas Lorie Saaty 1990). 55
4.2 Criteria pairwise comparison matrix. 55
4.3 Normalized criteria pairwise comparison matrix and relative priority. 56
4.4 Priority vector matrix. 56
4.5 Intermediate vector matrix. 56
4.6 Random index values. 57
4.7 Perceived value - benefits and sacrifices. 60

5.1 Development backlog. 67
5.2 API lifecycle stages 72

B.1 API lifecycle management - comparison matrix. 105
B.2 API lifecycle management - normalized comparison matrix and relative priority. 105
B.3 API lifecycle management - Priority vector matrix. 105
B.4 API lifecycle management - Intermediate vector matrix. 106
B.5 API documentation handling - comparison matrix. 106
B.6 API documentation handling - normalized comparison matrix and relative priority. 107
B.7 API documentation handling - Priority vector matrix. 107
B.8 API documentation handling - Intermediate vector matrix. 107
B.9 API schema registry - comparison matrix. 108
B.10 API schema registry - normalized comparison matrix and relative priority. . 108
B.11 API schema regitry - Priority vector matrix. 108
B.12 API schema registry - Intermediate vector matrix. 109
B.13 API versioning - comparison matrix. 109
B.14 API versioning - normalized comparison matrix and relative priority. 110
B.15 API versioning - Priority vector matrix. 110
B.16 API versioning - Intermediate vector matrix. 110
B.17 API testing - comparison matrix. 111
B.18 API testing - normalized comparison matrix and relative priority. 111

B.19 API testing - Priority vector matrix. 111
B.20 API testing - Intermediate vector matrix. 112
B.21 Pricing - comparison matrix. 112
B.22 Pricing - normalized comparison matrix and relative priority. 113
B.23 Pricing - Priority vector matrix. 113
B.24 Pricing - Intermediate vector matrix. 113

List of Acronyms

AHP Analytic Hierarchy Process.
ALM Application Lifecycle Management.
AMQP Advanced Message Queuing Protocol.
API Application Programming Interface.
AWS Amazon Web Services.

CI Consistency Index.
CI/CD Continuous Integration/Continuous Delivery.
CR Consistency Ratio.

DSRM Design Science Research Methodology.

EKS Elastic Kubernetes Service.
ESB Enterprise Service Bus.

FFE Fuzzy Front End.
FRs Functional Requirements.

GraphQL Graph Query Language.
GraphQL SDL GraphQL Schema Definition Language.
gRPC Google Remote Procedure Call.

HATEOAS Hypermedia As the Engine Of Application State.
HTML HyperText Markup Language.
HTTP Hypertext Transfer Protocol.

JSON JavaScript Object Notation.
JSON-RPC JSON Remote Procedure Call.

MQTT Message Queuing Telemetry Transport.

NCD New Concept Development.
NFRs Non-Functional Requirements.
NPD New Product Development.

OAS OpenAPI Specification.

PPB PaddyPower Betfair.

REST REpresentational State Transfer.
RI Random Index.

RPC	Remote Procedure Call.
SOA	Service-oriented Architecture.
SOAP	Simple Object Access Protocol.
STOMP	Streaming Text Orientated Messaging Protocol.
URI	Uniform Resource Identifier.
URL	Uniform Resource Locator.
URN	Uniform Resource Name.
US	User Stories.
VA	Value analysis.
WSDL	Web Service Description Language.
XML	eXtensible Markup Language.
XML-RPC	XML Remote Procedure Call.
YAML	Yet Another Markup Language.

Chapter 1

Introduction

This chapter introduces the topic that will be studied in this thesis: how to manage APIs evolution while maintaining systems' interoperability. Having this in mind, the necessary context regarding Blip (Flutter Entertainment plc) and its betting business is explained.

The problem and the goals of the presented work will also be detailed so that its scope is understood. Furthermore, the restrictions for the development task and the approach taken during this study are described and the document structure explained.

Additionally, Section 1.7 gives an overview of the Design Science Research Methodology, the research methodology that was used during the whole implementation of this work.

1.1 Context

This document was developed with the purpose of improving the student's research work development and also produce a document of a technical-scientific nature on a topic chosen by the student, in this case, Managing APIs evolution while maintaining system interoperability.

The work developed during this study was carried out at Blip, a tech company based in Porto, founded in 2009 and whose main focus is the development of software for online betting brands. Blip was considered by Exame journal (*Exame: E as melhores empresas Para Trabalhar São...* 2022) as one of the best companies to work for, multiple times, and this year it scored 2nd place as the Best Large Company to Work in Portugal, and 9th in the general ranking.

Blip started developing software for the betting brand Betfair and was later acquired by it. In 2016, Paddy Power and Betfair merged to become a unique brand, PaddyPower Betfair (PPB), which would eventually become Flutter Entertainment plc. In 2020, Flutter and The Starts Group joined forces to create a global sports betting, gaming and entertainment provider for over 17 million customers worldwide, with operations in more than 100 markets, annual revenues of over £6 billion, standing out has a global player (*Blip | Blip, a flutter company* 2022).

Today, Flutter Entertainment group has multiple brands associated with it, such as Betfair, Paddypower, Fanduel, Sportsbet, and more, meaning that there are millions of transactions every day split up throughout thousands of components. With more than 500 employees, Blip is responsible for the majority of the components that communicate through Web APIs that sustain all the betting features that are part of sites like PaddyPower and Fanduel, making it very challenging for developers to manage the system as a whole.

APIs are intermediary interfaces between two applications, which can be microservices, for example, which makes it possible to clearly limit the boundary of each service (Newman 2021). However, this study will be focusing on Web APIs, which make use of the Hypertext Transfer Protocol (HTTP) protocol to exchange messages across the network (Richardson, Amundsen, and Ruby 2013).

Right now, the API's contracts are defined in shared libraries that all those involved with the API have access to. When there is the need to change anything, it must be coordinated with all the teams that use the library, in order to attempt to maintain the system's interoperability.

Interoperability is most often associated with a way to enable disparate software applications to work together. It enables business processes to flow from one application to another, sharing critical business information, ultimately becoming the glue between systems and applications (Jacot 2009). As the system grows, the interoperability starts to decline, becoming a headache to control all the changes done to the contracts because of the amount of APIs, its clients, and the multiple changes occurring simultaneously.

Flutter is currently facing problems related to the evolution of its APIs and, overall, there has been a decrease in the system interoperability. Managing APIs' versions and documentation has been as challenging as keeping track of existing APIs and their features. These problems will be discussed in detail in Section 1.2.

So the question stands: how is it possible to manage APIs evolution while maintaining systems' interoperability? The objective of this work is to answer this exact question, studying how it is possible to handle APIs evolution while maintaining system interoperability and coming up with a conclusion for the best approach to handle this challenge.

1.2 Problem

With today's quick and easy access to information, we are experiencing a tremendous preference for online solutions, making the number of accesses by users increase drastically, regardless of the platform. In 2021, the global sports betting company Flutter Entertainment had a revenue of 6 billion GBP (*Flutter Entertainment plc Annual Report and Accounts 2021* 2022), meaning that there are billions of transactions every day. Focusing on PPB, which has approximately 100 components, these are translated, in 2021, to a mean of 3.70 million bets per day on Sportsbook and a mean of 4.38 million bets per day on Exchange.

Associated with the expansion of users every year, Flutter Entertainment wants to provide its betting brands' customers with the best experience possible. Therefore, it has the goal of not only improving the current features offered, to retain today's customers, but also creating new functionalities to attract more clients.

Having said this, it is normal that some components end up needing modifications to fulfil the product's requirements and sometimes these changes led to adaptations to the APIs' schema. As APIs have contracts defined between them, most of the time that they suffer transformations, their contracts with the APIs' clients are broken. Clients need to have time to adapt and do the necessary changes.

This means that through time, as more components are added, the system gets more complex and, as for PPB alone, there are more than 400 API's connections to handle in the system, becoming very challenging to handle the Web APIs' documentation. At the

moment, since the documentation is written and maintained by developers, with time and the number of changes, it quickly becomes outdated.

Additionally, all the necessary modifications to the API's contracts, including the ones demanded by the product's vision, are managed manually. It means that, when there are changes that will affect other services, the notification is done by one of the API's team members to another member of a team of the API' client.

In Flutter, there are capabilities to ensure that there is a well-defined scope of the APIs that have relations, and also to make sure that teams have a place to notify or request changes to the components. However, as it is possible to assume, as the system got bigger, communication started to be exhausting and some compatibility issues such as key information missing, wrong market pricing or even runtime crashes could start to appear.

As a cause of the system's growing complexity, the way modifications are made known between APIs and the fact that their documentation starts to get outdated, more schema compatibility issues start to come up between APIs and it gets more demanding to manage their versions.

As a result of the previously described problems, developers take more time to fix likely problems and to develop new features, which implies more money spent and fewer features released. It also means that it gets more challenging for architects to control their systems and to consider new technologies for their new components, as the interoperability starts to decrease.

In conclusion, today's needs make the product business to be in frequent transformation, which leads to an evolution of the system, meaning that there is an increase in services and connections, causing a constant change in the contracts between services, making it very demanding to keep the Web APIs documentation up to date. Additionally, there is no automation to notify the API's clients about a change in the contracts, which complicates version management. As a consequence of all that, the development time is aggravated and it turns out to be difficult to innovate on the technological stack, as the system's interoperability decreases.

To sum it up, the identified problems are:

1. Systems grow fast and it gets challenging to know which APIs exist and what are their contracts.
2. Constant changes on APIs' contract makes it hard to notify their clients and give them time to adapt.
3. Manual management of APIs documentation is error-prone and time consuming.
4. Increase of development time as teams have to align changes to their client-servers APIs.
5. Difficult to manage the system as a whole, maintain its interoperability and change the tech stack.

Only after addressing the determined problems, it is possible to produce a study and implement the approach that answers the question of how it is possible to handle APIs evolution while maintaining system interoperability.

1.3 Objectives

Currently, Flutter does not have a solution to solve the previously stated problems. A study of the state of the art should be done so that it is possible to consider existing solutions for the issue. If none is adequate, it should be proposed an in-house solution.

Therefore, the main objective of this work is to develop a study on how it is possible to handle APIs evolution while maintaining system interoperability and coming up with a conclusion for the best approach to handle this topic.

Consequently, some objectives have to be explored, being them the following:

1. **Gain an understanding of the relevant theoretical concepts**

   - **Understand and document the concept of Web API, system interoperability and API lifecycle management**

     Before getting into the study, it is crucial to have a deep understanding of the most important concepts related to the case study: the concept of an API, especially the Web API, and the concept of system interoperability. Additionally, the API Lifecycle Management topic should be explored. Only after acquiring the necessary knowledge on these two subjects, it's possible to start exploring the problem and to reach conclusions.

     Keywords: Application Programming Interface, API, Web API, interoperability, system interoperability

   - **Systematise differences between Web API protocols**

     After fully understanding what is a Web API, the different types that exist and their differences must be studied. To be more specific, the variations on the basic details, if they handle versioning and how the documentation process works.

     Keywords: Web API, Web API protocols, Web API types, API tools

   - **Solutions for APIs documentation**

     Another important objective is to find a more autonomous way to handle the APIs' documentation. So, it is essential to study the solutions available and compare them.

     Keywords: Web API documentation, API documentation

   - **Solutions for APIs schema registry**

     One of the major objectives of this study is how to follow the system APIs' evolution, so it is crucial to exist an APIs' schema automation registry. To reach this goal it should be explored what are the current approaches being used for APIs schema registry.

     Keywords: Web API schema, Web API registry, schema registry API

   - **Solutions for APIs versioning**

     Another significant objective is to identify best practices for API versioning. Therefore, it is crucial to analyze various solutions and document their features and advantages.

Keywords: Web API versioning, API versioning

- **Existing architecture APIs patterns and good practices**

  To reach a reasonable solution to the problem it is important to investigate the best Architecture APIs patterns and good practices that may be worth using. In addition, researching what problems could they possibly solve and how to combine them, is the main purpose of this subsection.

  Keywords: architecture APIs, Web API patterns, architecture APIs good practices

- **System interoperability architectures**

  There are multiple subjects to study regarding system interoperability architectures and the purpose of this topic is to gather these architectures' details.

  Keywords: interoperability architectures, API system interoperability

- **Current system APIs interoperability solutions**

  After understanding system interoperability architectures, current solutions to maintain system APIs' interoperability must be documented.

  Keywords: APIs interoperability solutions, APIs evolution, manage APIs interoperability

2. **Investigate solutions that enable the preservation of system interoperability during the evolution of APIs.**

   - **Best approach to solve the problem based on the findings**

     After gathering all the essential knowledge regarding Web APIs evolution and system interoperability, the designed solution for the problem should be documented.

     The description should be detailed enough to include all the necessities required to maintain the system interoperability with API's evolution. More specifically it should describe all the good practices and patterns, architectural design details, documentation, version handling, and APIs/schemas registry and discovery.

   - **Proof of concept**

     The ultimate objective of this document is to present, in the form of implementation, the best solution found for the problem. For the proof of concept, two Web APIs from Flutter's system should be used.

3. **Criticism of the developed solution**

   - **API testing**

     Finally, after gathering all the possible solutions to the problem, the API testing topic should be researched in order to test the solution and find out if it truly solves the problem. Subjects like static analysis, API's validation, and API's contract testing could be studied and applied.

     Keywords: API testing, API static analysis, API's validation, API contract testing

It is essential to acknowledge that there are numerous topics that could be explored in order to achieve the ultimate objective of this study. Concepts such as Continuous Integration/Continuous Delivery (CI/CD), API analytics, security, bug handling, configuration management, among others are already being adequately addressed by the company. It is important to emphasize that Flutter Entertainment deals with multiple brands, so configuration management plays a vital role. Since each brand requires unique configurations, the software system is designed to be flexible enough so that it allows the same product to be customized and delivered with different characteristics, meeting the specific requirements of each brand.

As a result, the previous concerns will not be explored in this work, despite their possible significance to the main subject of handling API evolution while ensuring system interoperability, and the main focus of this study will be on the stated objective topics.

1.4 Code of ethics and professional conduct

According to the Association for Computing Machinery 2023, the code of ethics and professional conduct is a set of principles and guidelines that outline the values, responsibilities, and obligations of individuals in the field of information technology. These principles promote ethical behaviour and ensure that professionals comply with standards of integrity and fairness in their practices. During the development of this study, the code of ethics and professional conduct was followed and respected.

The study follows the code of ethics and professional conduct to provide valuable insights and contribute to the progression of the field while preserving the highest level of professionalism.

The study was supervised by Flutter Entertainment's representative, Engineer Miguel Veiga, who ensured that all the legal requirements and ethical standards were strictly followed, including the appropriate use of licensed computer/software resources. This showcases the dedication of the study and its supervision to ethical practices and proper behaviour.

1.5 Restrictions

Although not many constraints were imposed, there are some topics which require attention. The first is related to the potential developed code to address the problem: the code should not be made public and it should only be accessible to Flutter Entertainment employees. All the developed software or documentation regarding patterns or good practices to deal with the problem should be saved within the organization's repositories using git as the distributed version control.

Furthermore, the repositories' names and documents' titles and structure should be approved by the company's supervisor of the study, since it will be for the company's benefit.

Additionally, there are multiples concepts associated with system interoperability worth exploring, but, for this case study, the topics explored should only be the ones stated in Chapter 1.3.

On top of this, there is a limited time range to develop the study and reach conclusions that answer the problem. The work should be developed and documented with the aim of being delivered by the end of July. In an exceptional case, there is the possibility of extending the deadline until October 2023.

Lastly, the acceptance criteria imposed by the company must be met. This includes addressing the problem at hand and compiling a comprehensive guide, containing all necessary information and best practices to achieve the goals of the study, but focusing only on the objectives' topics. Ultimately, the final document will be evaluated through its practical implementation using two components of the Flutter system as proof of concept.

1.6 Approach Taken

The development of this work was divided into three main sections: a study of the subject to gather all the necessary knowledge for the work, designing a solution that answers the problem based on the preliminary study and implementing what is thought to be the best solution to solve the problem.

The study will be carried out based on scientific documents obtained from platforms like B-on (*B-on* 2022), Google Scholar (*About Google Scholar* 2022), IEEE Xplore (*IEEE Xplore* 2022) or ACM Digital Library (*About ACM DL* 2022). Thereafter, the outcome of the study will be used to come up with a solution to solve the problem, which will be implemented using two Web APIs from Flutter's system.

In the end, API testing will be executed to validate if the designed and implemented solution solves the presented problem.

1.7 Research methodology

The present document is a research study produced to answer a real-life problem at Flutter Entertainment plc. With this said, this dissertation follows the Design Science Research Methodology (DSRM) which is a design science research framework that can be divided into six phases as shown in Figure 1.1.

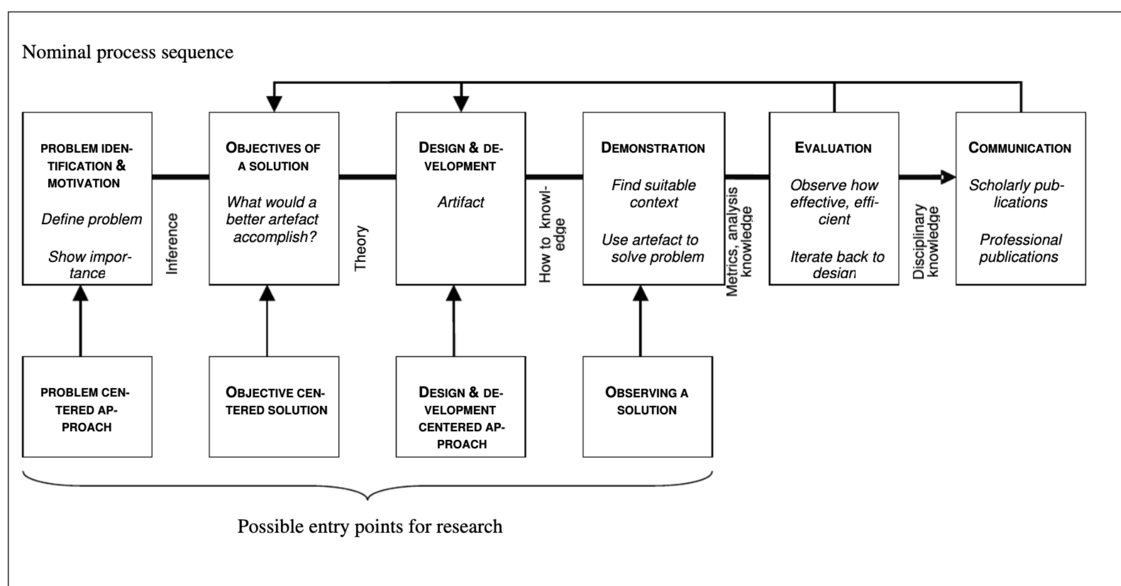


Figure 1.1: Design science research process (DSRP) model (Peffers et al. 2006).

To sum up, the DSRM is a research framework developed to be iterative, and the main iterations are (Peffers et al. 2006):

1. **Problem identification and motivation** The problem is detailed and the benefits that could be gained from the solution are shown. This allows the researcher to better understand the problem and for the solution to be accepted easier. This iteration can be found in Chapter 1.2.
2. **Define the objectives for the solution** The objectives, quantitative or qualitative, of the research are described as well as the knowledge of the state of the problem and other solutions. Chapter 1.3 represents this phase in the present document.
3. **Design and development** The design and architecture of the solution that solves the problem are developed and then the designed artefact should be developed. The design and development of the solution can be found in Chapters 3 and 6 respectively.
4. **Demonstration** Establish that the designed solution can solve the problem and how. This could involve its use in experimentation, simulation, case study, proof, or other appropriate activity. This iteration of the DSRM framework will be explored in Chapter 6.
5. **Evaluation** Analyze the solution by comparing the expected to the observed results. This represents the point where the researcher can choose to go back to a previous activity or continue. In order to validate the designed solution, there will be tests put in place. Those tests will be detailed in a forthcoming Section in Chapter 6.
6. **Communication** Inform the community about the study: problem, solution and results. The expected final iteration of this work is the presentation to a public audience and a panel of judges for review and evaluation.

1.8 Document Structure

The purpose of this thesis is to analyze and understand the issue of maintaining system interoperability during API evolution. To accomplish this, Section 2.2.2 has a comprehensive examination of current solutions in the market will be conducted to identify the most critical topics relevant to the problem.

A literature review will be presented in Chapter 2.1, encompassing the key concepts surrounding API lifecycle management, documentation, versioning, testing, schema registry, and in Chapter 2.2, API architecture patterns and best practices.

Once the necessary understanding of the relevant topics have been established, a solution will be proposed to address the interoperability challenge in Chapter 3.2. This solution will be compared to existing solutions in the market to assess its efficacy.

A cost-benefit analysis will then be conducted in Chapter 3, comparing the proposed solution with the existing solutions studied in the literature review to determine the most effective approach.

Following that, there is an experimentation and evaluation in Chapter 4 that will evaluate the results of the proposed solution through a series of statistical methods and surveys of relevant stakeholders.

The value analysis is succeeded by the implementation phase where the details of the implementation, testing and delivery choices of the proof of concept are shown. This phase can be seen in Chapter **??**.

After that, a detailed evaluation of the implemented proof of concept is presented and the results are reviewed. This evaluation can be consulted in Chapter 6.

Lastly, in Chapter 7, the conclusions of the carried work are presented. The achieved objectives and future work are identified, as well as it is done a critical analysis.

Chapter 2

State of the Art

This chapter intends to present some essential knowledge regarding the topic of study: how it is possible to handle APIs evolution while maintaining system interoperability.

With this said, to reach the best possible decision that addresses the problem, it is essential to make a theoretical introduction of the main concepts, being this the purpose of this Chapter.

Accordingly, the Chapter 2 is divided into two different parts; one related to the scientific topics that Sections 2.1, 2.1.1, 2.2, 2.2.1 and 2.2.2, are part of; and the other part about technologies, covered by Sections 2.1.2, 2.1.3, 2.1.4, 2.1.5, 2.1.6 and 2.2.3.

2.1 Web API concepts

The term Application Programming Interface appeared a long time ago and, although it can have multiple interpretations depending on the context, it can be generally described as an interface to a reusable software entity used by multiple clients (Robillard et al. 2013).

Although Robillard et al. described the term API as simple as a point of connection between two components, it also makes sense to see it as three different layers; the public interface, the actual implementation of the functionality and an intersecting layer which provides utilities such as error handling, third-party libraries and additional auxiliary features (Granli et al. 2015).

Having this said, APIs have become a vital external representation of an organization; the digital "face" upon which companies base their brand impressions (IBM 2016), which means that, eventually, APIs started being used on the web and named Web APIs.

As the name suggests, Web APIs are just like APIs, but typically use web services to enable the consumption of services and data over the network. Each of these services exposes one or more endpoints that accept requests and return responses, usually over HTTP, and allows to perform operations or access a particular piece of information (Martin-Lopez 2020).

While utilized for decades the concept of API gained momentum through web-based services. There are multiple Web APIs tools in use nowadays (e.g.: REpresentational State Transfer (REST), Simple Object Access Protocol (SOAP), Graph Query Language (GraphQL), Remote Procedure Call (RPC)-based protocols, etc.) but those will be explored in detail in Section 2.1.2.

The following Sections, spanning from 2.1.1 to 2.1.6, serve the purpose of providing a comprehensive contextualization of relevant components identified in the literature. This

contextualization is crucial for the subsequent phases of the research, enabling an extensive understanding of the research theme and facilitating the analysis and development processes.

Keywords: Application Programming Interface, API, Web API

2.1.1 API lifecycle management

When considering a new software development it is essential to keep in mind the whole lifecycle of the applications and not only the present development. It is crucial to contemplate how forthcoming updates, substitutions or even discontinuation of the application will be managed, and these types of concerns are typically addressed in the lifecycle management of APIs.

Furthermore, the establishment of protocols/policies for API usage are commonly addressed in this field of research, with the aim of promoting consistency and uniformity in API utilization practices.

Accordingly, several lifecycle models have been introduced that define different lifecycle activities focusing on different goals such as requirements engineering, design, development and testing (Tüzün et al. 2019).

As a consequence of the above, the concept of Application Lifecycle Management (ALM) started to appear. For Chappell 2008, ALM could be defined as: "the entire time during which an organization is spending money on this asset, from the initial idea to the end of the application's life".

With this said, Figure 2.1 represents the three parallel aspects that Chappell divided ALM into:

- **Governance**

  This first stage is the one that makes sure that the application always provides what the business needs, starting even before the development phase.

- **Development**

  Following, the development aspect represents the interactions between the development and maintenance of the application and where the necessary updates take place.

- **Operations**

  Lastly, the operations aspect will reflect the constant updates done during the development and maintenance process, resulting in constant monitoring of the application and deployment of the updates. Eventually, this phase leads to the end of the application life.

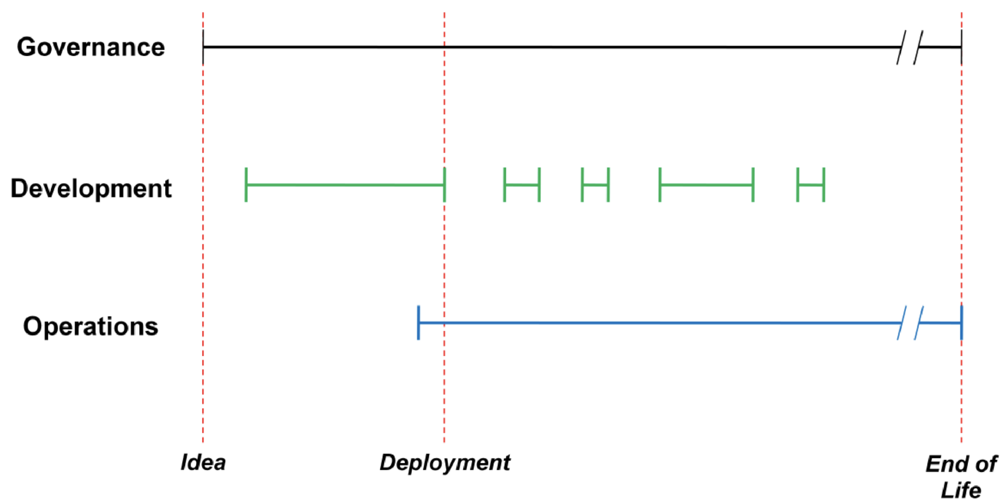


Figure 2.1: Chappell's view of ALM (Chappell 2008).

Another view of the concept of ALM is the one described by Vasudevan 2017 in an article written for the Swagger blog. The author defends that API lifecycle management has become a crucial engineering concern to build concrete API solutions. Therefore, the API lifecycle management is seen as five stages:

- **Planning and Designing the API**

  This stage involves the planning and design of all the resources and operations, aligned with the business case scenarios so that ultimately the development can start.

- **Developing the API**

  Having the vision and design the API development can take place. The ease and speed with which the desired API is implemented are crucial, but it is important to keep in mind that it is an iterative process.

- **Testing the API**

  Once the API is available to other developers it should be completely tested and monitored for performance issues and improvements.

- **Deploying the API**

  After testing, the API should be made available for customers, but to deliver guaranteed and high-quality performance, it is also essential to monitor the APIs in production continuously.

- **Retiring the API**

  Naturally, deprecation is part of an API lifecycle, and this is the phase where the support for a concrete version, or even the retirement of the entire API happens.

As for Red Hat, ALM is the people, tools, and processes that manage the lifecycle of an application from conception to end of life (Red Hat 2020). In similarity to Chappell's view, Red Hat divides ALM into the following stages:

- **Application governance**

Represents the beginning process of creating a new application, from the initial idea to the requirements imposed by the business needs and goals. Additionally, resource management, data security and user access are also described in this phase.

- **Application development**

  After gathering the expected requirements, the designing, building, testing, and deploying stages take place. Along with it, the updates through the application lifecycle can also be considered part of the application development.

- **Software testing**

  The goal of the testing stage is to make sure that the requirements outlined by governance have been met and that the application works as it should before being released to users. If not, updates will be implemented to fulfil the expected outcome of the application.

- **Operations and maintenance**

  The operations and maintenance stage is when the first version of the application or a new one is really to be deployed. This phase also defines at what point an application will no longer be supported or a newer version will become available.

To sum up, although there are more concrete definitions for API lifecycle management, Chappell's ideas have kept their values through time. Meaning, ALM is a process that starts from the initial vision, continues during the monitoring of the application and constant updates and goes all the way until the end of the application's life.

Keywords: API lifecycle management, Application lifecycle management

2.1.2 Web API types

As referred in Section 2.1 Web APIs are used to interconnect Web services, allowing software products to work via the Internet. As a result of the natural evolution through time and the different needs of each product, various Web API architectural types have emerged and are still used today, encompassing RPC-based, Webhooks, SOAP, REST, Message Queuing Telemetry Transport (MQTT), Falcor, GraphQL, and others.

Each of these styles has its own established patterns for standardizing data exchange. The plethora of choices often leads to endless debates on which architectural style is the best, but four styles are widely recognized as the major or most distinguished: RPC-based, SOAP, REST, and GraphQL.

In this Chapter, these Web API architectural types will be discussed in the order of their historical appearance, with a comprehensive comparison of their strengths and weaknesses.

Keywords: Web API protocols, API tools, Web API

1. **RPC-based**

   RPC-based APIs are a type of APIs that allow for the execution of a procedure or function on a remote system. It is typically implemented using a client-server architecture, where the client sends a request to the server to execute a procedure or function, and the server then sends back a response with the results of the execution.

Figure 2.2 illustrates the process of RPC, where function A invokes function B. According to Fischer et al. 2019, RPC allows the separation of the code into client-side and server-side components, with a RPC stub on the client side sending an encrypted request, via a communication protocol stack, to the server. On the other hand, a RPC skeleton is on the server side deserializing and processing the request, executing the function, and returning the response to the client. Prior to RPC, these functions would have to reside within the same machine or software process.

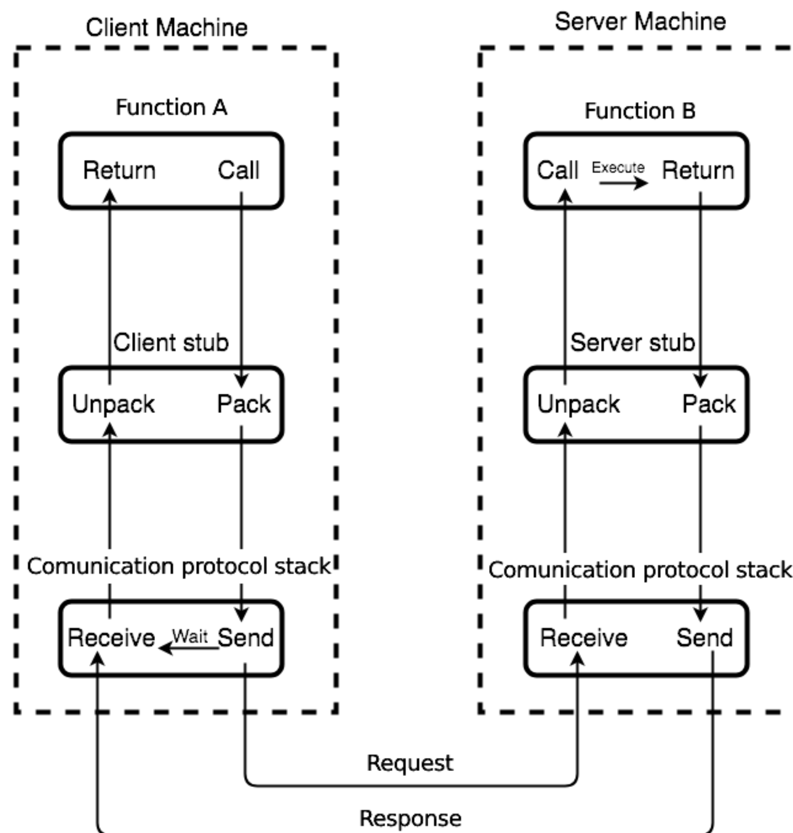


Figure 2.2: Remote Procedure Call flow.

There are different types of RPC-based APIs, such as XML Remote Procedure Call (XML-RPC)[1], JSON Remote Procedure Call (JSON-RPC)[2], Apache Thrift[3] or Google Remote Procedure Call (gRPC)[4], and each of these types uses different data serialization formats. These formats can include eXtensible Markup Language (XML), JavaScript Object Notation (JSON), Yet Another Markup Language (YAML), Protocol Buffers, Thrift, Avro, and others.

Although there is a variety of RPC-based APIs, typically RPC promotes system interoperability because any application that speaks RPC can communicate with another application that understands RPC (S. Wang, Hindman, and Stoica 2021).

[1] http://xmlrpc.com/
[2] https://www.jsonrpc.org/
[3] https://thrift.apache.org/
[4] https://grpc.io/

In fact, S. Wang, Hindman, and Stoica 2021 defend that the fundamental aspect that supports the efficacy of RPC is the simpleness yet powerful semantics of its programming paradigm. Notably, RPC does not rely on a shared state as arguments and return values are passed by value between processes, which forces them to be copied into the request or reply. As a result, arguments and return values are inherently immutable. These straightforward semantics promote highly efficient and dependable implementations, as it eliminates the need for distributed coordination, while still applying to a broad range of distributed applications.

Table 2.1: Pros and cons of RPC-based APIs

Pros	Cons
Simple interactions. Communication is done by calling an endpoint and receiving a response. It uses GET for data retrieval and POST for other actions.	**Tight coupling to the underlying system** The abstraction level of the API determines its reusability. RPC's lack of abstraction layer leads to security risks, scalability challenges, and difficulties in achieving loosely coupled teams. Clients may struggle with understanding the effects of calling a specific endpoint or determining which endpoint to call.
Easy-to-add functions. Easy to add new requirements to the API by creating a new function, exposing it via an endpoint and allowing clients to access it.	**Low discoverability**. No way to introspect the API or to get access to functions to call based on its requests.
Performance. Lightweight payloads improve performance and handle large amounts of messages between shared servers or for parallel computations.	**Overlapping functions**. Creating new functions is easy but it leads to a large number of overlapping functions that can be hard to understand.

Keywords: RPC protocols, RPC-based APIs

2. SOAP

SOAP is a protocol that uses XML and schemas to allow applications to communicate through HTTP. With this in mind, SOAP is at the heart of Web services architecture because it enables the interacting parties in the architecture to communicate with each other using a standard, well-understood message format (Jesus Ekie, Gueye, and Niang 2021).

Typically, a SOAP API flow starts with the service provider publishing, in the form of a list, the descriptions of the available services to the service registry. After this, the service consumer requests the right to discover the services' descriptions, so that it can query those services. Lastly, the service provider will process the request and return the correct response to the service consumer's request. This flow can be visualized in Figure 2.3.

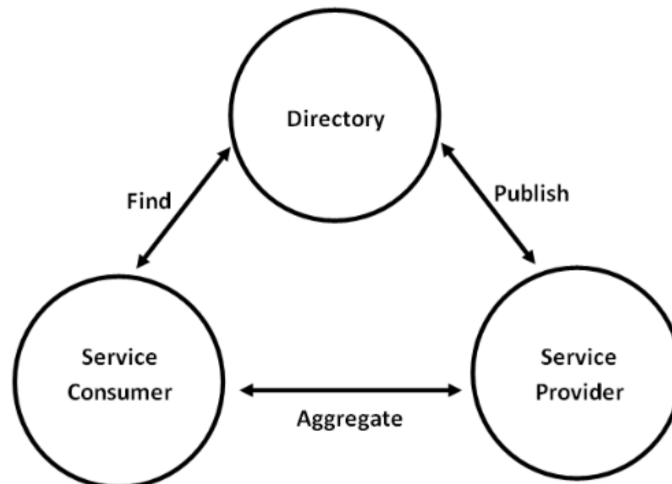


Figure 2.3: SOAP Web service model (Jesus Ekie, Gueye, and Niang 2021).

Regarding the SOAP messages, typically, they have a very simple structure: an XML element with two child elements, one of which contains the header and the other the body. The header contents and body elements are themselves arbitrary XML (Curbera et al. 2002). Figure 2.4 illustrates an example of a SOAP message, with no headers and a simple XML body representing an e-ticket with the person's and flight's details.

```
POST /travelservice
SOAPAction: “http://www.acme-travel.com/checkin”
Content-Type: text/xml; charset=“utf-8”
Content-Length: nnnn

<SOAP:Envelope xmlns:SOAP=
        “http://schemas.xmlsoap.org/soap/envelope/”>
    <SOAP:Body>
        <et:eTicket xmlns:et=
            “http://www.acme-travel.com/eticket/schema”>
            <et:passengerName first=“Joe” last=“Smith”/>
            <et:flightInfo airlineName=“AA”
                        flightNumber=“1111”
                        departureDate=“2002-01-01”
                        departureTime=“1905”/>
        </et:eTicket>
    </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.4: SOAP message example (Curbera et al. 2002).

The SOAP API logic is written in Web Service Description Language (WSDL), a description language document that describes a Web service's interface and provides users with a point of contact (Curbera et al. 2002). This means that all endpoints are defined using WSDL and all processes are described so that it is possible to quickly establish communication and perform requests.

To conclude, the Table 2.2 describes the advantages and disadvantages of a SOAP API.

Table 2.2: Pros and cons of SOAP APIs

Pros	Cons
Language and platform agnostic. SOAP Web services handle communications independently of the type of language, platform and transport.	**XML only**. SOAP messages only support verbose XML structures for requests and responses.
Variety of transport protocols. Regarding transfer protocols, SOAP can accommodate multiple protocols.	**Heavyweight**. As a result of its responses structure, SOAP services have a large sized XML response files.
Built-in error handling. It allows for specific error messages with error codes and explanations.	**Specialized knowledge**. Duo to all protocols involved with SOAP Web Services, it is required to have a deep understanding of all conventions involved.
Security extensions. Integrated with the WS-Security protocols.	**Clostly updates**. The rigid SOAP schemas slow down adoption.

Keywords: SOAP, SOAP API

3. **REST**

The modern Web APIs following the REST architectural style are referred to as RESTful Web APIs and they are implemented on top of the HTTP/s protocol and offer a uniform way to manage cloud resources usually decomposed into multiple RESTful Web services (Sohan, Anslow, and Maurer 2015). In REST architectural style the message exchanged can be transmitted directly over the HTTP protocol without the need for encapsulation and the use of envelopes, and this is one of the great advantages of the REST compared to other approaches like RPC/SOAP (Tavares and Vale 2013).

RESTful Web APIs use the HTTP Request-Response model that consists of 4 operations: GET for Read, POST for Create, PUT for Update, and DELETE for Delete (Lee and Liu 2022). Regarding the information exchange between the client and server, it can be done through XML, HTTP, JSON and others.

REST architectures have a special feature called Hypermedia As the Engine Of Application State (HATEOAS) which enables an API to be self-descriptive. It allows a client to discover and navigate the resources of an API by following hypermedia links within the responses, rather than by predefining a fixed set of endpoints, making it possible to evolve the API without breaking existing clients (Koschel et al. 2019).

Concerning documentation management, HATEOAS can provide a more dynamic and flexible approach to address documentiion in a RESTful API. Instead of predefining a set of endpoints and their corresponding documentation, HATEOAS allows for the documentation to be included as part of the API response, enabling the client to discover the available resources and their associated documentation at runtime (Koschel et al. 2019).

REST is not as strictly defined as SOAP but, nevertheless, it should comply with six architectural constraints defined by Fielding and Taylor (Fielding and Taylor 2000):

- **Uniform Interface**

  There must be a uniform way of interaction between client and server regardless of device or application type.

- **Stateless**

  The necessary state to handle the request must be sent from the client and the server does not store anything related to the session. With this, REST gains visibility, reliability, and scalability.

- **Cache**

  It must be possible to cache data from previous requests and reuse it.

- **Client-Server architecture**

  The client must know the available services so that it can send a request to them. this improves the portability and allows both the client and the server to evolve independently.

- **Layered system**

  It must be possible to isolate the responsibilities of the system's services using distinct layers.

- **Code on demand**

  Servers must be able to provide executable code to the clients so that it is available to download and execute the code on the client side.

To sum up, with Figure 2.5 it is possible to visualize an example of the big picture of a RESTful Web API Architecture.

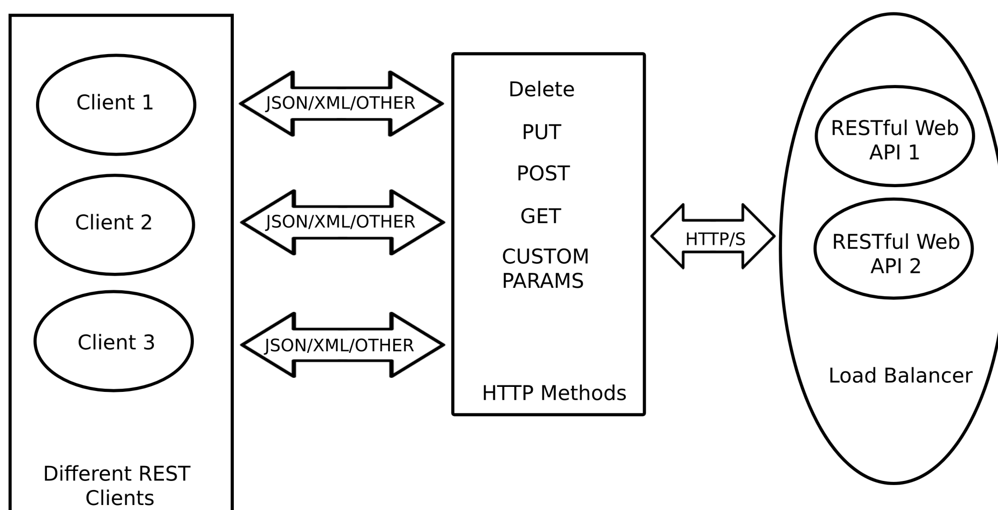


Figure 2.5: RESTful Web API Architecture.

After explaining what is a REST API, it is possible to conclude the pros and cons of this type of architecture. These advantages and disadvantages are represented in Table 2.3.

Table 2.3: Pros and cons of RESTful APIs

Pros	Cons
Decoupled client and server. REST APIs are abstract and flexible enough to evolve while remaining a stable system.	**There is not an exact way to build a REST API**. Depending on the specific scenario there are multiple ways to handle the system resources.
Discoverability. Communication between the client and server exposes all the necessary information regarding the Web API.	**Big payloads**. REST returns a lot of content so that the client can fully understand the state of the server from its responses.
Multiple formats support. REST supports multiple formats for storing and exchanging data.	**Over and Under fetching** problems. REST responses contain too much or not enough data (often there is the need for another request).
Cache-friendly. REST is one of the architectural styles that allow caching data on the HTTP level.	
Documentation management. HATEOAS allows for an agile approach to document the API, as it can evolve and change over time without requiring updates.	

Keywords: REST, RESTful Web API

4. **GraphQL**

GraphQL was developed and open-sourced by Facebook when they were facing some performance issues on their mobile applications and realized that they needed to optimize how data was sent to client applications. It is a query language for APIs that enables developers to request exactly the data they need and nothing more, making it more efficient and flexible than other APIs.

GraphQL does not dictate the specific method for constructing an API, but it does provide recommendations for conceptualizing a service. For Porcello and Banks 2018 these GraphQL recommendations can be divided into the following design principles:

- **Hierarchical**

  Organized like a hierarchical graph, where the fields are nested within other fields. Additionally, the query mirrors the shape of the data it retrieves.

- **Product centric**

  Should return exactly the data requested by the client so that their needs are satisfied.

- **Strong typing**

  GraphQL Server contains a type system that verifies the content of the data request.

- **Client-specified queries**

  Only accepts requests that follow the GraphQL Server's schema structure.

- **Introspective**

  Allows querying the GraphQL type system server, showing to clients its schema.

On the other hand, some authors have a simplified view of GraphQL, as in the case of Brito, Mombach, and Valente 2019, that state that GraphQL has only two key characteristics:

- **Support to a hierarchical data model**

  The creation of this data model is one of the first steps when building a GraphQL API because it describes the available data and its hierarchy. It is usually called schema and it has a substantial role in the reduction of the number of endpoints accessed by clients.

  A schema can be viewed as a comprehensive description of all the queries that can be made to an GraphQL API and the types of data that it returns. So, to design a schema of an API, it is required to have a strong understanding of the GraphQL's syntax language, also known as GraphQL Schema Definition Language (GraphQL SDL) (Brito, Mombach, and Valente 2019). In Figure 2.6 there is an example on a GraphQL schema with a query end-point and two types (Movie and Producer).

```
schema {
  query: Query
}

type Query {
  movie(title: String!): Movie
}

type Movie {
  id: String!
  director: Director
  title: String
  year: Int
}

type Director {
  id: String!
  name: String
}
```

Figure 2.6: GraphQL Schema example.

- **Support to client-specific queries**

  Implementing the schema prior to any data requests enables clients to validate their claims and confirm the server's capability to respond appropriately. These requests made by clients are known as queries and allow them to acquire only the precise data they need to perform a given task. Bellow in Figure 2.7 there is a valid example of a query that could have been made by a client to request the title and director name to the API with the schema of the Figure 2.6.

```
query getMovieByTitle {
  movie(title: "12 Angry Men"){
    title
    director {
      name
    }
  }
}
```

Figure 2.7: GraphQL Client's query example.

When the GraphQL receives the client's queries, they are interpreted against the entire schema and, using the received JSON, the operations needed to return the expected data are performed. As a way to retrieve data for a specific field of the client's query, asynchronous functions called *Resolvers* are used, which return the information in the type and shape specified by the schema (Landeiro and Azevedo 2020).

Given the preceding explanation of the GraphQL specifications, it is feasible to construct an understanding of the typical flow of an API that follows the GraphQL methodology. This flow can be visualized in Figure 2.8.

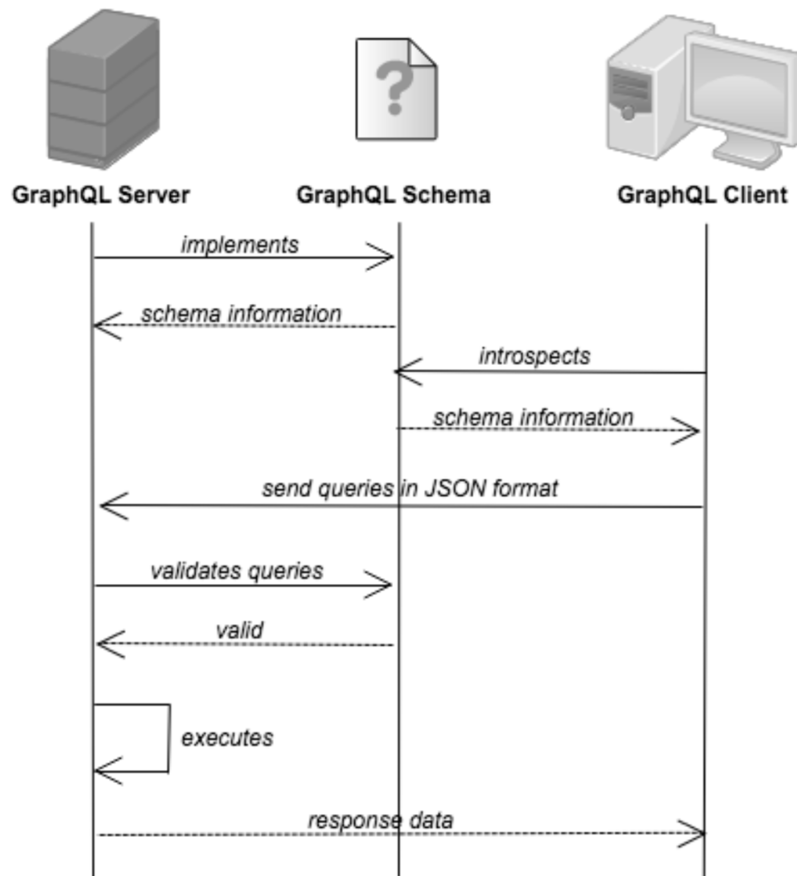


Figure 2.8: GraphQL API interactions (Landeiro and Azevedo 2020).

All these operations are not limited to a specific transport protocol. Although GraphQL is commonly served over HTTP, it is compatible with any application that relies on protocol's specifications.

With a comprehensive understanding of GraphQL, it is possible to identify the advantages and disadvantages of its usage. These benefits and limitations are summarized in Table 2.4.

Table 2.4: Pros and cons of APIs using GraphQL

Pros	Cons
Typed schema. Improves discoverability by publishing the available queries ahead of time and allowing clients to easily discover them by accessing the API.	**Performance issues**. GraphQL's power comes with increased complexity, particularly with nested fields and large requests.
Fits graph-like data. Well-suited for handling data with complex relationships, but not as effective for flat data structures.	**Caching complexity**. Requires a custom caching solution, as it does not use HTTP caching semantics.
No versioning. No need to manage versioning and the associated difficulties.	**Pre-development education**. As it is a relatively new approach developers need time to fully comprehend its specific operations and GraphQL SDL.
Detailed error messages. Offers detailed error messages, handled by the specific resolvers and the faulty part of the query.	
Flexible permissions. Selective exposure of certain functions while maintaining the confidentiality of private information.	

Keywords: GraphQL

2.1.3 Web API documentation handling

Web API documentation handling is an essential aspect of API development, as it allows developers to understand how the API functions and how to use it. It provides information such as the endpoint URL, request and response formats, authentication and authorization requirements, error handling, and sample code. API documentation is usually presented in the form of a developer portal, which is a web page or set of web pages that provide all the information a developer needs to use the API.

One of the main advantages of well-documented APIs, is that it can significantly reduce the time and effort required for integration as developers can quickly understand the functionality of the API, its inputs and outputs, and how it can be used (Watson et al. 2013).

To create and maintain API documentation, several best practices can be followed. As the API evolves, the documentation should be updated to reflect the changes. Additionally, the documentation should be easy to navigate and search, with clear and concise explanations (Watson et al. 2013).

Multiple tools can be used to create and maintain API documentation, such as Swagger[5], Postman[6], ReDoc[7] (Yang et al. 2018) or AsyncAPI[8]. Swagger is an open-source tool that allows developers to design, document, and test their APIs including what requests the

[5] https://swagger.io/tools/open-source/
[6] https://www.postman.com/
[7] https://github.com/Redocly/redoc
[8] https://www.asyncapi.com/

service can handle, what responses may be received, and the request and response formats. Accordingly, Postman is a tool that allows developers to easily test and document their APIs.

In contrast, ReDoc and AsyncAPI are tools that enable the evolution of documentation by generating it from the OpenAPI Specification (OAS)[9]. ReDoc is an open-source tool that functions as an add-on for the OAS and generates interactive documentation. AsyncAPI is a project that is utilized for describing and documenting message-driven APIs in a machine-readable format and is protocol-agnostic, meaning it can be used for APIs that operate over any protocol.

In conclusion, Web API documentation handling is a crucial aspect of API development that allows developers to understand how the API functions, how to use it and how to integrate it into their systems. Well-documented APIs can significantly reduce the time and effort required for integration, and can also help to reduce the debug time. Several tools can be used to create and maintain the APIs document updated.

Keywords: Web API documentation, API documentation management

2.1.4 Web APIs schema registry

Schema registry, also known as service registry, is a pattern that represents a central repository for storing, managing, and maintaining the schemas used to describe the data exchanged between services in a system and its usage can help to solve several issues, such as schema version management, schema compatibility, and schema evolution (Strengholt 2020).

With this said, the schema registry can be used to allow applications to dynamically discover and consume services, enabling interoperability and flexibility in a Web API architecture. One of the key advantages of using a schema registry is that it provides a centralized location for all schemas, making it easier to manage and maintain the schemas over time. In Figure 2.9 there is a visual representation of what a system's architecture using schema registry could be like.

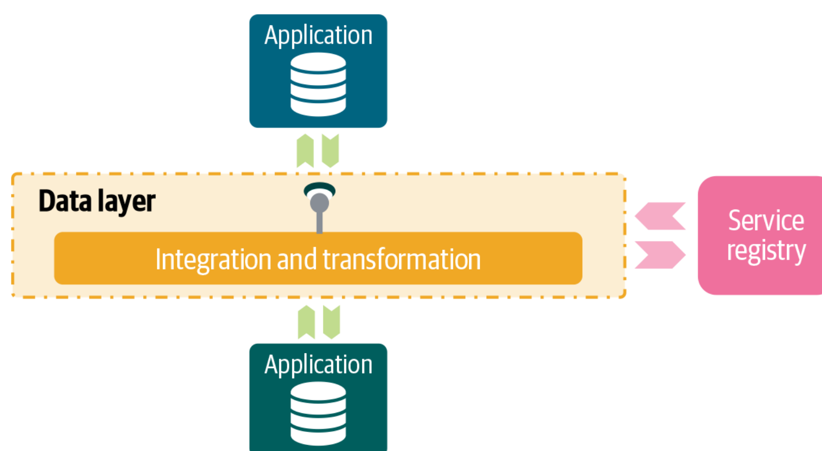


Figure 2.9: Schema registry system's architecture (Strengholt 2020).

The schema registry also represents a single source of truth for all schema versions, reducing the risk of schema drift and helping to ensure that all services are using the same version of

[9]https://swagger.io/specification/

the schema. In addition, the registry also helps to ensure that new services are compatible with existing schemas, reducing the risk of compatibility issues and enabling easier integration between services (Reselman 2021).

There are several tools available to implement this pattern, such as Confluent Schema Registry[10], Apicurio Registry[11], Red Hat Schema Registry[12] among others. These tools provide a range of features and capabilities such as schema version management, compatibility testing, and automatic schema evolution.

Having in context a messaging system using Apache Kafka[13], the usage of the schema registry could represent a central point for storing the schemas used by Kafka producers and consumers. This is the case given by Kreps, Narkhede, and Rao 2011, where Apache Avro[14] was used to serialize schemas that were later saved in a lightweight schema registry. Another example of this approach is the one stated by Quarkus 2023 that demonstrates the integration of a Quarkus[15] application with Apache Kafka, Apache Avro serialized records and a schema registry, both Confluent Schema Registry and Apicurio. According to the authors, this helped to maintain compatibility between the producers and consumers and made it easier to evolve the schemas over time.

Keywords: API schema registry, Schema registry

2.1.5 Web APIs versioning

When a Web API evolves, all the integrated applications using the Web API may not be able to evolve on the same schedule. This introduces unique challenges to the evolution of Web APIs as changes on a Web API may break the dependent applications. Old versions of the APIs may no longer be supported, a lack of adequate documentation to upgrade to a newer version, and insufficient communication of changes (Sohan, Anslow, and Maurer 2015).

This is where Web API versioning gets its relevance as it is the process of managing different versions of an API in order to ensure backward compatibility and maintain interoperability between different systems. It allows for different versions of an API to coexist and be used by different clients, enabling the API to evolve without breaking existing clients (Akbulut and Perros 2019).

For Akbulut and Perros 2019 API versioning can have two different approaches:

- **Versioning in the URI**

  This method uses the version information in the Uniform Resource Identifier (URI). This approach is semantically meaningful and keeps the representation of an API immutable. However, it can lead to a large URI footprint and inflexibility in evolving a single resource. This method can also be used as query strings.

  Example: *https://v1.apiserviceexample.com/home/* or
  https://apiserviceexample.com/home?version=1

[10] https://docs.confluent.io/platform/current/schema-registry/index.html
[11] https://www.apicur.io/registry/
[12] https://access.redhat.com/documentation/en-us/red_hat_integration/2020.q1/html/getting_started_with_service_registry/intro-to-the-registry
[13] https://kafka.apache.org/
[14] https://avro.apache.org/
[15] https://quarkus.io

- **Versioning in the HTTP header**

  This method provides the API's version in custom headers of HTTP requests, rather than in the URI. This keeps the URI clean and allows for easy migration to new versions, but can cause problems with caches and proxies and routing faults if requests are not constructed carefully. Compared to URI versioning, header versioning outputs less accessible artefacts but can make it harder to test and debug an URI.

  Example: *Api-version: 2.0* or *Accepts-version: 2.0*

Both API versioning methods have their advantages and disadvantages, but the mutual aspect is that the backend services are accessed through an API gateway pattern that, depending on the version, redirects the request to the correct service, as shown in Figure 2.10.

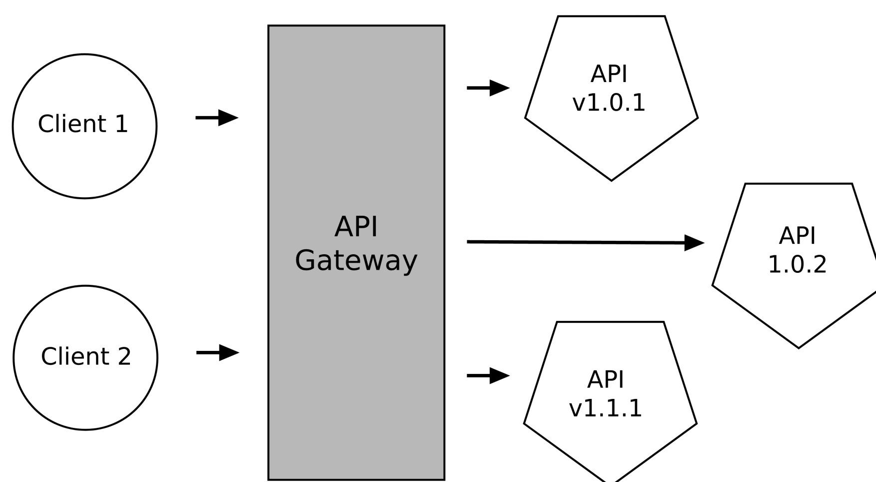


Figure 2.10: API Versioning using gateway.

Sohan, Anslow, and Maurer 2015 recommend the use of Semantic Versioning, a naming technique for versioning software so that it is possible to infer if two versions are backwards compatible simply by interpreting the version identifiers. This technique follows the format *x.y.z*, where *x* represents major changes, *y* represents minor improvements or optimizations, and *z* represents patch fixes. Major changes are typically made when the development team decides to implement changes that are not compatible with previous versions. Minor changes, on the other hand, are made to improve or optimize resources in a backwards-compatible way, allowing for requests to be redirected to services with only minor revision differences. Lastly, patches are applied to fix bugs or defects in the services Akbulut and Perros 2019.

Keywords: API versioning, API version strategies

2.1.6 API testing

Nowadays a bug in a Web API can significantly affect an organization both internally and externally. Internally, this is due to the fact that, in a big system, many services rely on other APIs. Externally, the Web API could be providing data to third-party applications and end-users which would also be affected by the bug.

With this said, testing of Web APIs is nowadays more critical than ever before, as they are the current standard for software integration (Martin-Lopez 2020). The later tests are executed, the more complex and challenging it becomes for the test engineer to find errors within a tight deadline in the product delivery software development model (Jain 2022).

Having established this, there are several advantages of Web API testing. It helps to ensure the reliability and robustness of the API, improves the user experience, and allows for easier integration with other systems. Additionally, it can detect and prevent potential issues before they can impact the business.

Web API testing can be performed manually or using automated testing tools. The process typically involves creating test cases, sending requests to the API, and comparing the results with the expected output. Automated testing tools can improve the efficiency and accuracy of the testing process.

Given all the previous information, it is important to understand that there are various forms of API testing, each with distinct methodologies and objectives. According to the author of *Learn API Testing* (Jain 2022), the different test types can be categorized into the following groups:

- **Functional testing**

  Functional testing evaluates the functional capabilities of the API, including the ability to return responses following the business requirements.

- **Performance testing**

  Assessing the response time under heavy usage is the main focus of performance testing. When a large number of requests are made to a specific API simultaneously, it must provide a response within a reasonable time frame.

- **Security testing**

  Security testing focuses on identifying and preventing unauthorized access to the API through various ways. The API must enforce strict authentication measures to ensure that only authorized individuals have access to its data.

- **Noise testing**

  Noise testing examines the presence of invalid data within requests sent to the API. It must provide an appropriate and timely response.

- **Error code and message testing**

  The process of testing for appropriate error codes and messages involves identifying situations where incorrect input data is provided, and verifying that the API responds with the appropriate error code and message.

- **Scale testing**

  Scale testing evaluates the infrastructure of an API. It involves determining the ability of the API to handle a large number of concurrent access requests and to ensure that it remains available at all times.

- **Compliance testing**

Compliance testing makes sure that the API follows the rules set by the local government where it is used. Personal information should be kept private by the company and it should not be shared.

- **Contract testing**

  Contract testing ensures that the service provider maintains a valid request payload. This is important for the service provider's business as any changes to the payload may cause issues for the consumer and negatively impact the business.

The subsequent section outlines the correlation between test types and the objectives of the present work. With this in mind, topics related to API testing, including web API static analysis, API validation, and web API contract testing are linked with testing types such as performance testing, security testing, error code and message testing, and contract testing.

1. **Web API static analysis**

   Web API static analysis is the process of examining the source code of a Web API application without executing it. The goal of static analysis is to identify potential security vulnerabilities, coding errors, and performance issues. This is done by analyzing the code's structure, logic, and data flow (Novak, Krajnc, and Žontar 2010).

   The goal of this type of testing is to detect potential errors and security vulnerabilities that would be difficult to find through testing. This approach can save time and resources by identifying errors early in the development process, allowing developers to fix them before they become costly or difficult to resolve. Static code analysis is considered an important practice in software development, with many industry experts recommending its use as part of a comprehensive software development strategy (Baca, Carlsson, and Lundberg 2008).

   Static analysis can be done using linting tools, which check for coding style and adherence to best practices, and security scanners, which look for known vulnerabilities and insecure coding patterns.

   It is important to note that while static analysis can identify potential issues, it cannot guarantee that an application is completely secure or free of bugs. It is also important to complement the static analysis with dynamic analysis, which involves executing the code and testing its behaviour in a live environment (Novak, Krajnc, and Žontar 2010).

2. **API's Validation**

   Web API validation refers to the process of ensuring that incoming requests to a Web API are valid, accurate, and conform to the expected format before processing it. This is typically done by implementing validation rules or constraints on the request data, such as checking that a required field is present or that a field contains a valid value (Pattigulla 2020).

   Validation can be performed in several ways, such as using built-in validation features of the Web framework or programming language being used, or by using third-party validation libraries.

   By doing these types of validations to Web APIs it is ensured that the request data is accurate and complete, preventing security vulnerabilities and improving the overall quality and reliability of the API by handling errors early.

3. **Web API contract testing**

   Web API contract testing is the process of validating that the interactions between a client and a service, through their APIs, meet the agreed-upon contract. It ensures that the client and the service are in sync with each other and that any changes made to the service's API do not break the client's integration (Westerveld 2021).

   The main purpose of contract testing is to catch breaking changes in the API before they are deployed to production, which can help prevent issues for consumers of the API (Diego 2021).

   This type of testing can be done using a variety of tools or frameworks and it can be addressed manually or autonomously. These tools allow for the creation of test cases that can be run against the API to ensure that the expected behaviour is occurring. Additionally, some of these tools can also be used to generate documentation for the API, which can be useful for both developers and consumers of the API.

   Depending on the Web API type there are different tools available, some more specified on contract testing than others. Some of the options that can be used are Postman, Pact[16], Testsigma[17], Karate[18], OAS and Microcks[19].

   Among these, one of the more specific tools for contract testing is Microks. Additionally, it is flexible enough to supports multiple APIs specifications including OpenAPI, AsyncAPI, a GraphQL schema, or a gRPC/Protobuf schema.

   Although there are multiple ways to implement Web API contract testing, the advantages are mutual. Web API contract testing helps to catch breaking changes, it allows the creation of test cases that can be run against the API to ensure that the expected behaviour is occurring and it also is a good excuse to generate documentation for the API.

Keywords: API testing, Static code analysis, API's validation, API contract testing

2.2 System interoperability

The definition of system interoperability can vary depending on the context, but commonly it refers to the ability of different components to communicate and exchange data with one another seamlessly and efficiently (Puspitasari et al. 2021).

Interoperability is a key requirement for the success of web services and API systems, as it enables different systems to work together flawlessly, regardless of their underlying technologies, platforms, and programming languages. Most of the time, the achievement of a higher level of system interoperability often leads to an increase in the efficiency of processes, enhancement in decision-making capabilities, and an overall improvement in the performance of the system.

According to Janssen, Estevez, and Janowski 2014 interoperability is a multifaceted concept that is commonly divided into four distinct levels:

[16] https://docs.pact.io/
[17] https://testsigma.com/
[18] https://github.com/karatelabs/karate
[19] https://microcks.io/

1. **Technical interoperability**

   The ability of systems to connect and communicate effectively through wired or wireless networks, enabling real-time data exchange and the bridging of different operating systems and software written in different programming languages through the use of web services technologies.

2. **Syntactic interoperability**

   Refers to the usage of standard protocols and formats, allowing for seamless communication and data exchange between systems. This can involve adopting specific web services technologies, such as REST or SOAP, and a specific data format, such as JSON or XML.

3. **Semantic interoperability**

   Information is understood consistently by all systems. This is achieved through the use of ontologies and semantic technologies, which provide a common understanding of the meaning of the data. Metadata describing the data, such as its publisher, quality, and potential uses, is essential for its success.

4. **Pragmatic/Organizational interoperability**

   Encompasses the organizational, collaborative and contextual aspects of data exchange, including issues of quality, trust, service agreements and context-sensitivity of meaning.

There is no singular solution to maintaining system interoperability, but various tools and best practices can be employed to achieve this objective. As it is stated by J. Wang 2019, it is important to follow good practices, such as using HTTP status codes, consistent error handling, and proper versioning, to ensure that different systems can communicate with one another. Further research and examination of will be explored in Section 2.2.2.

Additionally, tools such as an API gateway or an Enterprise Service Bus (ESB) can help to ensure that different systems are able to communicate with one another. An API Gateway is a fully managed service that can assist hosted service developers in developing or services that need to be deployed (Zuo et al. 2020). On the other hand, an ESB is a software architecture that provides a centralized platform for integrating and communicating between various applications and services in a distributed computing environment (Aziz et al. 2020). Both these concepts could be useful for this work-study but, as it is not intended to change the actual architecture in any way so, these topics will not be explored further.

Keywords: API system interoperability, Web API interoperability

2.2.1 System interoperability architecture

System interoperability architecture refers to the design and implementation of systems that can effectively communicate and exchange data with other systems, regardless of their underlying technologies or platforms.

According to a study by Abukwaik, Taibi, and Rombach 2014, system interoperability architectural problems affect all levels of interoperability, described in Section 2.2, with the syntactic level having the most identified problems.

The study analyzed twenty-two other studies to identify interoperability architectural problems and solutions and found that the most common problem, with eleven occurrences, was semantic heterogeneity of data; a problem concerning architects when designing interoperable systems that accurately interpret the meaning of data elements being exchanged among them.

The second most common problems, with seven occurrences, were syntactical heterogeneity of data and heterogeneity of communication protocols, platforms, and technical standards. The first requires architects to consider differences in data types, formats, and modelling languages of interoperating systems, while the second requires design decisions that enable the system to communicate with systems of different technical properties.

Some of the solutions found in the study of Abukwaik, Taibi, and Rombach 2014 can be examed below or, with more detail, in the authors' article:

- **Standards** can be used to solve semantic interoperability issues by creating a standard-based metadata repository for formal descriptions of data types and attributes, and by proposing standard-based modelling for processes and data.
- **Ontologies** can address issues of semantic and contextual interoperability. One possible approach is to model the basic concepts of services from a user perspective.
- **Semantic mediator** aligns semantically related concepts. It aligns behaviour, uses simpler knowledge structures, and facilitates standardized information exchange and orchestration.
- **Wrapper** encapsulates local data sources, translating queries from interoperating systems into a local form for processing. This enables the retrieval of required information from the local system.
- **Adaptors** embed connection state and logic to external systems and it can also allow data transformation between interfaces of different devices.
- **Facets** offer various implementations for a standard action interface, enabling different system types to invoke the action through its corresponding facet.
- **Middleware** handles heterogeneities in communication protocols and data formats.
- **External data models** enables all sources of data exchanged with other systems to be represented and accessible.
- **Internet data formats** used on the data level of distributed systems to ensure wide applicability of the associated components.
- **Technical reference model** helps select technical standards quickly using common vocabulary, promoting interoperability by providing system standard profiles.
- **Semantic reference model** guides in developing semantic interoperability by fulfilling a set of requirements.
- **Enterprise architecture framework** provides a systematic blueprint for building interoperability among enterprise information systems.
- **Central repository** enables the cooperative sharing of information among systems by providing a unified location for resources and context data. The use of Service-oriented

Architecture (SOA) style and web service technology was a common theme in nine studies, as a means of implementing this approach.

Another study conducted by Valle 2021 performed an extensive analysis of sixty-five studies to conduct a literature review to identify and evaluate architectural strategies that may be implemented in integration projects to enhance interoperability within software-intensive systems.

In this research, multiple strategies were found to resolve different problems. The extensive list of strategies can be found in Appendix A.

Following this, the author conducted a series of surveys to gain a comprehensive understanding of the architectural strategies employed in the industry to promote interoperability. After performing, processing and analysing these surveys, the author concluded that the most relevant strategies to impose system architecture interoperability were the ones represented in Table 2.5.

Table 2.5: Architectural strategies suggested by "Architectural decision-making on interoperability in software-intensive systems" for each interoperability level

Patterns	**Technical**	**Syntactic**	**Semantic**	**Pragmatic**
Adapter	X	X	X	
Broker			X	X
Messaging	X	X		
Pipes and Filters	X			
Share Database	X			
Wrapper			X	
REST	X			
Microservices				X

The full details of these choices can be viewed in detail in the author's study at "Architectural decision-making on interoperability in software-intensive systems" to fully understand the reasons why some patterns were not considered.

Summing up, to effectively design and implement a system interoperability architecture, a comprehensive approach that addresses technical and non-technical issues such as semantic, syntactic, and organizational mismatches must be taken. It is important to consider all relevant domains in the process of achieving interoperability.

Keywords: API system interoperability architecture, Web API interoperability

2.2.2 Architecture APIs patterns and good practices

API architecture is a crucial aspect of building and maintaining a successful system. There are several software patterns and best practices that can be used to ensure that APIs are well-designed, scalable, and easy to use.

As previously discussed in Section 2.2.1, authors referred that, to achieve interoperability in software, various patterns and best practices must be adopted, such as the Adapter, Broker, Event Bus, Facade, and Scatter-gather, among others.

Another good practice for promoting interoperability in software systems is the use of a consistent architectural structure. The Layered Architecture pattern is a possibility as it separates the API into distinct layers, such as presentation, business logic, and data access, making it easier to maintain and update while also enabling component replacement without disrupting overall functionality (Richards 2015).

Regarding API's endpoints, Strengholt 2020 recommends the usage of standard HTTP/S methods (GET, POST, PUT, DELETE) and status codes. Additionally, it is advised to use a consistent naming convention for endpoints and to include appropriate documentation and error handling, including the correct HTTP codes and descriptive error message.

Additionally, it is a recommended practice to maintain the documentation current, as it is a vital factor in promoting interoperability between systems and ensuring that all parties involved have a clear comprehension of the system's functionality and capabilities, as previously mentioned.

Another compatibility good practice is related to versioning where a standard and transversal versioning approach should be defined for the system. Typically, the semantic versioning approach[20] could be followed. Minor changes result in a slight increment of the version number and significant, incompatible changes result in a major increase (Strengholt 2020).

2.2.3 Current system APIs interoperability solutions

As previously mentioned, there is not one single solution for managing the evolution of APIs while maintaining system interoperability. However, some solutions can assist in achieving this goal. Each solution employs different approaches and specifications to achieve system interoperability focusing on different concerns, so the decision depends on the problem to be addressed. These solutions will be discussed in further detail, highlighting the specific issues they address.

1. **Apache APIsix**

   Apache APISIX[21] is an open-source, high-performance, and scalable Full Lifecycle API that provides features like Load Balancing, Dynamic Upstream, Canary Release, Circuit Breaking, Authentication, Observability, and more (Apache Software Foundation 2022). It was designed to handle interoperability issues and provide a centralized solution for managing APIs.

   According to Wen 2022, one of the main advantages of APISIX is its ability to handle high concurrency. This makes it a great solution for performance, handling high traffic and ensuring that the APIs remain responsive and performant.

   Another advantage of APISIX is its support for dynamic routing, meaning that, it can automatically route requests to the appropriate backend service based on predefined custom rules (supports both path-based and domain-based routing). This allows for more flexibility and reduces the need for manual configuration. After presenting the aforementioned information, it is important to note that all of these configurations and more can be effectively managed through the use of APISIX's built-in dashboard, as illustrated in Figure 2.11.

[20] https://semver.org/
[21] https://apisix.apache.org/

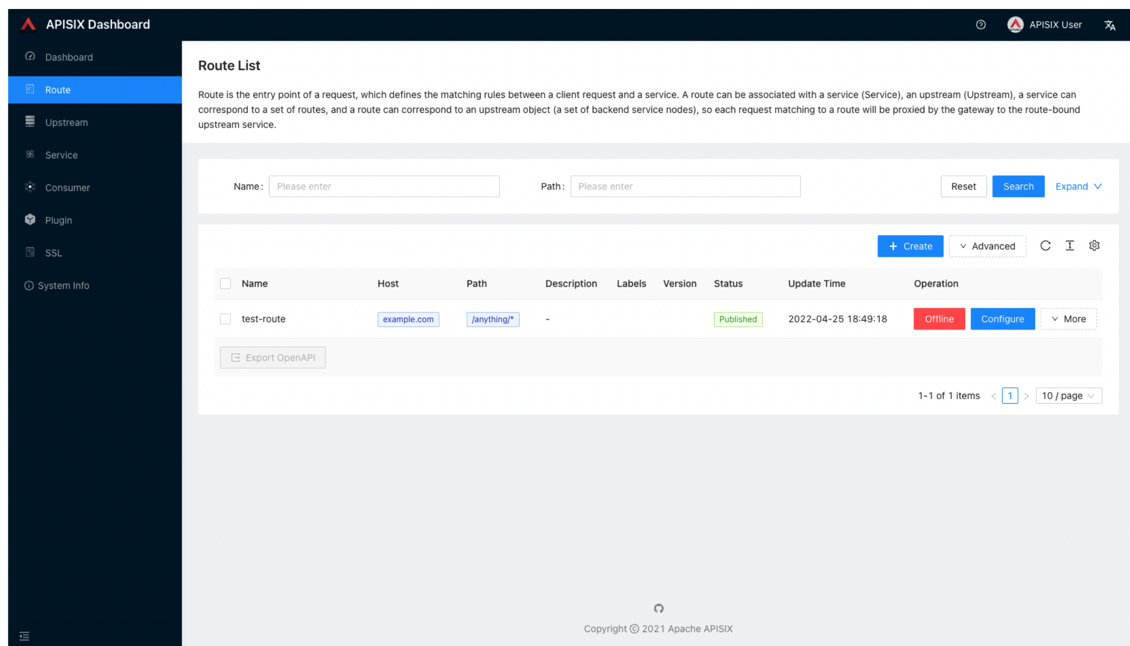


Figure 2.11: APISIX's dashboard (Apache Software Foundation 2022).

As stated by Apache Software Foundation 2022, APISIX also has a built-in service discovery feature, which allows it to automatically discover and register new APIs and middleware functions that can be used for load balancing, rate limiting, traffic management, and security concerns. These features make it easier to manage and scale the system as a whole as it minimizes the needed manual handling.

In terms of lifecycle management, the APISIX platform offers a comprehensive approach to API lifecycle management, including functionality for handling documentation, versioning, and a centralized API registry. This allows for easy management of the system's APIs, including the ability to view endpoints and their responses, enabling built-in API testing, and including the possibility to deprecate or evolve to a newer version an API (Apache Software Foundation 2022).

APISIX can handle multiple protocols and formats, making it a suitable solution for handling interoperability issues. Based on Wen 2022, it can handle multiple protocols and formats and it has multiple key features that make it a great solution for increasing systems' interoperability.

Having made all these statements, Figure 2.12 illustrates the high-level architecture of APISIX. It serves as a visual representation of the system's components and their interactions.

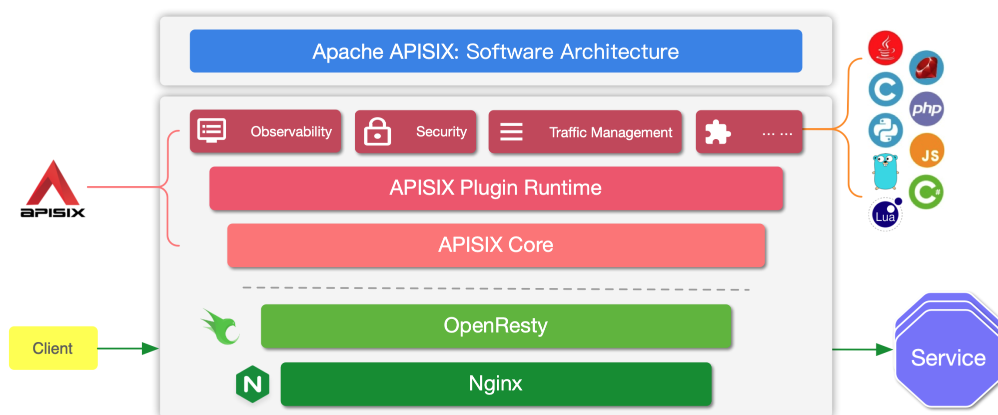


Figure 2.12: High-level overview of APISIX's software architecture (Apache Software Foundation 2022).

2. **AsyncAPI**

AsyncAPI is a specification for building APIs that handle asynchronous communication. It allows developers to describe and document the message-based interactions between services, making it easier to understand and consume these APIs. AsyncAPI uses the OAS as its foundation and extends it to support message-based communication. It is protocol-agnostic, so it can be used for APIs that work over any protocol (e.g., Advanced Message Queuing Protocol (AMQP), MQTT, WebSockets, Kafka, Streaming Text Orientated Messaging Protocol (STOMP), HTTP, etc) (AsyncAPI Initiative 2022).

According to AsyncAPI Initiative 2022, AsyncAPI also provides tools to generate documentation, code, test cases, client libraries and server skeletons in various programming languages. These abilities not only facilitate the developer's life during the APIs development process but also allow for flexible system evolution, as the APIs' documentation is created and updated automatically.

In terms of versioning and API lifecycle management, Garrote 2022 explains that AsyncAPI provides a way to version the specification and manage the lifecycle of APIs. This allows developers to create different versions and manage the transition from one version to another in a controlled and predictable way.

The utilization of features such as documentation and code generation, messages validation and versioning handling, requires the creation of an AsyncAPI specification document. This document, can be written in either YAML or JSON, serves as a machine-readable definition of the event-driven API and serves as the foundation for the aforementioned features. Figure 2.13 represents one example of an AsyncAPI specification document.

```yaml
asyncapi: '2.5.0'
info:
  title: Streetlights Kafka API
  version: '1.0.0'
  description: |
    The Smartylighting Streetlights API allows you to remotely
manage the city lights.
    ### Check out its awesome features:
    * Turn a specific streetlight on/off 🌃
    * Dim a specific streetlight 😎
    * Receive real-time information about environmental lighting
conditions 📈
  license:
    name: Apache 2.0
    url: https://www.apache.org/licenses/LICENSE-2.0
servers:
  test:
    url: test.mykafkacluster.org:8092
    protocol: kafka-secure
    description: Test broker
    security:
      - saslScram: []
defaultContentType: application/json
channels:
  smartylighting.streetlights.1.0.event.{streetlightId}.lighting.
measured:
    description: The topic on which measured values may be
produced and consumed.
    parameters:
      streetlightId:
        $ref: '#/components/parameters/streetlightId'
    publish:
      summary: Inform about environmental lighting conditions of
a particular streetlight.
      operationId: receiveLightMeasurement
      traits:
        - $ref: '#/components/operationTraits/kafka'
      message:
        $ref: '#/components/messages/lightMeasured'
  smartylighting.streetlights.1.0.action.{streetlightId}.turn.on:
    parameters:
      streetlightId:
        $ref: '#/components/parameters/streetlightId'
    subscribe:
      operationId: turnOn
      traits:
        - $ref: '#/components/operationTraits/kafka'
      message:
        $ref: '#/components/messages/turnOnOff'
```


Figure 2.13: AsyncAPI specification document example.

With the precious specification in mind, it is understandable that AsyncAPI was specifically designed to enhance the interoperability of APIs by providing a standardized method for describing the message-based interactions between services, thereby facilitating communication and understanding between different systems. Additionally, AsyncAPI can be integrated with different technologies such as Apache Kafka and RabbitMQ[22], making a possible solution for interoperability management (Garrote 2022).

3. **AWS API Gateway**

AWS API Gateway[23] is a fully managed service, offered by Amazon Web Services (AWS), that makes it easy for developers since it handles all of the heavy lifting that goes into managing one API. It provides services to create, publish, maintain, test, monitor, scale and secure APIs being these the reasons for it to be a possible solution to address system interoperability.

AWS API Gateway makes it possible to operate APIs by creating RESTful or WebSocket APIs that route HTTP type requests to the appropriate backend. It also

[22] https://www.rabbitmq.com/
[23] https://aws.amazon.com/api-gateway/

provides a way to authenticate and authorize clients and customize the request and response payloads, a way to monitor the API's performance, error handling, and security (Amazon Web Services 2023a). Figure 2.14 represents a typical AWS API Gateway architecture and its interactions.

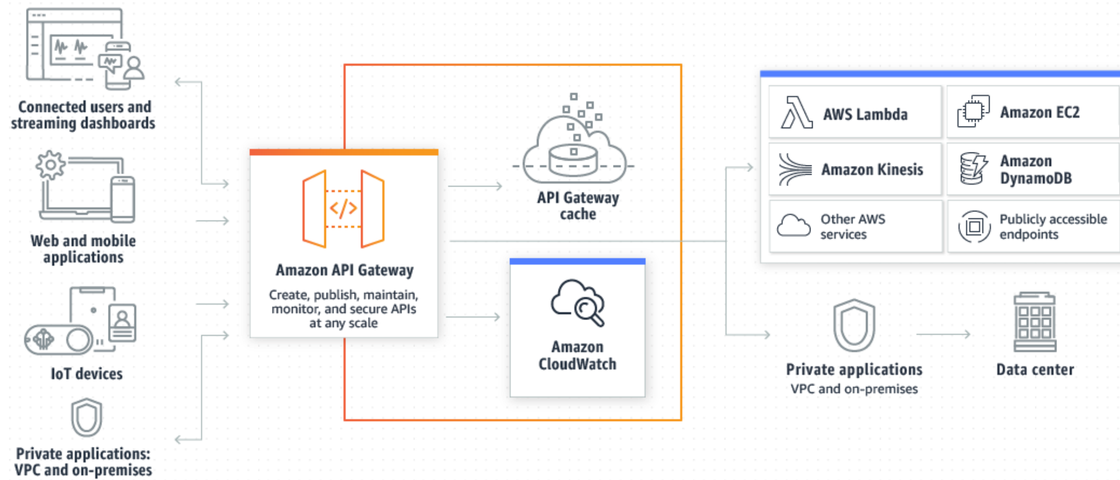


Figure 2.14: AWS API Gateway architecture (Amazon Web Services 2023a).

Regarding the systems' interoperability, AWS API Gateway is a great solution to handle communication between different systems, as it not only supports the integration with private applications but also offers the possibility to integrate with other AWS services such as AWS Lambda, Amazon SNS, Amazon SQS, and more which allows for easy integration with other systems.

Additionally, AWS API Gateway allows for versioning of the API enabling backwards compatibility, thus ensuring that the systems continue to work together even when changes are made. Furthermore, it offers monitoring and logging capabilities to track the APIs performance and usage, which is essential for maintaining system interoperability by quickly identifying and resolving issues (Amazon Web Services 2023a).

AWS API Gateway provides an interface for creating and updating APIs' documentation, as well as a flexible system for representing the content of the documentation in a variety of formats (e.g., JSON and YAML). This feature provides centralized documentation and the ability to associate documentation versions with specific API stages, to be later exported to external OpenAPI files, and distribute the files as a means of publishing the documentation (Amazon Web Services 2023b).

4. **Solutions comparison**

The objective of this study was to evaluate various solutions for managing system evolution and ensuring interoperability. A comprehensive research was conducted and several solutions were identified and analyzed. However, it was noted that some solutions did not address all the necessary aspects, while others were not well-suited for the problem at hand.

Given these considerations, Apache APIsix and AsyncAPI were selected as the initial focus of the study. These solutions are commonly used and provide an appropriate level of functionality to meet the requirements. Furthermore, it was deemed relevant to

explore solutions within the AWS environment, as the use of AWS is rapidly increasing in the company and the development of a solution there could have significant benefits.

Table 2.6 presents a comparison of the studied solutions, summarizing their differences and similarities. It helps to understand the strengths and weaknesses of each solution providing valuable insights into the solutions and being valuable for decision-making.

Table 2.6: Interoperability solutions comparison

Feature	Apache APISIX	AsyncAPI	AWS API Gateway
Documentation	Documentation can be handled in the tool's platform.	Generates API documentation based on the schema of the API.	Provides an interface for creating and updating APIs' documentation.
Versioning	Usage of different route definitions.	Usage of different schema versions.	Usage of stages variables.
Lifecycle Management	Managed through different route definitions and customization.	Managed through different route definitions and customization.	Usage of stages variables.
Schema Registry	Centralized API registry.	Usage of AsyncAPI schema files.	Usage of AWS tools (i.e. Amazon S3 or Amazon DynamoDB).
Testing	Done through Apache APISIX Dashboard.	Done through AsyncAPI files and a partner plugin.	Done through Amazon API Gateway Console.
Others	Generation of code, client libraries and server skeletons. Also allows message validation.	Monitoring, scaling and security of APIs, error handling, logging, authentication and authorisation and customization of requests and responses payloads.	Load balancing, dynamic upstream, canary release, circuit breaking, authentication, observability, and more.
Pricing	Open-source and free to use.	Open-source and free to use.	Pay-as-you-go pricing model based on API calls and data transfers.

Although the literature review primarily focused on Apache APIsix, AsyncAPI, and AWS API Gateway, it is important to mention that there are other viable options available in the market, but the choices were made based on several important factors that need to be highlighted to understand the rationale behind them.

The chosen solutions were carefully evaluated based on their suitability to address the problem at hand and meet the project requirements, while taking into consideration the specific environment in which they will be implemented.

First and foremost, it is essential to emphasize that the desired outcome of this work is the integration with an Event-driven architecture system. In this system, the Apache Storm framework is utilized as the event distribution system, and Apache Kafka is employed for streaming events between APIs (topologies written in Java or Scala). Hence, the solutions under analysis need to support Event-driven architectures.

Furthermore, the preference was given to open-source solutions due to their inherent flexibility and ability to adapt to diverse needs. Additionally, the cost-effectiveness of these solutions played a significant role in their selection. However, it is worth mentioning that Flutter Entertainment is currently exploring options within the AWS ecosystem, which led to the inclusion of AWS API Gateway as one of the solutions to be explored.

Considering all these factors, solutions such as such as WSO2 API Manager[24], Google's Apigee API Management[25], Microsoft's Azure API Management[26], Kong[27], KrakenD[28], were not considered in the comparison of the existing market solutions.

In conclusion, the comprehensive findings presented in Table 2.6 highlight a portion of the available solutions on the market, but they represent the ones that are thought to better address the problem of this study and its context. All three chosen solutions discussed in this study adequately address the required topics of investigation, although with distinct approaches. Subsequently, a detailed value analysis was conducted in Chapter 4 to identify the solution that is most likely to have the most value to the company, taking into consideration the specific case study conducted.

[24] https://apim.docs.wso2.com/en/latest/
[25] https://cloud.google.com/apigee
[26] https://learn.microsoft.com/en-us/azure/api-management/api-management-key-concepts
[27] https://konghq.com/
[28] https://www.krakend.io/

Chapter 3

Requirements and proposed solution

3.1 Requirements engineering

Requirements engineering is widely recognized as the first and most critical phase of the software engineering process. It is considered the key task of software development since ambiguous requirements could be the reason for software project failure or product defects. So, an effective requirements engineering process is essential for the success of the software project and it plays a crucial role in influencing productivity and product quality (Ramingwong 2012).

Regarding the concept of the requirement, two main types of requirements need to be considered: Functional Requirements (FRs) and Non-Functional Requirements (NFRs). FRs refer to the user needs that the system must fulfil and can be described as a precise demand that the system must be able to perform. On the other hand, NFRs refer to the properties and constraints under which a system must operate (Alsaleh and Haron 2016).

Both FRs and NFRs are essential for the success of a system project, and it is important to ensure that they are fully elicited and documented to avoid ambiguities, incorrectness, or contradictions. To be able to define these requirements, the *FURPS+* model was applied. The categories of this model are: for FRs, *F*unctionality; for NFRs, *U*sability, *R*eliability, *P*erformance, *S*upportability, and other types of NFRs that end up denoted by the *plus* symbol.

In this Section, both types of requirements have been taken into consideration according to the problem of this study, and taking into account the objectives stated in Section 1.3 and the intended value for each stakeholder involved in the project. In Section 4.3.1, each stakeholder's expectations are addressed in more detail.

3.1.1 System actors

The motivation behind this dissertation was submitted as a problem by the architecture team responsible for the Sportsbook Flutter's betting area. Therefore, the architects are one of the main actors in this project.

On the other hand, the other actors that will interact deeply with the solution are the developers, given that one of the purposes of this study is to speed up their work, namely by decreasing their manual operations and their components' problems.

3.1.2 Functional requirements

After conducting meetings with the principal stakeholders of this project to promote discussions of ideas and to gather necessary information regarding their vision and objectives, the necessary FRs were identified. The FRs were then documented in a Use Case diagram, which can be seen in Figure 3.1.

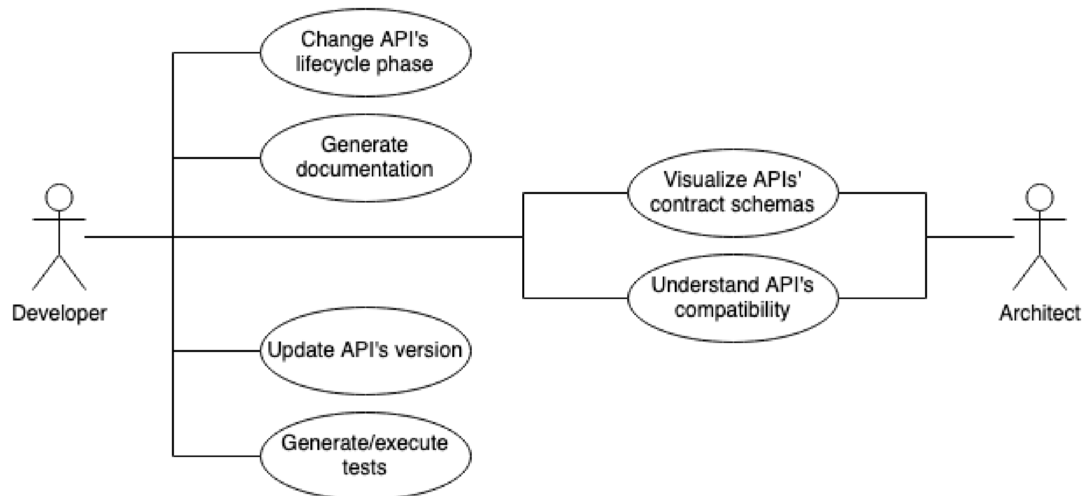


Figure 3.1: Use case diagram

Upon obtaining an initial understanding of the requirements, a detailed set of the FRs was created, as presented in Table 3.1.

Table 3.1: Functional requirements by stakeholders

Functional Requirement	Actor	Description
FR1: Change API's lifecycle	Developers	Developers want to be able to easily change the lifecycle of an API and notify clients about this change.
FR2: Generate documentation	Developers	Developers what to spend less time updating documentation. So they require a way to generate API's documentation automatically so that no manual operation is required.
FR3: Visualize APIs' contract schemas	Developers and Architects	Both actors want to be able to know API's contract without having to search for those schemas in multiple locations.
FR4: Understand API's compatibility	Developers and Architects	Developers and Architects want to understand the compatibility between API's based on their contracts and versions.
FR5: Update API's version	Developers	Developers want to have a uniform versioning strategy to have a clear view of how to update their APIs' versions and to know how other APIs' versions work.
FR6: Generate/execute tests	Developers	Developers want to be able to perform tests easily, without having to create multiple payloads or scenarios manually, so, the least manual effort the better. Additionally, if it is possible there could be a way to generate these tests based on the API's endpoints.

3.1.3 Non-functional requirements

Functional and non-functional requirements have distinct objectives. FRs dictate what the system should do and, on the other hand, NFRs reflect on how the system should operate to fulfil the FRs.

Having stated this, the NFRs identified from the meetings with the principal stakeholders are represented in Table 3.2.

Table 3.2: Non-Functional requirements by category

Non-Functional Requirement	Category	Description
NFR1: Easy to use	Usability	The solution must be clear and simple to implement.
NFR2: Support to different types of APIs	Supportability	The implemented project must be prepared to include different types of APIs.
NFR3: Adaptability	Supportability	The solution must allow the integration of new requirements/features, including integration with other frameworks to solve interoperability.
NFR4: Security in communication	Performance	The developed work shall process each request in the shortest time possible to promote a better development experience.
NFR5: Privacy	Plus (+)	The code implemented as the proof of concept must not be made public and should be within the private company's repositories.

3.2 Solutions choices for requirements

This Section intends to provide a detailed overview of the proposed solution for addressing the problem. To begin, it is important to have a clear understanding of the expected outcome interactions of the solution, so the diagram presented in Figure 3.2 provides a visual representation of those interactions.

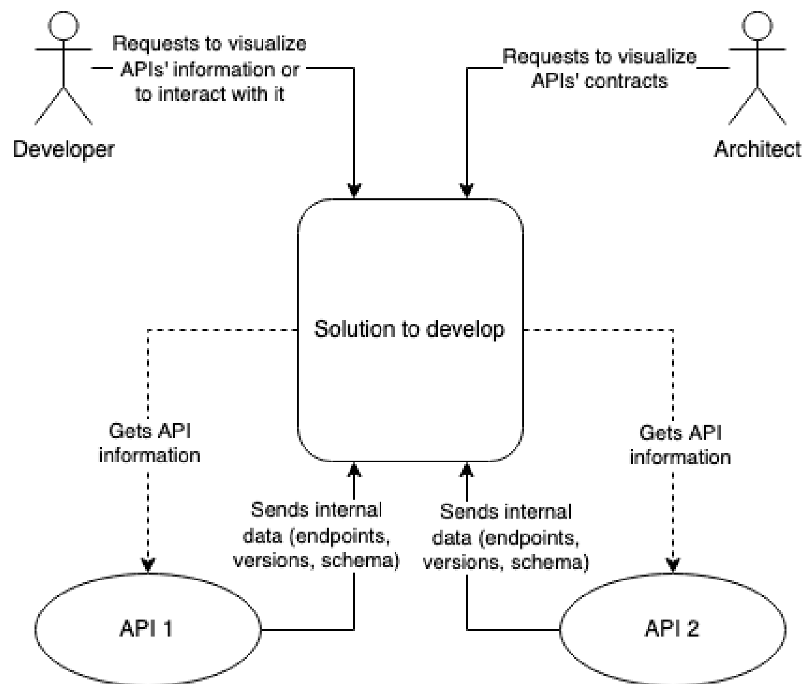


Figure 3.2: Solution's architecture high level draft diagram

In the upcoming sections, detailed explanations of the solutions selected for each objective of this project will be provided. This will enable a thorough understanding of the proposed solution.

3.2.1 Lifecycle management and versioning

The solution chosen for managing the API lifecycle and versioning incorporates two key decisions. Firstly, was adopted to use semantic versioning as the technique for versioning APIs. This method enables clear communication of the APIs' changes and supports backward compatibility.

Secondly, the versioning in the HTTP header approach was chosen to provide a cleaner API URI, allowing for easy migration to new versions. This option will help to maintain consistency since it will also require a custom HTTP header to notify clients of the current lifecycle state of an API version.

Using the combined solution of semantic versioning and versioning in the HTTP header it is promoted clear communication and ease of use for clients while ensuring consistent approaches and simplified API maintenance and it also enables efficient and effective management of the API throughout its lifecycle.

3.2.2 Documentation

AsyncAPI is an open-sourced and protocol-agnostic specification that describes and documents message-driven APIs from configurable files. It can provide several powerful features for documenting APIs, being this the reason why it was selected to be the solution for the documentation requirement.

This project includes the generation of documentation but it also supports the generation of schemas and response messages examples, integration with existing toolchains, and it is highly extensible and flexible since it supports custom plugins and the ability to generate documentation in a variety of formats.

These features mean that it allows fulfilling specific needs and makes it easier to understand and use APIs, as it facilitates documentation manipulation and reduces the risk of errors and misunderstandings.

3.2.3 Schema registry

It was decided to follow the Qarkus example and use Apache Avro to serialize records and store them in the Apicurio Registry. This solution will promote the compatibility between the producers and consumers and facilitate the evolution of their schemas over time.

The ability of Apache Avro allows for efficient data serialization and deserialization while the use of a centralized schema registry simplifies the process of managing and versioning schemas.

3.2.4 API testing

Regarding API contract testing and static analisys, the elected choice was Pact, a testing tool for API contracts that enable consumer-driven contract testing. It permits the consumer and provider of an API to define the terms of their interaction in a Pact file, which is then used to verify that the provider conforms to the terms agreed upon in the file.

Pact allows for the testing of complex interactions, such as those involving authentication and authorization, and provides detailed feedback about the areas that need attention in the API.

3.2.5 Conclusion

To summarize, Table 3.3 presents the decisions made for each requirement and explains the reasoning for those choices.

Table 3.3: Solutions choices per requirement

Requirement	**Solution**	**Reason/s**
API lifecycle management	Custom header notifying lifecycle stage	Consistency of choices and simple yet efficient and effective management.
API versioning	Semantic versioning and versioning in the header	Clear communication and level of detail.
API documentation	AsyncAPI	Generation of documentation, protocol-agnostic, highly extensible and flexible.
API schema registry	Apicurio Registry with Apache Avro	Efficient data serialization and deserialization and simplifies the process of managing and versioning schemas.
API testing	Pact	Allow schema validation, complex interactions and configurable test scenarios.

As evident from the analysis, there isn't a single solution that fulfils all the requirements comprehensively. Therefore, the decision for each specific requirement was made based on selecting the solution that was deemed most suitable for addressing the corresponding problem.

Chapter 4

Value analysis

In this Chapter, the value proposed by the solution developed in this dissertation will be analyzed. For this, a Value analysis (VA) will be performed by following the innovation process and using the New Concept Development (NCD). This Chapter focus on opportunity identification and analysis, idea genesis and selection (using Analytic Hierarchy Process (AHP) method) and finally the concept definition. In the end, the value proposition is defined.

4.1 Terminology

Over the years, the concept of value had several definitions depending on the context and various factors such as needs, desires, beliefs, and attitudes. In business, value is created through the exchange of goods or services that are deemed valuable and accepted by customers. The aim is to receive a reward for the value created, either within the enterprise or external network (Nicola, E. P. Ferreira, and J. J. Ferreira 2012).

With this said, the output of a VA ends up being a key resource to validate the success of a product or service. VA can be defined as a systematic and structured approach to improving the value of a product or service. The purpose of VA is to optimize the resources used in the creation of a product or service, reduce the costs and improve its quality, aiming to identify and eliminate activities that do not add value (Rich 2000).

The study by Rich 2000 highlights the existence of various forms of VA, including:

- **VA for Existing Products**: focused on optimizing the costs and performance of existing products without changing the product design.
- **VA for New Products**: applied to new products and is used to optimize the design and improve the value of the product. It considers factors such as cost, performance, quality, and customer requirements.
- **VA for Product Families - Horizontal Deployment**: focused on optimizing the entire product family, rather than just a single product. It considers the commonalities between products in the family to identify areas for cost optimization and performance improvement.
- **Competitive VA**: involves comparing the value of products from different suppliers or competitors. It helps to identify areas for improvement and to determine which supplier provides the best value for money.

Regardless of the type of VA being conducted, usually, there is an innovation process in place with well-defined stages. The innovation process facilitates the development and introduction of new ideas, products or services into the market and, generally, is divided into the three distinct areas shown in Figure 4.1: the **Fuzzy Front End (FFE)**, the **New Product Development (NPD)**, and **commercialization** (Koen et al. 2002).

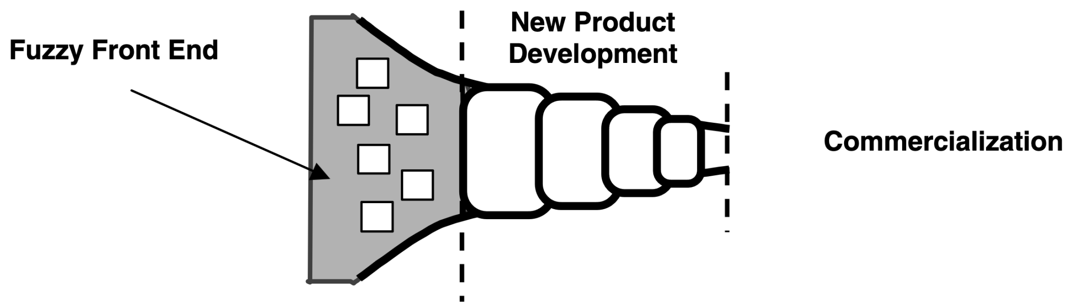


Figure 4.1: Innovation process stages (Koen et al. 2002).

The FFE, being the initial stage of the innovation process, provides significant benefits in enhancing the value, scope, and probability of success of various ideas prior to their progression into the development and commercialization phase. Considering this, the following Section 4.2 provides a deeper understanding of the FFE stage.

4.2 New concept development model

The FFE is the first stage of the innovation process, during which the company identifies new product ideas and evaluates the potential of each one. This stage is known as the phase where ideas are still in the early stages of development and are not fully defined. To help bring more clarity to this phase, a theoretical construct, the NCD Model represented in Figure 4.2, has been developed to provide a common language and improve the understanding of the FFE.

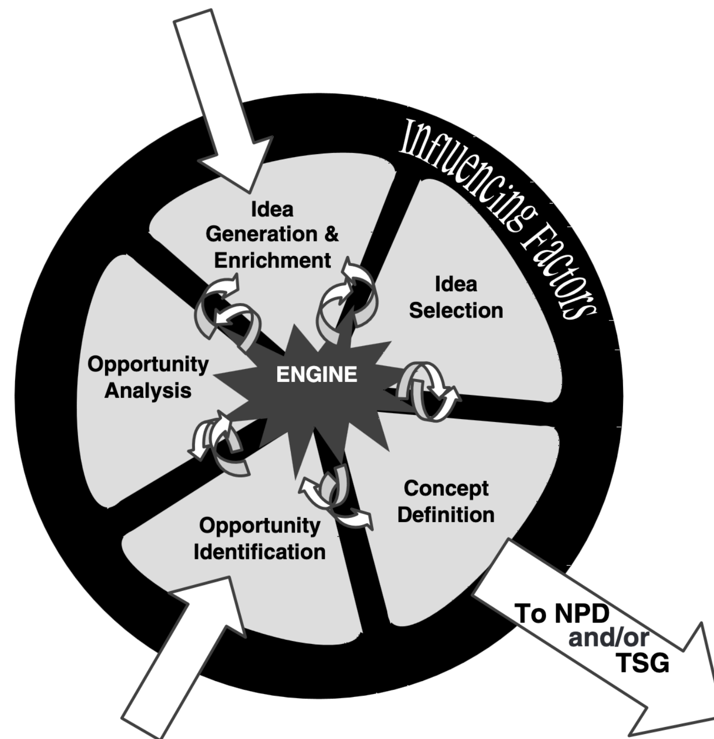


Figure 4.2: New concept development model (Koen et al. 2002).

The NCD Model, as proved by Figure 4.2, can be divided into three sections (Koen et al. 2002):

- **Engine (centre)** drives the five front-end elements and is fueled by the organization's leadership and cultural dynamics.
- **Five activity elements of the NCD (inner)** which are controlable and are part of FFE. These elements are opportunity identification, opportunity analysis, idea generation and enrichment, idea selection, and concept definition and will be covered in Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4, and 4.2.5 respectively.
- **Influencing factors (exterior)** play a crucial role in the innovation process, but their impact cannot be fully controlled since they are influenced by external factors (i.e. law, government policy, competitors).

It is important to state that the NCD model of Figure 4.2, has a circular shape to indicate that the flow of ideas is intended to circulate through the elements. Based on (Koen et al. 2002), despite the potential for looping back causing delays in the FFE phase, it ultimately reduces the overall product development and commercialization timeline by clarifying the requirements, potential risks, and business plan.

4.2.1 Opportunity identification

Opportunity identification is a crucial element in the NCD model where the organization identifies potential areas of growth and development. This phase considers both business and technological opportunities that align with the company's goals and objectives to ultimately

allocate resources to new areas of market growth, operating effectiveness, and efficiency (Koen et al. 2002).

In the past five years, Flutter Entertainment Group has experienced substantial growth through a series of acquisitions including FanDuel, Sky Betting and Gaming, Adjarabet, Stars Group, Tombola and Sisal. This has resulted in the need for the management of multiple platforms as a unified entity in order to ensure their compatibility and interoperability. In response to this challenge, Flutter Entertainment has initiated a project aimed at enhancing interoperability across all its brands and standardizing its platform.

This presents an opportunity for this study, as the goals of the project undertaken by Flutter Entertainment align with the objectives of this study which are to address the incompatibility issues and improve the interoperability between systems.

In addition, the trend migration to AWS also presents an opportunity as it increases the inclination to optimize components and improve system interoperability

4.2.2 Opportunity analysis

After identifying potential opportunities, it is necessary to conduct a comprehensive evaluation to determine their viability. The process of opportunity analysis involves making preliminary assessments of both technology and market condition (Koen et al. 2002).

The implementation of a solution to manage the evolution of APIs and enhance system interoperability has the potential to bring numerous benefits, however, it is necessary to conduct a more in-depth analysis to establish the opportunity in detail.

As previously noted, the presence of multiple brands across multiple locations within Flutter Entertainment poses challenges in terms of system compatibility and maintenance.

According to Flutter Entertainment's *Annual Report and Accounts 2021*, the average monthly players showed a twenty-three per cent increase compared to 2020 and the group's revenue experienced growth of fifteen per cent as is possible to confirm in Figure 4.3.

Group

Unaudited Adjusted Pro forma	**FY 2021 £m**	FY 2020 £m	Change %	CC Change %
Average monthly players ('000s)	**7,619**	6,174	*+23%*	
Sports revenue	**3,774**	3,000	*+26%*	*+27%*
Gaming revenue	**2,262**	2,264	*—%*	*+4%*
Total revenue	**6,036**	5,264	*+15%*	*+17%*
Cost of sales	**(2,262)**	(1,782)	*+27%*	*+29%*
Cost of sales as a % of net revenue	***37.5%***	*33.8%*	*+360bps*	*+350bps*
Gross profit	**3,774**	3,483	*+8%*	*+11%*
Sales and marketing	**(1,508)**	(1,130)	*+33%*	*+38%*

Figure 4.3: Flutter Entertainment plc 2021 growth (Flutter Entertainment plc 2022).

This means duplication of issues across various systems and the need to coordinate with multiple teams from different brands when changes are required which not only incur additional costs in terms of human resources, but it also poses a risk to the overall business, as issues affecting clients could impact multiple brands. Given these challenges, the company

has recognized the need for a unified approach to system evolution and improvement of interoperability.

This recognition has driven the creation of a strategy for improving the overall systems at Flutter and presents an opportunity for this thesis to address the issue. The topic of this thesis was a request from the architecture team at Flutter, further emphasizing the importance and market opportunity of this research.

4.2.3 Idea generation and enrichment

The concept of idea generation and enrichment concerns the beginning, growth, and refinement of a specific idea. The idea undergoes various changes and modifications during the various elements of the NCD model until its final realization.

In this phase, it is crucial to take into account a diverse range of possible solutions rather than solely focusing on a single idea. This can be achieved by considering various perspectives and evaluating different types of solutions. This process may take the form of a structured approach, including brainstorming sessions and the compilation of new ideas, to stimulate the organization to generate fresh concepts for the previously identified opportunity (Koen et al. 2002).

The process of idea generation for the theme of this thesis involved utilizing the brainstorming technique to develop concepts related to the previously identified opportunity. During this session, some questions were made:

- What problems are being experienced that difficult APIs evolution managed?
- What can be done to automate tasks that currently require manual intervention?
- What solutions exist in the market that could solve the problem?
- How easily can the solution be implemented?
- How can the solution solve the existing interoperability issues?

After examining and deliberating on the previous questions, it became evident that it was necessary to define criteria for the success of this study. To begin with, it was critical to gather all relevant information about the fundamental theoretical concepts, to achieve a thorough understanding of the subject matter under research. It was then necessary to study the best solution to the problem, either an available solution on the market or one developed within the scope of this work. Lastly, a proof of concept should be developed to test and validate the solution. These criteria, which originated the objectives of this study, are resumed below:

- **C1:** API lifecycle management
- **C2:** Possibility to generate API documentation
- **C3:** Existence of an API schema registry
- **C4:** API versioning standardisation
- **C5:** API testing tools
- **C6:** Acquisition price/costs regarding implementation time

Having conducted a comprehensive search for potential solutions (Section 2.2.3), and developed a guide that represents a possible alternative (Section 3.2), the resultant solutions that effectively address the problem are listed below:

- **S1:** Apache APISIX
- **S2:** Async API
- **S3:** AWS API Gateway
- **S4:** Developed guide

4.2.4 Idea selection

Selecting the right ideas to pursue is a crucial challenge faced by organizations. A successful selection is vital for the future prosperity of the business (Koen et al. 2002).

The process of evaluating and determining the most valuable ideas among the proposals made in the previous stage is crucial and so different methods and techniques can be applied to identify the most promising idea for further examination, but the one that will be used in this study is the AHP.

The AHP was developed by Thomas L. Saaty in 1980 and has been widely used since. It is a method for making complex decisions by evaluating and prioritizing multiple factors that influence the final decision.

The process of this method incorporates both quantitative and qualitative factors in a multi-criteria decision method which evaluates the alternatives based on specified criteria. The AHP approach breaks down the decision-making process into a hierarchical tree structure, simplifying the evaluation process and making it easier to comprehend (Thomas L. Saaty 1980).

Having in mind the problem of this thesis, the idea generation results of Section 4.2.3, the researched solutions of Section 2.2.3 and the designed solution of Section 3.2:, Figure 4.4 represents the hierarchy tree of AHP.

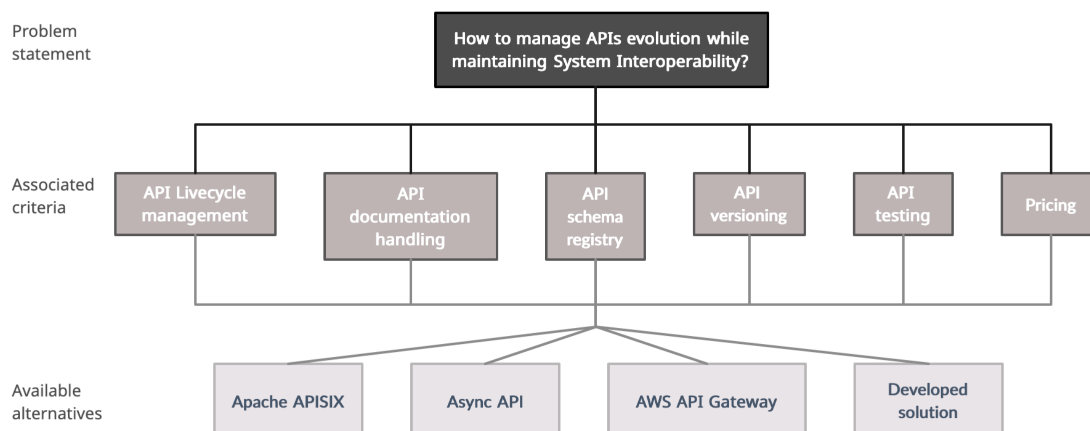


Figure 4.4: AHP Hierarchy Structure.

The next stage of the AHP method involves constructing a matrix for evaluating the criteria by assigning a value through the utilization of the Saaty Scale. This scale is represented in Table 4.1.

Table 4.1: Saaty fundamental scale (Thomas Lorie Saaty 1990).

Importance Level	**Definition**	**Description**
1	Equal Importance	The two activities contribute equally to the objective.
3	Weak importance	One activity is slightly more important than the other.
5	Strong importance	One activity is strongly more important than the other.
7	Demonstrated importance	One activity is strongly more important than the other and it is demonstrated in practice.
9	Absolute Importance	The evidence favours one activity with the highest possible order of affirmation.
2,4,6,8	Intermediate values	When a compromise is needed.

Since the values for the pairwise comparison matrix are now standardized, it is possible to create the criteria pairwise comparison matrix, represented in Table 4.2.

Table 4.2: Criteria pairwise comparison matrix.

	C1	**C2**	**C3**	**C4**	**C5**	**C6**
C1	1	1/7	1/3	1/5	3	3
C2	7	1	5	3	9	9
C3	3	1/5	1	1/3	5	5
C4	5	1/3	3	1	7	7
C5	1/3	1/9	1/5	1/7	1	1
C6	1/3	1/9	1/5	1/7	1	1
Sum	16,67	1,90	9,73	4,82	26,00	26,00

With the criteria pairwise comparison matrix table, the next phase involves determining the relative priority of each criterion. This is achieved by normalizing the comparison matrix and calculating the average of each line in the normalized matrix, which yields the relative priority, or priority vector, as represented in Table 4.3.

Table 4.3: Normalized criteria pairwise comparison matrix and relative priority.

	C1	C2	C3	C4	C5	C6	Relative priority
C1	0,0600	0,0753	0,0342	0,0415	0,1154	0,1154	0,073628
C2	0,4200	0,5268	0,5137	0,6225	0,3462	0,3462	0,462549
C3	0,1800	0,1054	0,1027	0,0692	0,1923	0,1923	0,140313
C4	0,3000	0,1756	0,3082	0,2075	0,2692	0,2692	0,254963
C5	0,0200	0,0585	0,0205	0,0296	0,0385	0,0385	0,034274
C6	0,0200	0,0585	0,0205	0,0296	0,0385	0,0385	0,034274
Sum	1,0000	1,0000	1,0000	1,0000	1,0000	1,0000	1,000000

Upon calculating the relative priority, it becomes imperative to perform a consistency check on the criteria being evaluated and, to do this, it is required to calculate the Consistency Ratio (CR). To calculate its value a series of steps must be followed to, ultimately, get a CR result below 0,1, indicating that the relative priorities are consistent and therefore valid.

1. **Priority vector matrix (Ax):**

   The initial comparison matrix (Table 4.2) must be multiplied by the priority vector (*Relative priority* column of Table 4.3) to obtain the resulting matrix (Ax) represented in Table 4.4.

   Table 4.4: Priority vector matrix.

	Resulted value
C1	0,4431
C2	3,0613
C3	0,8814
C4	1,6781
C5	0,2090
C6	0,2090

2. **Intermediate vector:**

   By dividing the priority vector matrix (Table 4.4) by the values of the column *Relative priority* of Table 4.3 an intermediate vector matrix is calculated, as represented in Table 4.5.

   Table 4.5: Intermediate vector matrix.

	Resulted value
C1	6,018268
C2	6,618392
C3	6,281924
C4	6,581590
C5	6,097070
C6	6,097070

3. **Self Value (λmax)**:

λmax is obtained by dividing the priority vector matrix by the values of the column *Relative priority* of Table 4.3 and then calculating the average, so it can be calculated using the intermediate vector matrix of Table 4.5:

$$\lambda max = \frac{Ax}{X}$$

So, having in mind the results of the previous step:

$$\lambda max = \frac{6,018268 + 6,618392 + 6,281924 + 6,581590 + 6,097070 + 6,097070}{6}$$

$$\lambda max = 6,28239$$

4. **Consistency Index (CI):**

Consistency Index (CI) is calculated by the formula:

$$CI = \frac{\lambda max - n}{n - 1}$$

In this formula, n represents the number of criteria and, having the value of $\lambda max = 6,28239$, CI can be calculated by:

$$CI = \frac{6,28239 - 6}{6 - 1}$$

$$CI = 0,056477$$

5. **Consistency Ratio (CR):**

To determine the CR the CI is divided by a value known as the Random Index (RI), which varies based on the number of criteria being considered. The possible values for the RI are listed in Table 4.6.

Table 4.6: Random index values.

Number of criteria	1	2	3	4	5	6	7	...
RI	0,00	0,00	0,58	0,90	1,12	1,24	1,32	...

As the number of criteria for this study was 6 CR can be calculated by:

$$CR = \frac{CI}{IR}$$

$$CR = \frac{0,056477}{1,24}$$

$$CR = 0,045546$$

Having calculated a $CR = 0,045546$, it was concluded that the results obtained so far are consistent, as the CR value is below the threshold of 0,1. The next step involves conducting a similar analysis for the criteria (C1, C2, C3, C4, C5 and C6) concerning the solutions (S1, S2, S3 and S4). This process involved the generation of multiple tables, which can be reviewed in full detail in Appendix B.

The final step is to obtain the comparison matrix between criteria and solutions. This is done by multiplying the relative priority obtained from each solutions criteria's comparison matrix (Tables B.2, B.6, B.10, B.14, B.18 and B.22) with the relative priority of the initial comparison matrix (Table 4.3). This calculation is represented in Figure 4.5.

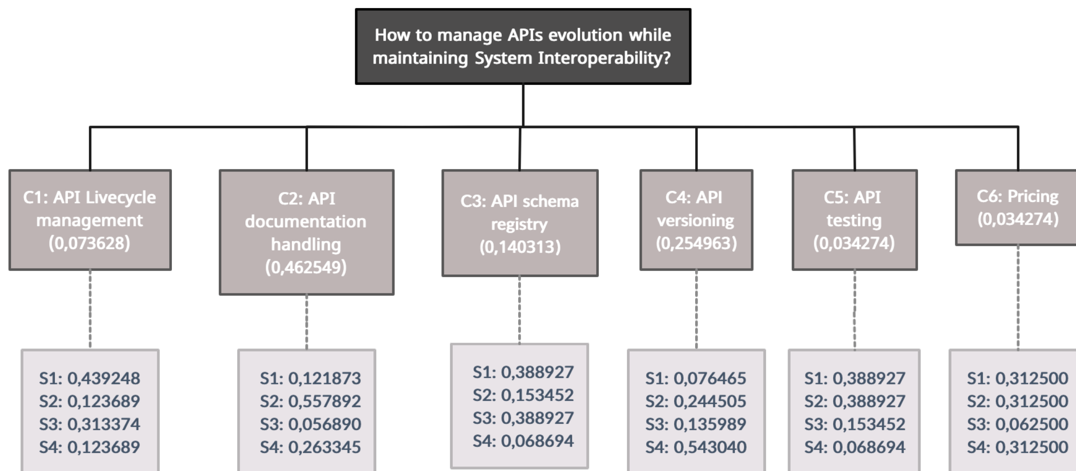


Figure 4.5: Comparison matrix between criteria and solutions.

The final determination of the priority of each solution was made based on the acquired data represented in Figure 4.5, which was carried out as follows:

$$\begin{bmatrix} 0,439248 & 0,121873 & 0,388927 & 0,076465 & 0,388927 & 0,312500 \\ 0,123689 & 0,557892 & 0,153452 & 0,244505 & 0,388927 & 0,312500 \\ 0,313374 & 0,056890 & 0,388927 & 0,135989 & 0,153452 & 0,062500 \\ 0,123689 & 0,263345 & 0,068694 & 0,543040 & 0,068694 & 0,312500 \end{bmatrix} \times \begin{bmatrix} 0,073628 \\ 0,462549 \\ 0,140313 \\ 0,254963 \\ 0,034274 \\ 0,034274 \end{bmatrix} =$$

$$= \begin{bmatrix} 0,186821 \\ 0,375071 \\ 0,146032 \\ 0,292076 \end{bmatrix} \approx \begin{bmatrix} 0,1868 \\ 0,3751 \\ 0,1460 \\ 0,29211 \end{bmatrix}$$

With this analysis, it was possible to conclude that the alternative of the thought solutions with the most value is the solution **S2 - AsyncAPI** with a percentage of 37,51 compared to the others.

4.2.5 Concept definition

The final aspect of the NCD model focuses on establishing clear qualitative and quantitative data for future decision-making purposes. This includes, among other things, the establishment of guidelines related to objectives, corporate vision, market demands, opportunity and benefits size, potential risks, financial impact, among others (Koen et al. 2002).

Taking into account the object of this study, which involves identifying strategies for managing APIs evolution, the initial phase includes compiling and analyzing relevant data. This will require research to gather information from published sources on the subject matter. Upon collecting the necessary data, solutions to the problem at hand must be collected and an effective response develop.

The most valuable idea identified during the idea selection phase in Section 4.2.4 will be explored in the following Section 4.3, to demonstrate its value.

4.3 Value

The concept of value plays a crucial role in management. The value of a product is influenced by the desire of its owners or buyers to retain or acquire it, thereby introducing a subjective element to the evaluation of value (Neap and Celik 1999).

With this said, it is vital to ensure that the perceived advantages and sacrifices of a product so that the consumer knows the solution is worthwhile. This analysis can be reviewed in detail in Section 4.3.1.

Additionally, a value proposition is conducted in Section 4.3.2 as it is an indispensable tool to further review the solution value.

4.3.1 Perceived value

The concept of perceived values has been discussed through time and, depending on the author, can have multiple definitions but, generally it consists of interaction between perceived advantages and sacrifices and if the advantages outweigh the sacrifices, it lets to believe for a consumer that the solution is worthwhile.

The concept of perceived value has been extensively studied and has suffered various interpretations by different authors. However, it generally involves the relationship between perceived benefits and sacrifices, with the positive points having to surpass the negative in order for a consumer to consider the solution to be valuable (Lapierre 2000).

With this in mind, to determine the perceived value of the solution proposed in this thesis for its customers, it is crucial to first identify the key stakeholders and their respective perspectives on the perceived benefits and drawbacks of the solution, as represented in Table 4.7.

Table 4.7: Perceived value - benefits and sacrifices.

Stakeholders	Benefits	Sacrifices
Developers	• Less time spent on documentation • Less time spent dealing with APIs' compatibilities • Easier to modify APIs • Easier to test APIs	• Time spent on understanding the solution and implementing it
Architects	• Single point of truth for APIs' standards and rules • More scalable and reliable system • Less time spent understanding the overall system interactions	• Time spent to plan changes regarding the implementation of the solution
Company	• Less money spent on incidents and synchronization meetings between API's owners and clients	• Money spent on developers' time to implement the solution

4.3.2 Value proposition

According to Pigneur and Osterwalder 2010 value proposition is a representation of the solution a company offers to solve a customer's problem. It describes the unique benefit a company provides to its customers and how it differentiates itself from its competitors. The Value Proposition Canvas highlights the relationship between two building blocks of Osterwalder's comprehensive Business Model Canvas: Customer Segment and Value Proposition (Pigneur and Osterwalder 2010).

The Customer Segment defines the different groups of people or organizations an enterprise aims to reach and serve. Customers are the heart of a business model, and grouping them into distinct segments helps better satisfy their needs. With this in mind, this building block is divided into three topics:

- **Customer Jobs**: tasks that customers want to be fulfilled.
- **Pains**: issues, feelings, and dangers expressed by customers.
- **Gains**: benefits towards customer needs and expectations.

On the other hand, the Value Proposition building block describes the collection of products and services that provide value to a specific Customer Segment. It is the reason customers choose a particular company over others, satisfying a customer's problem or need by outlying a selected bundle of benefits. This building block is structured into three distinct areas:

- **Products and Services**: alleviate customer pains and enhance customer satisfaction.

- **Pain Relievers**: outlining how customer's pains eased.
- **Gain Creators**: how the product or service creates customer gains and adds value to the customer, normally responding to some of the expected gains by the user.

In order to effectively articulate the value that the proposed solution for the problem offers to the customers, a tool, known as the Value Proposition Canvas was designed, as outlined in Figure 4.6.

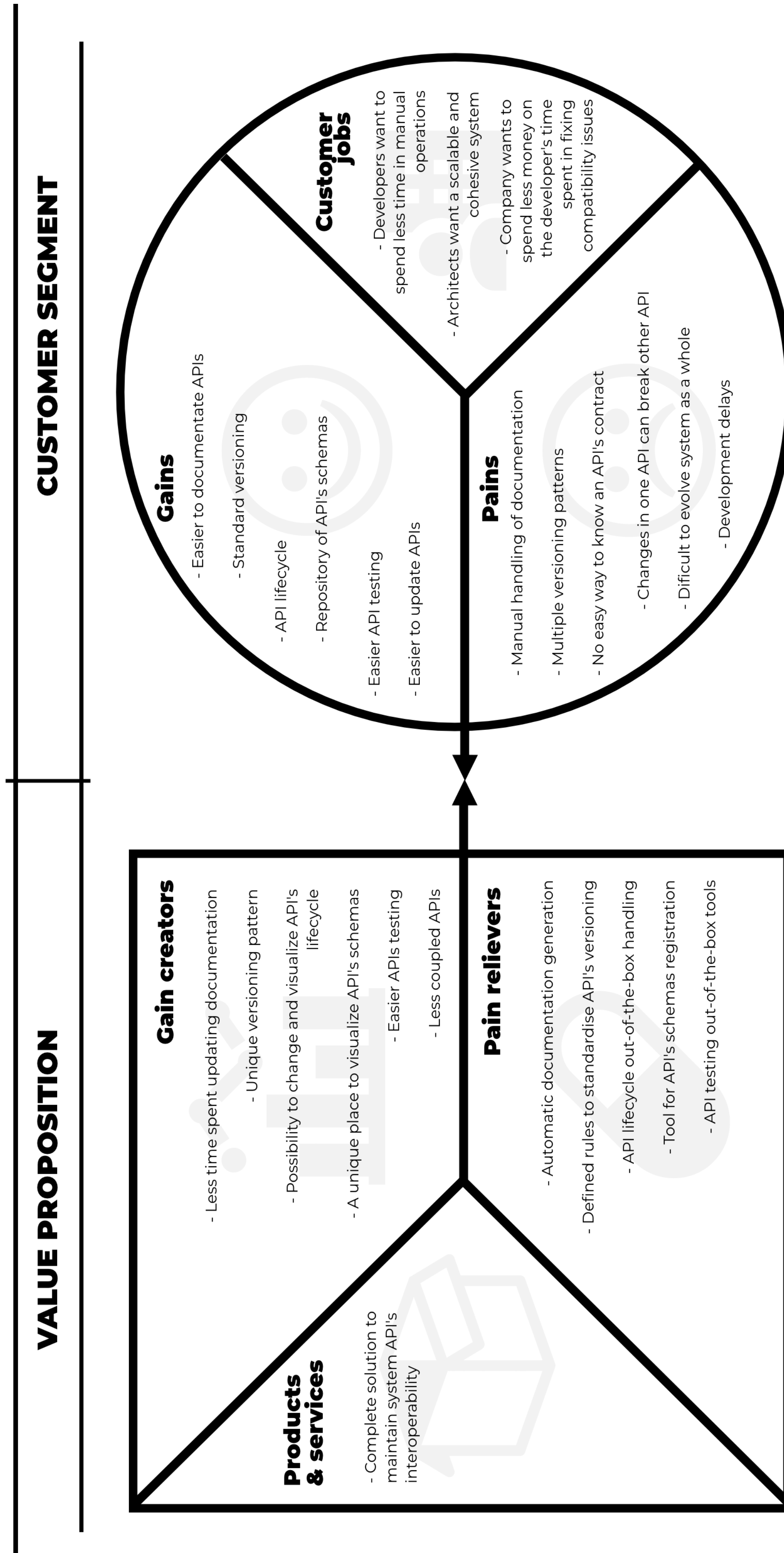


Figure 4.6: Value proposition canvas.

4.4 Value analysis conclusions

Based on the results presented in Section 4.2.4, where solution **S2 - AsyncAPI** was found to be the most valuable, it may seem counterintuitive that the developed solution is not the best mean for the problem.

To explain this result it is important to note that the solutions found in the market address all the criteria although with varying degrees of detail. Some may deal with certain criteria in more detail or with the characteristics of others, but, generally, all solutions have a solution for each criterion.

Additionally, the comparison matrices created during the evaluation process took into account the effort required to implement a solution for a given criterion, such as the ability to test and/or generate tests for APIs, which all solutions in the market had out of the box. The reason for this basis is the fact that implementing a similar feature in a custom solution would require a significant investment in time, effort, and resources.

Another contributing factor to this outcome may be attributed to the inherent complexity of each criterion and the multiple ways in which they can be addressed and solved. As a result, designing a solution that is flexible enough to tackle such complex issues and accommodate a wide range of API types is inherently challenging. This consideration was also factored into the value analysis study, which evaluated the solutions.

After considering the aforementioned arguments, the solution deemed most valuable in the evaluation was selected for implementation: **S2 - AsyncAPI**.

Chapter 5

Development, Testing and Delivery

This chapter presents a comprehensive overview of the work implemented during the proof of concept. It begins with the introduction of the chosen APIs for the proof of concept, namely IMG and TMT, along with their respective responsibilities within the system and the tennis flow. This is followed by descriptive User Stories (US) to enhance understanding of the developed features. Furthermore, a complete contextualization of both APIs within the system is provided through both container and deployment diagrams.

Next, the actual code implementation is discussed, specifically focusing on the creation of AsyncAPI schema specification files for both APIs. A thorough overview of the AsyncAPI schema specification details is provided, and the produced document for the TMT API is showcased, demonstrating how all the objectives were addressed and fulfilled.

Subsequently, the testing phase is explored, where the Microcks tool was applied to generate tests and validate API contracts. The details of this tool, including its functionality and the necessary steps taken to make it work effectively, are presented.

Lastly, the precautions taken during the project delivery are outlined, including an explanation of the GitHub workflow created and the AsyncAPI guide. The GitHub CI pipeline was established to streamline the validation process and automatically generate documentation, ensuring it remains up to date with minimal manual effort. The AsyncAPI guide aims to assist developers in getting started with AsyncAPI, understanding the project's implementation, and following best practices for maintaining interoperability.

5.1 Development

In preparation for the implementation phase, careful consideration was given to the selection of APIs that would be part of the proof of concept work. The chosen APIs, namely TMT and IMG, play a pivotal role in the Mapping Tennis flow, enabling the provision of tennis events to multiple brands within the Flutter Entertainment group.

These APIs not only communicate with each other but also interact with various other APIs within the system, with the exception that TMT solely consumes messages from IMG but is consumed by others. For a more comprehensive understanding of these APIs, additional context can be found in Section 5.1.1.

The decision to include TMT and IMG APIs in the proof of concept was driven by their critical importance within the organization. They form an integral part of the long-standing mapping tennis flow, which has undergone numerous changes over time due to evolving information providers. Consequently, the logic within these APIs has become increasingly

complicated, making it challenging to comprehend the internal domains and the payloads being consumed and produced.

Furthermore, given the presence of multiple clients and diverse input and output payloads associated with these APIs, it was deemed essential to include them in the proof of concept. By focusing on them, a more comprehensive evaluation of the proposed solution's feasibility and effectiveness can be achieved.

With the APIs chosen, it was created a well-organized list outlining the work that was required, the expected outcomes, and the acceptance criteria for each task.

To structure the work in a familiar and standardized manner, USs were used from the developers' perspective, following the established practices within the organization. As a result, a total of eight US were identified and incorporated into a development backlog, which is presented in Table 5.1. This backlog serves as a roadmap for the development tasks, systematically guiding the implementation process.

Table 5.1: Development backlog.

#	User Story	Acceptance Criteria	FR/ NFR
1	*As a* Developer *I want to* investigate AsyncAPI specification *So that* we can easily understand know to use it	• Have read documentation on how to start with AsyncAPI • Have made a POC on how to use AsyncAPI	*NFR1*, *NFR2*
2	*As a* Developer *I want to* create a repository with all the payloads of the APIs *So that* we can easily have access to the client and consumer APIs' payloads	• Created a repository to work as a schema registry endpoint • Have identified necessary API's schemas	*FR3*, *FR4*, *NFR5*
3	*As a* Developer *I want to* Implement IMG's AsyncAPI schema spec file *So that* we can save time updating documentation manually	• Have implemented the AsyncAPI schema spec file for IMG • Have Doc automatically generated from the file	*FR1*, *FR2*, *FR5*, *NFR4*, *NFR5*
4	*As a* Developer *I want to* Implement TMT's AsyncAPI schema spec file *So that* we can save time updating documentation manually	• Have implemented the AsyncAPI schema spec file for TMT • Have Doc automatically generated from the file	*FR1*, *FR2*, *FR5*, *NFR4*, *NFR5*
5	*As a* Developer *I want to* create makefile with the basic commands of AsyncAPI *So that* we can validate and generate doc in a single command	• Have identified relevant commands to implement • Have created a working makefile for both APIs	*FR2*, *NFR1*, *NFR3*
6	*As a* Developer *I want to* create a documentation AsyncAPI How to *So that* we have a guide to help understand how AsyncAPI schema works	• Have created the document as a guide on how to • Document should have sections Introduction, AsyncAPI Structure, How to Generate Doc, API's version standardisation, Lifecycle Management, Schema Registry	*FR2*, *NFR1*
7	*As a* Developer *I want to* study how AsyncAPI's tests generation work *So that* we can easily implement the necessary changes to generate tests	• Have read documentation on how to generate tests with AsyncAPI • Identified necessary commands/tools • Have made a POC on how to use AsyncAPI	*FR6*, *NFR1*, *NFR3*
8	*As a* Developer *I want to* implement necessary changes to generate tests *So that* we can save time creating manual tests to validate APIs	• Tests can be generated from AsyncAPIs schema spec file • Added necessary commands to makefiles • Documentation is updated if needed	*FR6*, *NFR1*, *NFR3*, *NFR5*

All the implemented code has been securely stored within the organization's GitHub repositories. In cases where committing code to the organization's repositories was not possible, private repositories were utilized as an alternative. This approach ensures that all the developed work remains private and accessible only to employees of Flutter Entertainment.

Throughout the development process, strict adherence to the organization's rules and best practices was maintained. This ensured seamless integration of the work into the organization's existing infrastructure and processes, promoting simplicity and effectiveness in the delivery phase.

5.1.1 APIs design

To gain a comprehensive understanding of the responsibilities and system architecture of both the TMT and IMG APIs, two diagrams were created following the principles of the C4 model[1]. This model comprises a collection of diagrams that provide varying levels of detail about applications. Among these levels, the container level will be focused on.

The container diagram presents an overview of the software architecture, illustrating the distribution of responsibilities within it. Furthermore, it can highlight the technological choices and demonstrate how the containers communicate with each other. This diagram, which is both straightforward and technology-focused, proves beneficial for software developers as well as support and operations personnel (2023).

Concerning the IMG and TMT APIs, both are integral to the tennis mapping flow and are responsible for generating tennis matches available on Flutter Entertainment websites. This includes managing competitions, tournaments, players, and live match information. In this context, the IMG API consumes data from two APIs, namely AFS and SPC. These APIs act as collectors, retrieving information about upcoming events and live data from external providers (IMG and RB), and forwarding it for further processing.

Upon processing this data, IMG API makes it accessible through four Kafka topics. One topic is dedicated to informational data, containing detailed information about future matches, including the competition, tournament, date, teams, and other relevant details of the tennis match. Additionally, the three live topics offered information regarding live details of a tennis match such as the points scored, sets won, breaks, faults and more.

On the other hand, the TMT API functions as a middleware, matching tennis events created within Flutter's system with those from external providers. It utilizes a context database to store entries from both sources: the InfoData topic from IMG API to determine which matches will be offered by external providers, and the matches made available internally through GSM. When a match occurs, TMT API publishes a mappings message to a Kafka topic, which is then read by GMW, enabling the match to be offered within Flutter's system with all available details.

Furthermore, the live topics are utilized by APIs within the Exchange betting platform to provide customers with access to the specific betting product, Exchange. These topics are also used by FVS, a tool that was designed to facilitate easy access to every payload generated by IMG API, assisting in bug tracing and incident investigations.

[1] https://c4model.com

The diagram in Figure 5.1 illustrates a container diagram based on the C4 model, which serves as a valuable diagram for analyzing the overall system design, data flow, and communication between TMT, IMG and their clients.

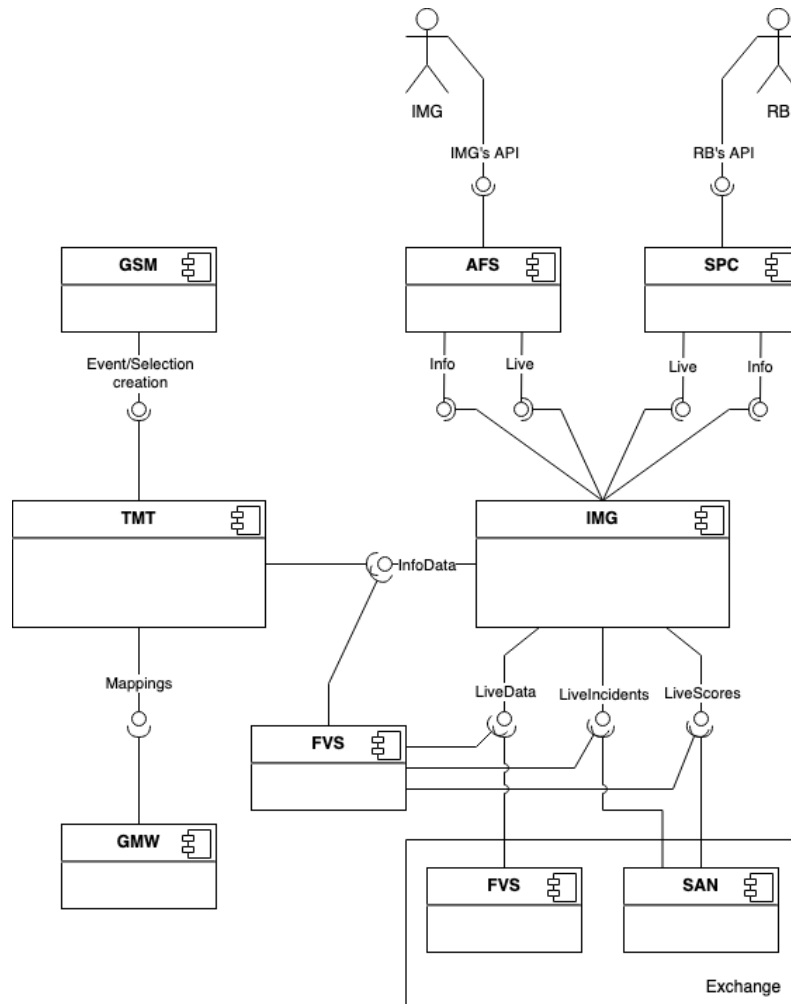


Figure 5.1: IMG and TMT - Container Diagram.

Additionally, it was considered relevant to include a deployment diagram in order to enhance the comprehension of the overall system architecture and the specific deployment locations of the APIs within the production environment's infrastructure. This said the designed deployment diagram can be viewed in Figure 5.2.

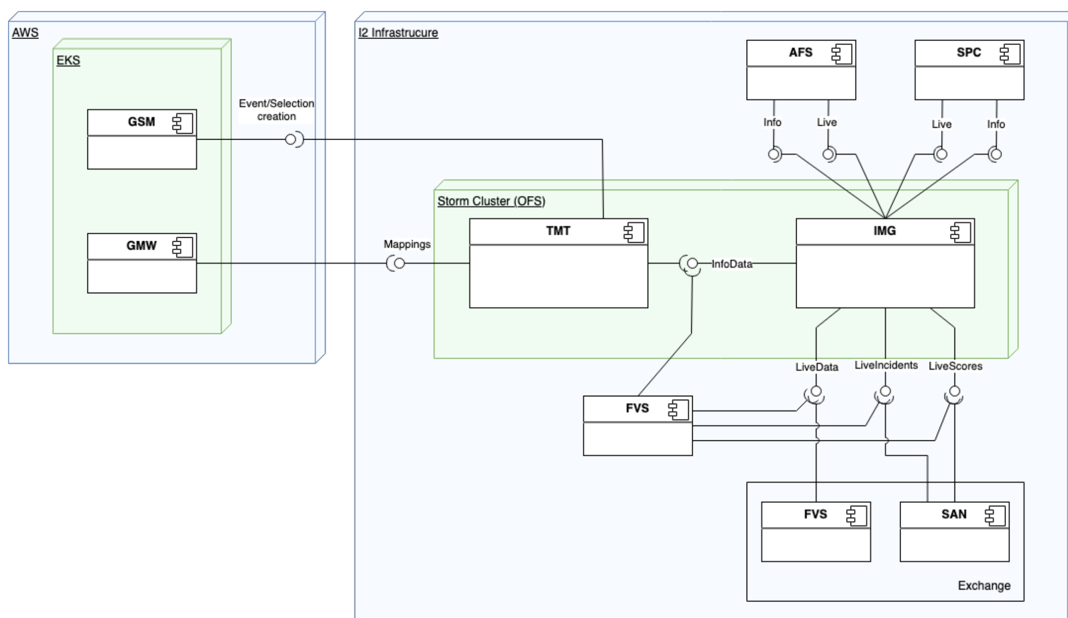


Figure 5.2: IMG and TMT - Deployment Diagram.

It is evident from the deployment diagram that the majority of APIs, which directly interact with IMG, are deployed within Flutter Entertainment's internal infrastructure (i2) on Linux Virtual Machines. However, both TMT and IMG topology APIs are situated within an Apache Storm Cluster (OFS). On the other hand, the APIs associated with the Global Betting Platform flow, GSM and GMW, are deployed within an Elastic Kubernetes Service (EKS) cluster in the AWS ecosystem.

Furthermore, since all APIs communicate through Kafka topics, there are two Kafka clusters in place to facilitate the required communication. One cluster is located within the i2 infrastructure, enabling communication between APIs deployed in i2. The other cluster resides in AWS, serving the APIs deployed within that environment.

5.1.2 Implementation details

Before beginning the actual implementation phase, an extensive examination of the AsyncAPI schema specification was indispensable. This task was carried out as part of the US #1, enabling an extensive comprehension of AsyncAPI, its functionality, and its applicability.

The AsyncAPI specification plays a crucial role in providing a standardized format for documenting and describing message-driven APIs. It serves as a comprehensive means of defining the structure, format, and behaviour of messages exchanged between various components within an application or system.

To achieve this, the AsyncAPI specification utilizes a file or a set of connected files, which are necessary for describing APIs. These files can be in either JSON or YAML format and follow a specific structure defining a schema. The schema consists of multiple components, but usually, there are nine root components, each with distinct responsibilities, including:

- **asyncapi** - *Required* - Specifies the AsyncAPI Specification version being used. It can be used by tooling Specifications and clients to interpret the version.
- **id** - Identifier of the application the AsyncAPI document is defining.

- **defaultContentType** - Default content type to use when encoding/decoding a message's payload.
- **info** - *Required* - Provides metadata about the API. The metadata can be used by the clients if needed.
- **tags** - A list of tags used by the specification with additional metadata. Each tag name in the list must be unique.
- **externalDocs** - Additional external documentation.
- **components** - An element to hold various schema objects to be reused in the specification.
- **servers** - Provides connection details of servers.
- **channels** - *Required* - The available channels and messages for the API.

Additionally, the AsyncAPI Specification allows for the addition of custom data through extension properties, which are denoted by fields starting with `x-`. These extensions provide flexibility to include additional information that may not be covered by the core specification. The value of an extension can be null, a primitive type, an array, or an object in valid JSON format.

In this case study, all the nine root components of the AsyncAPI specification were defined, and below, the implementation details of each component will be explored. This exploration will provide a comprehensive understanding of how each component contributes to the overall structure and behaviour of the message-driven APIs under investigation and it was conducted under the USs #3 and #4.

1. **asyncapi**

   The version string denotes the AsyncAPI Specification version that the document complies with. It follows the format `major.minor.patch`, and the current stable version is `2.6.0`.

   While version `3.0.0` is in development, it is worth noting that many external plugins and tools for AsyncAPI do not fully support this version yet. Therefore, to ensure compatibility and stability, the decision was made to use the latest stable version as demonstrated in Figure 5.3

```
1   # ---
2
3   asyncapi: 2.6.0
4
5   # ---
```

Figure 5.3: AsyncAPI version.

2. **id**

   This field in the AsyncAPI document serves as a unique universal identifier for the defined application. AsyncAPI recommends using Uniform Resource Name (URN)[2] to ensure global identification.

[2] https://datatracker.ietf.org/doc/html/rfc8141

In Figure 5.4, an example identifier of a created AsyncAPI schema spec file is presented. Both APIs are Maven[3] projects, so the identifier follows the following format:

`groupId:artifactId:version:currentLifecyclePhase`.

This identifier provides comprehensive information, including the artefact details and the current lifecycle stage of the API. Consequently, by referring to the identifier alone, it is possible to retrieve the essential details about the API.

```
23    # ---
24
25    id: 'com:ppb:feeds:mapping:ppb-tennis-mapping-topology:1.47.0:published'
26
27    # ---
```

Figure 5.4: AsyncAPI identifier.

Regarding the API lifecycle stages, after considering the API lifecycle models stated in Section 2.1.1, the defined stages were:

Table 5.2: API lifecycle stages

Stage	Description
Prototype	An API prototype allows early testing by providing a preview of a new or updated API. Clients can try out the API, gaining early access to its features.
Created	The API has been developed but is not yet accessible to the public.
Published	The API has been developed, tested, deployed and is now ready for users.
Blocked	The API is blocked for some time and cannot be used, although it can be published again.
Deprecated	When a new API version is published, the old version should be deprecated.
Retired	The API is retired and no longer is or will be in use.

3. **defaultContentType**

The defaultContentType field specifies the default media type to be used when encoding or decoding a message's payload. It must be a specific media type, such as `application/xml`. It is used by schema parsers when the contentType property of the message's payloads is not provided.

An example of a possible defaultContentType is demonstrated in Figure 5.5.

[3] https://maven.apache.org/

```
10
11  # ---
12
13  defaultContentType: application/json
14
15  # ---
16
```

Figure 5.5: AsyncAPI default content type.

4. **info**

   This section contains metadata about the API, including the mandatory fields title and version. In addition to these two fields, the info section can include other optional fields.

   Bellow, are all the available fields:

   - **title** - *Required* - The name of the application.
   - **version** - *Required* - Provides the version of the application.
   - **description** - A short description of the application responsibilities.
   - **termsOfService** - An Uniform Resource Locator (URL) to the Terms of Service for the application.
   - **contact** - The contact information for the exposed API.
   - **license** - The license information for the exposed API.

   An example of an info section created during the development of the proof of concept development can be visualized in Figure 5.6

```
15  # ---
16
17  info:
18    title: TMT - Tennis Mapping Topology
19    version: '1.47.0'
20    contact:
21      name: MOB team
22      email: mobdev@betfair.com
23      url: https://confluence.app.betfair/display/blip/Mob+Team
24    description: |
25      Tennis Mapping Topology (TMT) is responsible to consume messages
26      from IMG/RB and RAMP and produce the mapping between the
27      Exchange Feeds Sportex event and RAMP id to set in Openbet.
28
29      As part of the GBP project, tennis messages coming from Ramp
30      will arrive at TMT from GSM. So, TMT receives messages from
31      GSM, processes them and emits to the global mappings output
32      topic, so they are replicated to GMW.
33
34      There are two main flows within TMT:
35        - regular events flow
36        - outright flow: flow to specially publish outright mappings
37          for special scenarios, namely when IMG reuses the same
38          competition id over the years and emits a different
39          creation message.
40
41  # ---
```

Figure 5.6: AsyncAPI information.

5. **tags**

   The tags section allows for the inclusion of metadata as a form of individual tags. Each tag consists of a mandatory name field, which serves as an identifier for the tag. Additionally, the tag can include an optional description, providing a brief description of the tag and the field externalDocs, to provide additional external documentation, including a description and URL.

   In Figure 5.7 there is an example of the tags created for the TMT API where it is shown the betting style, the sport flow and the project that TMT bellows. Additionally, the API lifecycle stage is displayed again.

```
57   # ---
58
59   tags:
60     - name: exchange
61       description: Exchange Betting
62     - name: tennis
63       description: Tennis Flow
64     - name: global
65       description: Part of the Global Betting Platform
66     - name: published
67       description: API Livecycle Stage
68
69   # ---
```

Figure 5.7: AsyncAPI tags.

6. **externalDocs**

   Allows referencing an external resource for extended documentation regarding the API. This includes a short description as a form of resume of the documentation and the mandatory URL to external documentation, as it can be seen in Figure 5.8.

```
47   # ---
48
49   externalDocs:
50     description: Further detail about TMT topology can be found on TMT Hub.
51     url: https://confluence.app.betfair/display/FP/TMT+-+Tennis+Mapping+Topology
52
53   # ---
54
```

Figure 5.8: AsyncAPI external documentation.

7. **components**

   The components section serves as a repository for reusable objects that cover various aspects of the AsyncAPI specification. These objects, do not impact the AsyncAPI specification unless they are specifically referenced from properties outside the components section.

   Its purpose is to provide a centralized location for defining and managing reusable elements in the AsyncAPI specification.

8. **servers**

   The servers section in the AsyncAPI specification represents an array of objects that define message brokers, servers, or another mechanism capable of sending and/or

receiving data. These servers contain important details such as URI, protocols, and security configurations. The URL field supports variable substitution, allowing for the dynamic replacement of specific details such as credentials or datacenters by generation tools.

In the context of this case study, the servers correspond to Kafka clusters, and there are two entries defined: one for the development environment (`mfs-dev`) and another for production (`mfs-prd`), as pictured in Figure 5.9 and 5.10.

```
59  servers:
60    mfs-dev:
61      url: '{dc}-mfs{vmNum}-{env}.{env}.betfair:{port}'
62      protocol: kafka
63      protocolVersion: 2.0.1
64      description: '[ QA ] Kafka cluster from where TMT topology get the input data.'
65      tags:
66        - name: 'env:qa'
67          description: 'This environment is meant for running internal component
68                  tests during the development.'
69        - name: 'kind:remote'
70          description: 'This server is a remote server. Not exposed by the application.'
71      variables:
72        dc:
73          enum:
74            - 'ie1'
75            - 'ie2'
76          default: 'ie1'
77          description: 'The datacenter that the cluster belongs to.'
78        vmNum:
79          enum:
80            - '001'
81            - '002'
82            - '003'
83          default: '001'
84          description: 'The virtual machine identification. In this case, the
85                  cluster has 3 machines.'
86        env:
87          enum:
88            - 'qa'
89            - 'nxt'
90          default: 'qa'
91          description: 'The development environment of the cluster.'
92        port:
93          enum:
94            - '9092'
95          default: '9092'
96          description: 'The port that must be used to connect to the cluster.'
```

Figure 5.9: AsyncAPI servers - dev.

Figure 5.9 represent the details of the Kafka cluster of the development environment. Similarly, there is a definition of the production cluster and it can be seen in Figure 5.10.

```
 59  servers:
 97    mfs-prd:
 98      url: '{dc}-mfs{vmNum}-prd.prd.betfair:{port}'
 99      protocol: kafka
100      protocolVersion: 2.0.1
101      description: '[ PRD ] Kafka cluster from where TMT topology get the
102                      input data.'
103      tags:
104        - name: 'env:prd'
105          description: 'This environment is the live environment available for
106                          final users.'
107        - name: 'kind:remote'
108          description: 'This server is a remote server. Not exposed by the
109                          application.'
110      variables:
111        dc:
112          enum:
113            - 'ie1'
114            - 'ie2'
115          default: 'ie1'
116          description: 'The datacenter that the cluster belongs to.'
117        vmNum:
118          enum:
119            - '001'
120            - '002'
121            - '003'
122          default: '001'
123          description: 'The virtual machine identification. In this case, the
124                          cluster has 3 machines.'
125        port:
126          enum:
127            - '9092'
128          default: '9092'
129          description: 'The port that must be used to connect to the cluster'
```

Figure 5.10: AsyncAPI servers - prd.

Detailed explanations of each field within the servers section can be found below:

- **url** - *Required* - The target host's URL. It supports variable substitutions and can be relative to the document's serving location.
- **protocol** - *Required* - The protocol this URL supports for connection.
- **protocolVersion** - The version of the protocol used for the connection.
- **description** - A short explanation of the message broker.
- **variables** - It maps variable names to their corresponding values. These values are used for substitution in the server's URL template.
- **security** - The security mechanisms that can be used with the server. It contains a list of alternative security requirement objects, and satisfying any one of these security requirements is sufficient to authorize a connection or operation.
- **tags** - A logical grouping and categorization of servers, helping to organize and categorize servers based on specific criteria or characteristics.
- **bindings** - Specifies different protocols used by the server, with corresponding definitions or configurations.

9. **channels**

This section represents the relative paths to individual channels and their operations available on the servers. The channels, depending on the message broker in use, can also be known as topics, routing keys, event types, or paths.

In this case study they represent the available Kafka topics and the content messages associated with them.

The channels section can include multiple fields, but the most important ones are subscribe/publish as they define the content for subscribing to and publishing messages on the channels. The full list of available properties are:

- **description** - A short description of the channel.
- **servers** - Specifies the availability of a channel on specific servers. If not specified, the channel is available on all servers defined in the servers section.
- **subscribe/publish** - Represent an operation, such as subscribe or publish, that defines the messages produced by the application and sent to the channel. These operations can have additional fields within them, which will be further detailed below.
- **parameters** - A map that contains the parameters included in the channel name. It is typically used when working with channels that involve expressions.
- **bindings** - Specifies different protocols used by the channel, with corresponding definitions or configurations.

As for the subscribe and publish operations, they allow documenting the process and rationale behind sending and receiving messages. For instance, consider the example of Figure 5.11, the publish operation describes a message received by the TMT API, while the subscribe operation describes the messages sent by the API.

```
131  # ---
132
133  channels:
134    feeds.topic.events.tennis:
135      description: 'Input topic with catalogue of Events coming from
136                      IMG topology (RB and IMG providers).'
137      publish:
138        summary: Input topic containing the Events that happened or will
139                  happen in a short future an all their information.
140        message:
141          $ref: '#/components/messages/feeds.events.tennis'
142    global.exchange.input.events:
143      description: 'Input topic of Ramp Events coming from
144                      Global Sportex Mapping flow.'
145      publish:
146        summary: Input topic from where Global Sportex Mapping flow
147                  requests Ramp Events to be mapped.
148        message:
149          $ref: '#/components/messages/global.input.events'
150    global.exchange.output.instructions:
151      description: 'Output topic of Event Mappings going to
152                      Global Sportex Mapping flow.'
153      subscribe:
154        summary: Output topic where Event Mappings are sent to be
155                  processed by Global Sportex Mapping flow.
156        message:
157          $ref: '#/components/messages/global.output.mappings'
158
159  # ---
```

Figure 5.11: AsyncAPI channels.

The available fields of these operations are:

- **operationId** - A unique string used to identify the operation within the API. It must be unique among all the operations described in the API.
- **summary** - A summary of the operation's responsibilities.
- **description** - A verbose explanation of the operation.
- **security** - Declares operation-specific security requirements. Only one security requirement needs to be satisfied to authorize the operation, including server security if applicable.
- **tags** - List of tags for logical grouping and categorization of operations.
- **externalDocs** - Additional external documentation.
- **bindings** - Protocol-specific definitions for the operation are provided in a map, where the keys represent the protocol names. In this case, it represents Kafka's topic details.
- **traits** - A list of traits to apply to the operation object. In this example, it represents the Kafka's headers.
- **message** - A map that allows the usage of the oneOf key to specify multiple messages. However, each message must be valid against only one of the message objects.

Furthermore, to enhance the organization of the AsyncAPI schema spec file, the operation's definition can be extracted using the ref field. This allows for better structuring and reutilization of operation definitions.

In the example of Figure 5.11, it is possible to see that the message definition is being referenced. The definition is then specified in the components section, as demonstrated in Figure 5.12, where the relevant information is shown, like the payload's contentType, the payload's schema, the Kafka headers provided with the message's payload, the Kafka's topic consuming details and others.

```
145  components:
146    messages:
147      global.input.events:
148        name: global.input.events
149        title: Global Input Events
150        summary: Input payload of a GSM event mapping request.
151        contentType: application/json
152        payload:
153          $ref: '#/components/schemas/global.input.events.payload'
154        bindings:
155          $ref: '#/components/operationBindings/global.input.events'
156        traits: [
157          { $ref: '#/components/messageTraits/requiredHeaders' },
158          { $ref: '#/components/messageTraits/globalHeaders' }
159        ]
```

Figure 5.12: AsyncAPI compoenents messages definition.

The message object also references other objects, such as the bindings and traits. To facilitate reusability across different message payloads, these definitions are included in the components section as separate objects, allowing them to be easily referenced and reused in the specification. These referenced objects, along with their respective definitions and details, are represented in Figure 5.13.

```
161  components:
162    messages: ...
188    # ---
189    messageTraits:
190      requiredHeaders:
191        headers:
192          type: object
193          properties:
194            ID:
195              type: string
196              description: The provider id for the given payload.
197            CORRELATION_ID:
198              type: string
199              description: The correlation id of all messages
200                          related to input date.
201      globalHeaders:
202        headers:
203          type: object
204          properties:
205            RAMP_SPORT:
206              type: integer
207              description: 'The Ramp Sport id. As this is the Tennis
208                          flow, it should be "13"'
209            SEQUENCE:
210              type: string
211              description: 'Sequence number that indicated the order
212                          of the message.'
213    # ---
214    operationBindings:
215      global.input.events:
216        kafka:
217          key:
218            type: string
219            description: The kafka topic key is the event id
220          consumerGroupId:
221            type: string
222            enum: [ 'feeds.tmt.consumer.gsm' ]
223    # ---
```

Figure 5.13: AsyncAPI components bindings and traits definition.

In addition to the bindings and traits referenced example, the message object in the AsyncAPI specification also mentions the schemas field, which describes the structure and format of the message's payload. The schema definition for the TMT API can be found in Figure 5.14.

```
161  components:
162    schemas:
163      feeds.events.tennis.payload:
164        type: object
165        properties:
166          $ref: 'https://<TOKEN>@raw.githubusercontent.com/<ORGANIZATION>/flutter-schema-registry
167                /main/sbk-catalogue/tmt/input/events_tennis.yml#/events.tennis.payload'
168      global.input.events.payload:
169        type: object
170        properties:
171          $ref: 'https://<TOKEN>@raw.githubusercontent.com/<ORGANIZATION>/flutter-schema-registry
172                /main/sbk-catalogue/tmt/input/global_event.yml#/global.event.payload'
173        examples:
174          - $ref: 'https://<TOKEN>@raw.githubusercontent.com/<ORGANIZATION>/flutter-schema-registry
175                  /main/sbk-catalogue/tmt/input/example_feeds_topic_events_tennis.yml#/example'
176      global.output.event.mapping.payload:
177        type: object
178        properties:
179          $ref: 'https://<TOKEN>@raw.githubusercontent.com/<ORGANIZATION>/flutter-schema-registry
180                /main/sbk-catalogue/tmt/output/global_event_mapping.yml#/global.event.mapping.payload'
181    # ---
```

Figure 5.14: AsyncAPI components schemas definition.

In the schema definition shown in Figure 5.14, the payload structure is not explicitly defined within the AsyncAPI specification. Instead, it is referenced using a GitHub

URL. This URL points to a private repository within the organization's GitHub repositories called flutter-schema-registry that contains all the API's payload definitions, acting as the schema registry solution. This has been done to fulfil the requirements of US #2 and the output result of the structure of the repository is shown in Figure 5.15.

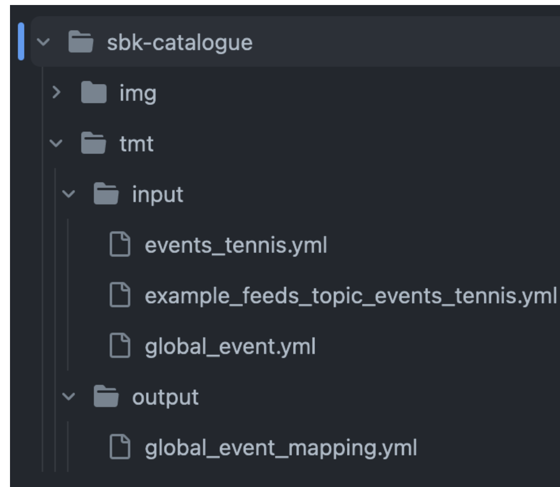


Figure 5.15: GitHub schema registry repository - flutter-schema-registry.

It follows a folder-based distribution where the root folders are named after the respective departments, such as Sportsbook Catalogue in this case. Within each root folder, there is a folder for each API that contains two additional folders: input and output. These folders represent, respectively, the input payloads that the API consumes and the output payloads that the API produces. Furthermore, all the files within these folders are expected to be in YAML format.

As it also is possible to state from Figure 5.14, to access the private repository, a token was generated to provide access to all users within the organization on GitHub. Additionally, in order for the content of the file to be accessible to the AsyncAPI specification, the domain `raw.githubusercontent.com` was used. This domain allows querying the raw content of unprotected GitHub files, making it possible for the AsyncAPI specification to retrieve and utilize the payload definitions from the private repository.

Regarding the structure of the YAML files within the flutter-schema-registry repository, Figure 5.16 demonstrates an example of the payload produced by the TMT API.

```
1   global.event.mapping.payload:
2     eventId:
3       type: string
4     event:
5       type: object
6       properties:
7         id:
8           type: object
9           properties:
10            rampId:
11              type: string
12            sportexId:
13              type: string
14        sportexTypeId:
15          type: string
16        sportexSportId:
17          type: string
```

Figure 5.16: AsyncAPI TMT's output payload.

> The payload contains the data defined by the application, which needs to be serialized into a specific format such as JSON, XML, Avro, binary, among others. this can be explicitly defined with the contentType field.

In order to facilitate the validation and generation of the documentation, a makefile was created. This work was address as part of the US #5 and Figure 5.17 represents the makefile created for the TMT API.

```
1   validate:
2       asyncapi validate tennis_mapping_topology_asyncapi.yml
3
4   generate_html:
5       asyncapi generate fromTemplate tennis_mapping_topology_asyncapi.yml @asyncapi/html-template
6           -o generated-doc/html/
7           --force-write
8
9   generate_md:
10      asyncapi generate fromTemplate tennis_mapping_topology_asyncapi.yml @asyncapi/markdown-template
11          -o generated-doc/md/
12          -p outFilename=tennis_mapping_topology_asyncapi.md
13          --force-write
14
15  delete_all_generated_file:
16      rm -r generated-doc/*
```

Figure 5.17: AsyncAPI makefile.

Having the makefile, the validation and generation of documentation can be done using simple commands making it easier to use. The result of the generated documentation for the TMT API, in HyperText Markup Language (HTML) format, can be found in Appendix C.

5.2 Testing

During this phase, the aim was to validate the solution's variability and enable autonomous test generation. To accomplish this, the usage of Microcks was put into place as suggested by AsyncAPI. After a study of this tool as required by US #7, a detailed understanding of the tool and the required knowledge for its usage was obtained.

Microcks is an open-source tool designed for API and microservices mocking and testing. Its primary objective is to provide a platform for referencing, deploying mocks, and facilitating contract testing of services and APIs. It can also be regarded as a service virtualization

solution, as it allows for the implementation of fake APIs or services before the actual development takes place.

To begin the testing process as stated in US #8, the first step involved cloning the Microcks repository to access and utilize the tool. The repository contains a Docker Compose file that facilitates the setup of all necessary components for AsyncAPI within containers. This configuration allows for the execution of Microcks' user interface, as pictured in Figure 5.18.

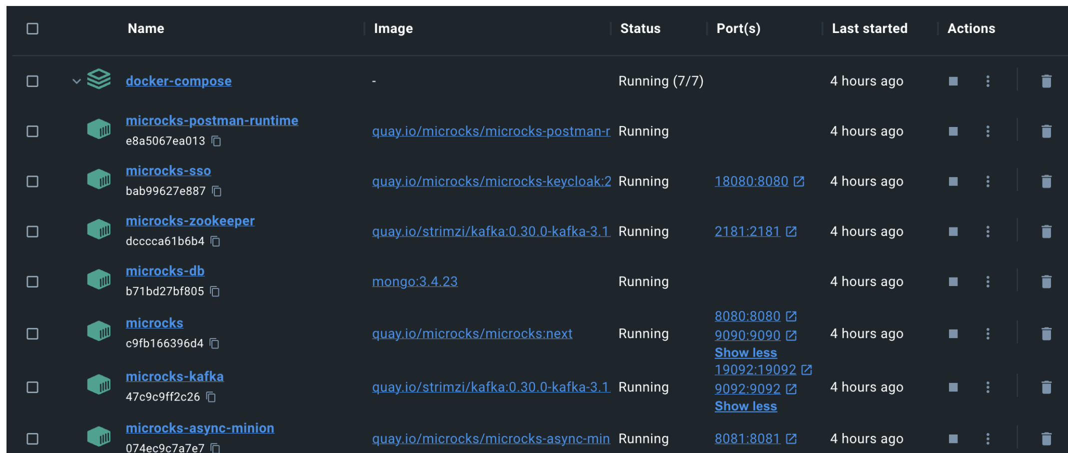


Figure 5.18: Microcks docker containers for AsyncAPI.

This user interface provides the capability to import AsyncAPI schema specification files, enabling Microcks to gain a comprehensive understanding of the APIs by extracting all the relevant details. This specification is then utilized by Microcks to generate tests or mock the APIs accordingly. Furthermore, Microcks can serve as a centralized hub for managing the APIs, as depicted in the figure. 5.19.

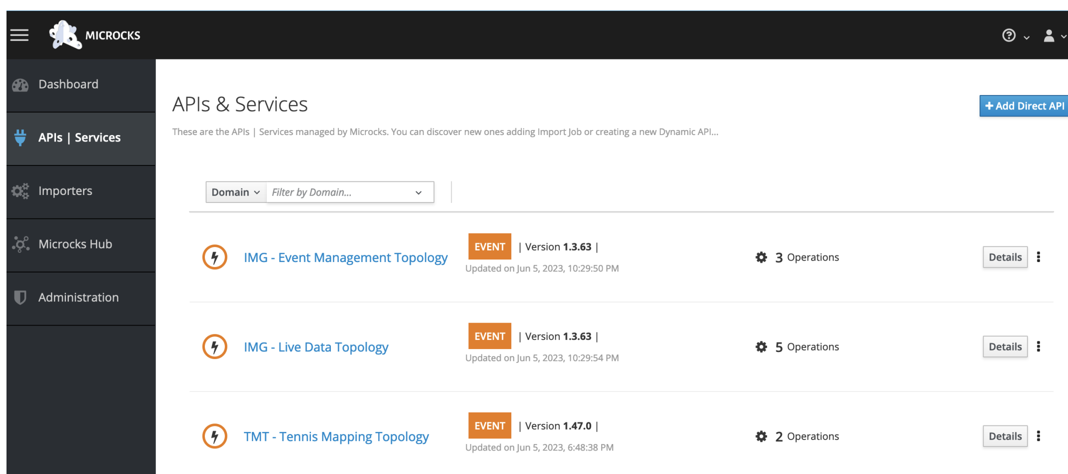


Figure 5.19: Microcks APIs.

For every entry, it is possible to examine the operations (equivalent to the channels part of the AsyncAPI schema specification so, in this case, it represents the Kafka topics) associated with each API and the specific details of the mocked messages. This can be observed in Figure 5.20, where the payload used for testing is displayed. It is worth noting that certain

functions are used to generate values such as timestamps, strings, or integers, ensuring the test cases are personalized and cover the expected scenarios.

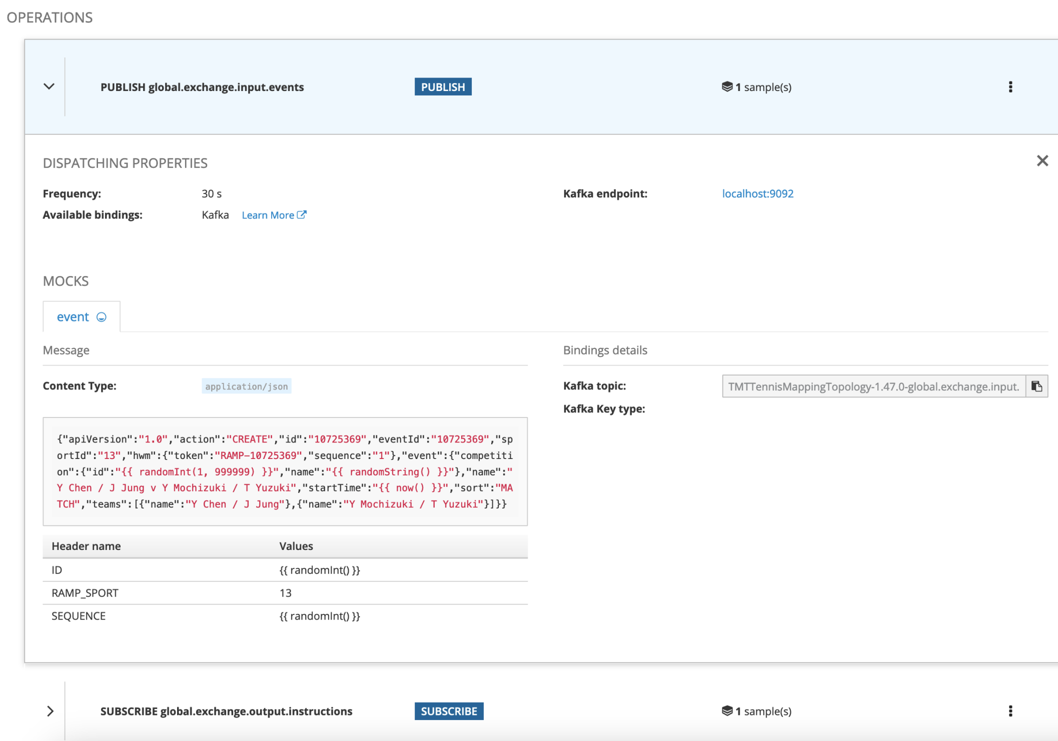


Figure 5.20: Microcks operations details.

To enable the usage of functions for generating diverse payloads within the messages, modifications were made to the AsyncAPI schema specification files. Explicit examples of payloads were added to the schema, incorporating the necessary functions within the relevant fields. This can be observed in Figure 5.21, where the modified AsyncAPI schema specification demonstrates the inclusion of the `randomInt()` function for payload generation.

```
1  components:
2    messages:
3      global.output.mappings:
4        name: global.output.mappings
5        title: Global output Mappings
6        summary: Output payload of a Mapping going to GSM flow.
7        contentType: application/json
8        payload:
9          $ref: '#/components/schemas/global.output.event.mapping.payload'
10       traits: [
11         { $ref: '#/components/messageTraits/requiredHeaders' },
12         { $ref: '#/components/messageTraits/globalHeaders' },
13         { $ref: '#/components/messageTraits/globalOutputHeaders' }
14       ]
15       examples:
16         - name: event
17           summary: Example of a global event mapping
18           headers: {
19             "RAMP_SPORT": 13,
20             "SEQUENCE": '{{ randomInt() }}',
21             "ID": '{{ randomInt() }}'
22           }
23           payload: {
24             "eventId": "{{ randomInt() }}",
25             "event": {
26               "id": {
27                 "rampId": "{{ randomInt() }}",
28                 "sportexId": "{{ randomInt() }}"
29               },
30               "sportexTypeId": "28242095",
31               "sportexSportId": "2"
32             }
33           }
34 # ---
```

Figure 5.21: Microcks AsyncAPI changes.

Moreover, Microcks provides the option to contract test APIs by checking if the actual output result of the API is conformant to its contract. This process requires the input of some configurations, including the operation to be tested and the server endpoint to be utilized during the testing process, among other parameters, as illustrated in Figure 5.22.

Services & APIs › TMT - Tennis Mapping Topology - 1.47.0 › New Test

New Test

Service under test

TMT - Tennis Mapping Topc | v. | 1.47.0

The service and its version we're going to test.

*** Test Endpoint**

localhost:9092/global.exchange.output.events

A valid endpoint to use for testing (Kafka/MQTT/NATS/AMQP broker or WebSocket server + topic name for Asynchronous API).

*** Operation**

SUBSCRIBE global.exchange.output.instructions

Select the Asynchronous operation you want to test (only SUBSCRIBE is supported at the moment).

*** Runner**

ASYNC API SCHEMA

The runner to use for this test (for now we only have ASYNC API SCHEMA compliance). Learn More

Show advanced options

Cancel | + Launch test !

Figure 5.22: Microcks test generation.

To achieve this, Microcks examines the messages on the provided Kafka topic and compares their payload against the AsyncAPI schema specification contract. It then assigns a conformance score ranging from 0% to 100%, indicating the level of cohesion of the APIS to the contract. Figure 5.23 illustrates the conformance score for the TMT API and its output contract.

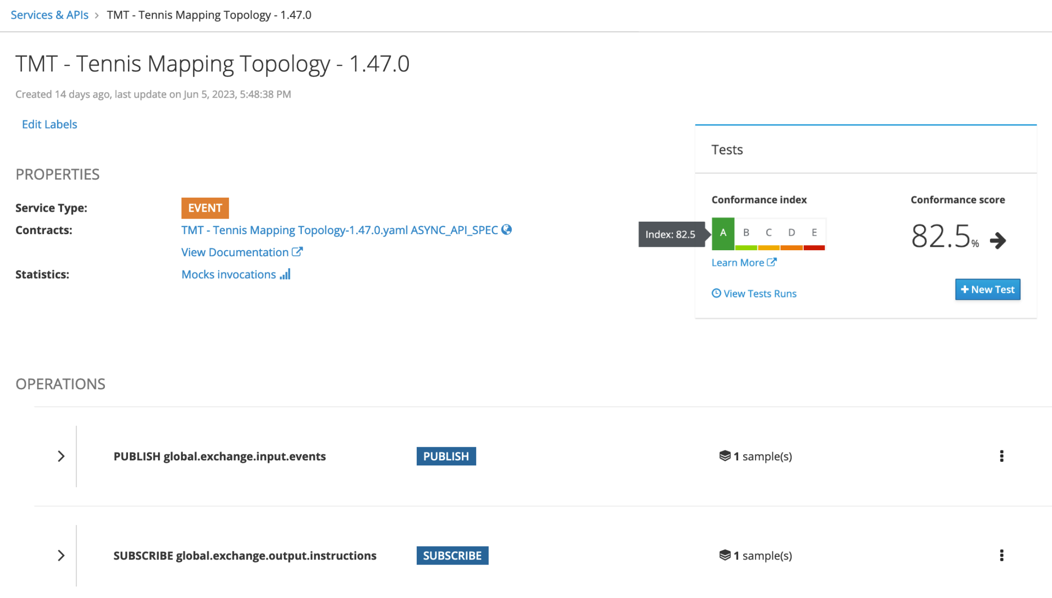


Figure 5.23: Microcks test conformance score.

Based on the score in Figure 5.23, it is apparent that there might be some discrepancies between the actual output of the TMT API and the defined contract. However, it is important to note that these differences are not significant enough to be considered breaking, as the score remains within the acceptable range (index A).

Microcks also provides detailed error messages that explain the reasons for test failures. These error messages can be as specific as fields with incorrect types, unsupported values, and missing required fields, among others. It also provides tests' execution history, as shown in Figure 5.24 so that it is possible to track if the APIs maintain their contracts or, on the other hand, if they are losing their loyalty to the established contract.

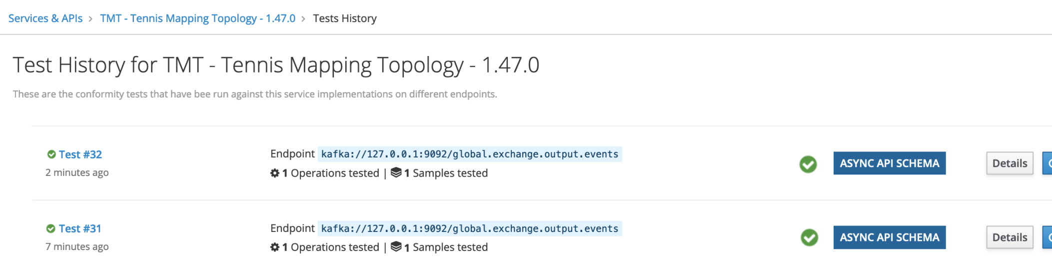


Figure 5.24: Microcks test history.

5.3 Delivery

After completing the implementation and conducting the API testing phase, the focus shifted towards the delivery of the solution. This phase involved two crucial steps.

Firstly, it encompassed the generation of API documentation within the API pipelines, ensuring that the documentation remains up-to-date and aligned with the implemented APIs.

Secondly, it involved the delivery and transfer of knowledge regarding the AsyncAPI specification and the schemas' files derived from it, ensuring that the relevant information was effectively communicated to the stakeholders involved.

5.3.1 Integration in pipelines

Both APIs chosen for the proof of concept already have a well-organized and thoughtful CI/CD pipeline in place. With this, it was decided to integrate the validation and generation of API documentation seamlessly into the existing CI/CD pipeline and a dedicated workflow was created using GitHub Actions.

This workflow, as depicted in Figure 5.25, focuses specifically on the AsyncAPI directory where the AsyncAPI's Schema Spec files are located. By doing so, unnecessary resource consumption and processing time are avoided when there are no changes to the AsyncAPI files.

```yaml
name: AsyncAPI documentation generation

on:
  push:
    paths:
      - asyncapi/**
    branches: [main]
permissions:
  contents: write

jobs:
  asyncapi-generate-doc:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repo
        uses: actions/checkout@v3

      - name: Install AsyncAPI CLI
        run: npm install -g @asyncapi/cli

      - name: Validating AsyncAPI document
        run: make validate -C asyncapi/

      - name: Generating HTML from my AsyncAPI document
        run: make generate_html -C asyncapi/

      - name: Deploy HTML docs to GitHub Pages
        uses: JamesIves/github-pages-deploy-action@v4.4.1
        with:
          branch: main
          target-folder: asyncapi/generated-doc/html/
          folder: asyncapi/generated-doc/html

      - name: Generating Markdown from my AsyncAPI document
        run: make generate_md -C asyncapi/

      - name: Deploy Markdown docs to GitHub Pages
        uses: JamesIves/github-pages-deploy-action@v4.4.1
        with:
          branch: main
          target-folder: asyncapi/generated-doc/md/
          folder: asyncapi/generated-doc/md
```

Figure 5.25: GitHub workflow - AsyncAPI documentation generation.

The workflow begins by checking out the repository and installing the required dependencies. It leverages the commands available in the makefile to validate the AsyncAPI schema spec files and generate both HTML and Markdown documentation. With write access granted, the workflow publishes the generated documentation back to the repository using the github-pages-deploy-action[4], available on the GitHub Marketplace, as it can be seen in Figure 5.26.

[4] https://github.com/marketplace/actions/deploy-to-github-pages

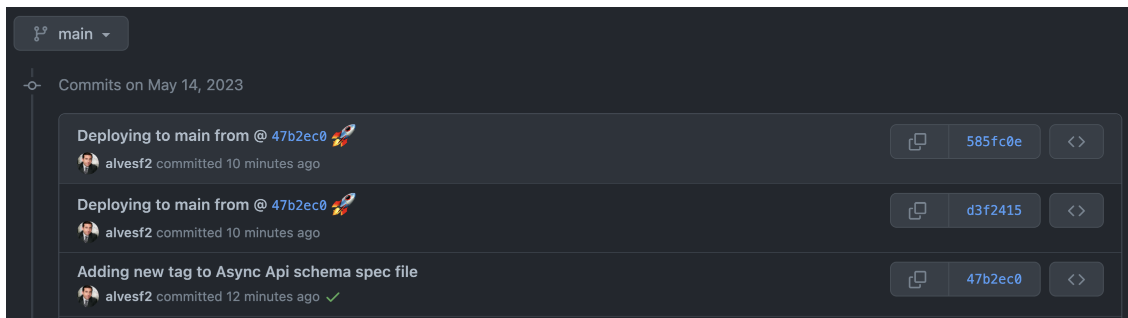


Figure 5.26: GitHub workflow - Generated documentation commit example.

This documentation is available in both APIs' repositories on the directory *asyncapi/generated-doc/html* or *asyncapi/generated-doc/md* so that it can be accessed within the APIs' GitHub repository through the browser. An example of how the documentation can be visualized o GitHub is demonstrated in Figure 5.27.

Figure 5.27: GitHub workflow - Generated documentation location.

This efficient process ensures that the documentation is automatically validated and updated whenever changes are made to the AsyncAPI files. By incorporating it into the CI pipeline, the documentation remains consistent and easily accessible alongside the APIs, contributing to a more efficient and smooth development workflow.

5.3.2 AsyncAPI how to guide

According to the requirements of US #6, and to facilitate the adoption of AsyncAPI among developers comprehensive documentation was created within the respective API repositories.

The documentation introduces the concept of AsyncAPI, highlights its features, and provides guidance on the structure of an AsyncAPI schema spec file. Furthermore, new patterns were introduced to standardize the implementation approach, and specific commands were recommended for developers to use.

The documentation begins by providing an overview of AsyncAPI, explaining its purpose and functionality. It then delves into the topic of versioning, emphasizing the use of Semantic Versioning as the new standard for version management. Additionally, the documentation

covers the lifecycle management of AsyncAPI documents, outlining best practices for handling updates and modifications. Lastly, the recommended schema registry for managing and referencing API payloads is discussed.

To ensure easy accessibility and centralized storage of this valuable information, a dedicated GitHub repository was created within the organization. The repository is structured according to different areas and APIs, housing their payloads. This centralized repository serves as a single point of reference for developers, allowing them to quickly access the necessary information. Moreover, these payloads are meticulously referenced within the AsyncAPI schema spec files, promoting consistency and ease of use.

This guide can be visualized in Appendix D.

Chapter 6

Experimentation and Evaluation

The objective of this chapter is to provide a comprehensive overview of the problem addressed in this study and outline the approach taken to evaluate it.

The research focuses on investigating a viable solution for addressing system interoperability challenges resulting from API evolution. To accomplish this, the work was divided into several sections, including an examination of relevant concepts, an identification of potential solutions, and a comparison and evaluation of those solutions.

The evaluation criteria were determined based on the investigation topics, which encompassed areas such as API lifecycle management, API versioning, API documentation, API schema registry, API testing, and the overall usability of the developed proof of concept.

To gather insights and assess the effectiveness of the proof of concept, a questionnaire was distributed to key stakeholders. The results were positive although there still is room for improvement.

6.1 Problem Description

The work developed during this study was carried out at Flutter Entertainment, a global sports betting, gaming and entertainment provider. With the latest acquisitions of multiple international brands to the Flutter group, it started to be challenging to manage the system's APIs as a whole due to its complexity and the interoperability eventually started to decrease.

With this, it is difficult to know the existing APIs and their contracts which could cause compatibility issues between APIs resulting in increased development time. On the other hand, the constant changes in contracts make it difficult to keep the documentation up-to-date as it is manually managed and so it can result in outdated documentation.

All these issues are very time-consuming and require alignment between multiple teams to implement a change in an API. These increase costs as more people need to be involved in projects. Lastly, these issues could ultimately result in incidents being raised that could impact the betting experience of clients and it also results in a more challenging task to manage the system as a whole.

6.2 Objectives

The purpose of this research study is to identify a method for preserving system interoperability during the evolution of APIs.

To achieve this goal, it was deemed necessary to conduct comprehensive research on the related theoretical concepts, including the notion of Web APIs, system interoperability, and API lifecycle management.

This study also involved a study of different types of Web APIs and their respective implications for system interoperability. Moreover, the study aimed to explore potential solutions for documenting APIs, managing API schema registries, performing API testing, and ensuring the interoperability of system APIs.

Finally, the ultimate objective of the research was to implement the most suitable solution and assess its effectiveness.

6.3 Hypotheses

The evaluation criteria serve the purpose of facilitating the assessment of the project and validating whether the produced work successfully addressed the problem of evolving the system while maintaining APIs' interoperability.

These criteria provide a framework for evaluating the effectiveness of the implemented solution in achieving this objective. Based on the problem and objectives, the following hypotheses were established:

- **Null hypothesis (H0)** - The adoption of the selected solution did not resolve any of the interoperability problems.
- **Alternative hypothesis (H1)** - The adoption of the selected solution had benefits towards the interoperability problem.

To validate these hypotheses, feedback surveys will be performed to the valuable stakeholders as well as applied both the Wilcoxon and Shapiro-Wilk methods to validate the hypotheses.

6.4 Identification of indicators and sources of information

In this section, the evaluation indicators are listed, and the sources of information to be utilized within the context of this work are identified.

6.4.1 Indicators

The evaluation indicators serve the primary purpose of analyzing the identified hypotheses and assessing the achievement of the associated objectives based on a set of predetermined criteria. With this in mind, the following indicators have been identified:

- **API lifecycle management** - If the issue regarding the API lifecycle management was fully fulfilled.
- **API versioning** - If the problems of current inconsistency and misinformation on the API versioning were fully fulfilled.
- **API documentation** - If documentation automation was been improved.
- **API schema registry** - If there is a valid solution for a centralized location of APIs' schemas.
- **API testing** - If API testing has been improved.

- **Usability** - Make use of a questionnaire to evaluate satisfaction with the developed solution and the level of convenience of its usage.

6.4.2 Sources of information

The measurement of the evaluation indicators identified in the previous section is conducted based on the identified following sources of information:

- **Questionnaire** - both software developers and key stakeholders answered a questionnaire with the purpose of gathering feedback regarding the fulfilment of the main goals of the work. This provided valuable insights to conduct the evaluation.
- **Implementation** - Conclusions were drawn from the implemented proof of concept, highlighting the strengths and weaknesses of the implemented solution.

6.5 Description of the evaluation methodology

The application of an evaluation methodology serves the purpose of assessing the stated hypotheses of Chapter 6.3 and their associated objectives. To achieve this, it is essential to utilize the defined evaluation indicators of Section 6.4.1 and information sources discussed in Section 6.4.2.

By leveraging these indicators and sources, a comprehensive evaluation and analysis of the work presented in this document can be conducted.

6.5.1 Methodology - questionnaire

The evaluation methodology process begins with the development of a questionnaire to collect insights into the main problems and difficulties present in Flutter Entertainment's current API system. This questionnaire was sent after the implementation to both developers and stakeholders so that it would be possible to addresses key aspects related to the main objectives of the work, including API lifecycle management, versioning, documentation handling, schema registry, testing, and usability of the solution.

By soliciting responses related to these specific areas, the questionnaire serves as a valuable tool for gathering relevant information and feedback necessary for the evaluation process.

6.5.2 Methodology - Implementation

Through the implementation process, it was determined whether or not there were notable improvements in relation to the various objectives defined. Additionally, the implementation provided valuable insights into the challenges encountered while using the solution and facilitated a comprehensive understanding of the level of accomplishment for each objective.

6.6 Evaluation results

Upon concluding the implementation phase, a thorough assessment of the accomplished work was conducted, aiming to determine the level of satisfaction and understand which objectives were achieved. To reach this, the responses from the questionnaire were analyzed, and a comprehensive evaluation of the implementation process was undertaken, incorporating critical thinking and analysis.

6.6.1 Results - questionnaire

To evaluate the level of satisfaction regarding the developed work, a questionnaire was prepared and included in Appendix E. This questionnaire was developed using the Google Forms platform, taking into account previous experience with it, and the responses were given by both software developers and stakeholders who had interacted with the designed solution.

The questionnaire aims to assess the satisfaction level across various topics, including API lifecycle management, API versioning, API documentation, API schema registry, and API testing, being these the areas targeted for improvement in this work. It consists of three sections: the first section gathers the users' satisfaction levels for each topic before the application of the AsyncAPI specification, while the second section captures their satisfaction levels after its application, and lastly the third evaluates the solution's usability.

The primary purpose of these questions is to assess whether any improvements were observed and to identify the objectives that were achieved.

The questionnaire was completed by a total of eleven individuals that interacted with the developed work. With this said Figure 6.1 presents the evaluations received for each topic, specifically related to the user experience, before the adoption of AsyncAPI.

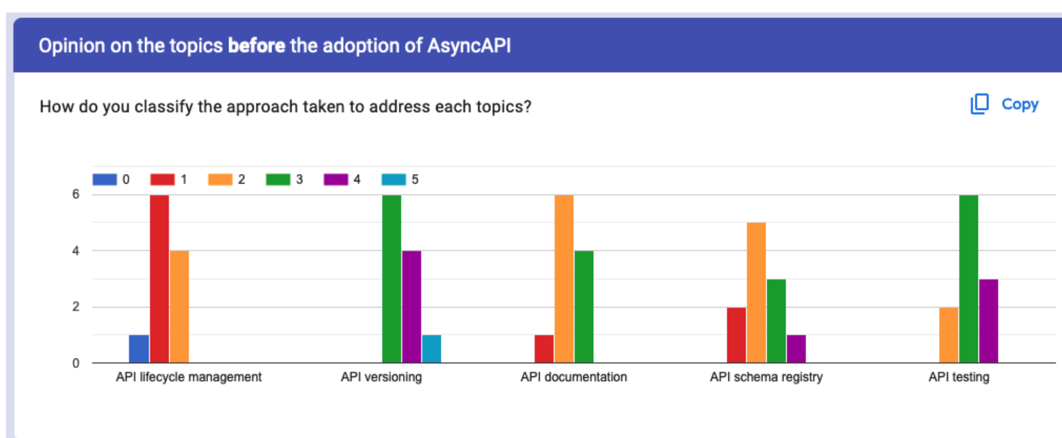


Figure 6.1: Questionnaire - section before AsyncAPI results.

Upon observation, it is evident that the topic with the lowest evaluation is API lifecycle management, closely followed by API documentation and API schema registry, which also received comparatively lower evaluations.

Moving on to the subsequent section, the results of the developed work are presented in Figure 6.2, demonstrating a positive outcome.

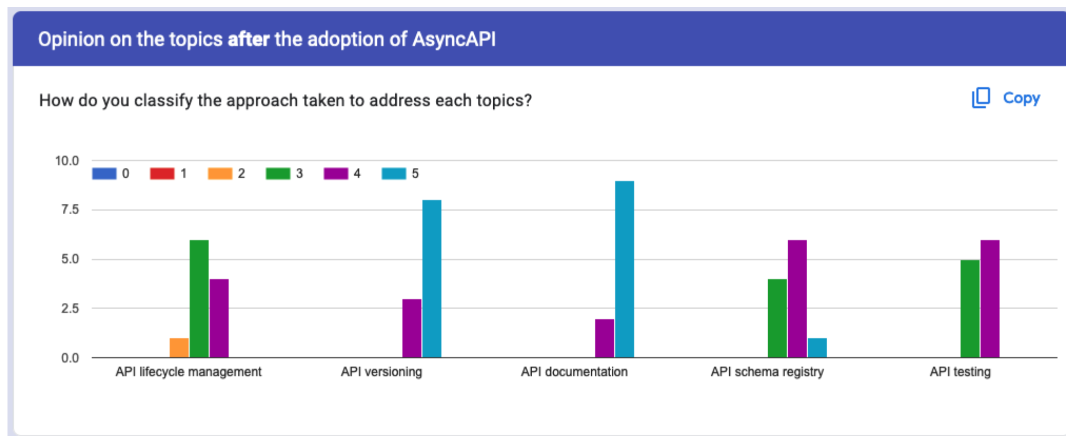


Figure 6.2: Questionnaire - section after AsyncAPI results.

The topics receiving the highest evaluations were API versioning and API documentation, with the majority of ratings given being the maximum value of the scale, 5. Interestingly, API documentation showed the highest increase, while API testing had the lowest increase of all topics. However, it is worth noting that API lifecycle management, despite experiencing an increase of approximately 2 rate scales, still received the lowest overall evaluation compared to the other topics.

These results can be attributed to the fact that AsyncAPI, the chosen solution, primarily focuses on documentation handling. Although AsyncAPI was applied to address the other topics as well, the specific solutions for each topic were not provided by default within the AsyncAPI specification. Instead, the approach for each other topic was developed based on a combination of the retrieved theoretical literature and the details outlined in the AsyncAPI schema specification.

Lastly, the results for the overall usability of the developed proof of concept can be seen in Figure 6.3.

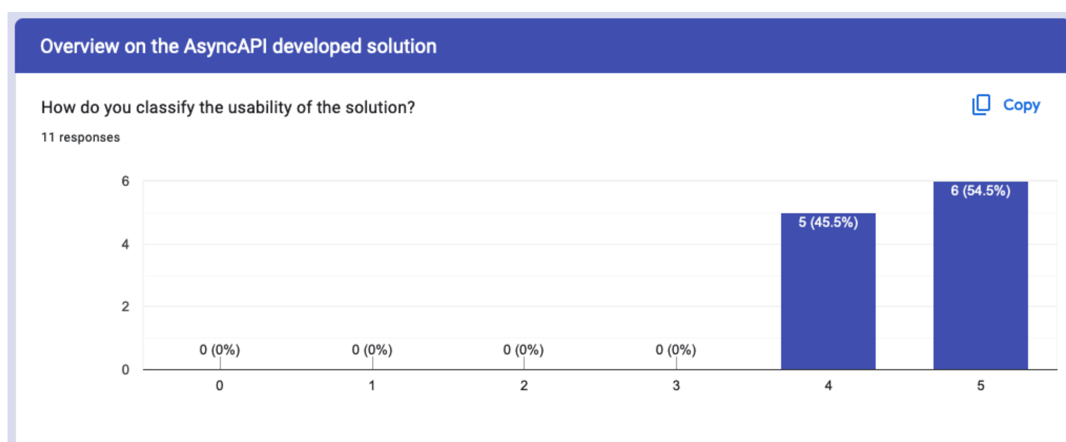


Figure 6.3: Questionnaire - section for solution usability results.

While there is still room for improvement, the overall result is significantly positive. Possible factors contributing to this outcome could be associated with the GitHub workflow

implemented to automate the documentation generation process and the utilization of the makefile that simplifies local validation and documentation generation.

6.6.2 Results - implementation

Throughout the development phase, a clearer understanding was gained regarding the chosen proof of concept solution and how it responds to the objectives, primarily due to encountering various challenges during its implementation. In light of this, below are the identified drawbacks along with their explanations:

- **API lifecycle management** - One notable drawback is that despite defining specific API lifecycle stages to be followed, the current stage of an API lifecycle can only be observed within its documentation. There is currently no centralized platform or mechanism to easily track and view all APIs along with their respective lifecycles as there is for the API schemas.
- **API schema registry** - Despite significant improvements in this topic, it still necessitates manual interventions from developers as teams are responsible for manually updating their API schemas in the schema registry repository. Ideally, the schemas should be automatically updated based on the API endpoints, but achieving this automation has not been possible thus far.
- **API testing** - The API testing using only the AsyncAPI specification proved to be impracticable. Although AsyncAPI does provide the option for testing, it requires the usage of an external tool called Microcks, which is contrary to what was initially assumed during the research phase.
- **Open Source** - The usage of both open source AsyncAPI and Microcks throughout the development process posed certain challenges. It was evident the lack and outdated documentation and some functionalities were abandoned over time and did not evolve alongside the rest of the tool's features. However, it is important to highlight that by being open source, it was possible to address these challenges by opening issues and submitting pull requests to implement or fix the required functionalities. On the other hand, the community surrounding these tools proved to be immensely helpful and supportive during the implementation phase.
- **Solution as a hub** - The APIs' specifications will be available at the repository level, which enables documentation to be integrated within the APIs' code. However, this approach lacks a centralized solution for API identification, which is considered a drawback. On the other hand, the APIs centralized hub can be achieved by integrating the AsyncAPI schema specification files of APIs into Backstage[1], an open platform that offers the capability to centralize and manage APIs.

[1] https://backstage.io/

Chapter 7

Conclusions

This section presents the conclusions to be drawn from the work carried out considering the initial objective of studying approaches to maintaining interoperability with systems' evolution. Additionally, it includes a critical analysis of the work carried out, once it was acknowledged areas that require improvement. In the end, the planned future work is outlined to enhance the topics that exhibited lower levels of confidence in the developed work.

7.1 Achieved objectives

This work aimed to address the existing challenges faced by Flutter Entertainment concerning the evolution of its APIs system. Therefore, the main objective of this study was to conduct a case study on how to effectively manage API evolution while ensuring system interoperability, having this been successfully achieved. ChatGPT

To fulfil this objective, three key phases were meticulously executed. The first phase involved conducting an extensive exploration of the relevant theoretical concepts, which laid a strong foundation of knowledge. This objective was further divided into sections, each of which was thoroughly studied and documented. With this said, it is possible to report that the *Gain an understanding of the relevant theoretical concepts* objective has been successfully achieved with a high level of satisfaction.

The second objective aimed to conduct extensive research on the available solutions and implement a proof of concept based on the solutions that were deemed to be the best fit for addressing the problem. In terms of studying the available solutions, this objective has been successfully achieved with a high level of fulfilment. A detailed comparison of the relevant solutions was conducted and thoroughly documented.

Regarding the implementation of the proof of concept, as mentioned in the conclusion of Chapter 6.6, there is certainly room for improvement, particularly in the area of API lifecycle management. Therefore, while this objective has been fully completed, the level of satisfaction of the *Investigate solutions that enable the preservation of system interoperability during the evolution of APIs.* objective is moderate to high, as there are areas that can still be enhanced to further improve the proof of concept.

Lastly, a critical evaluation of the developed solution was performed. This involved the exploration of a solution for testing the APIs and a possible solution has been explored and documented. The evaluation was high, although some aspected could be improved. With this, the objective *Criticism of the developed solution* was proven to be fully addressed with a high level of satisfaction.

In conclusion, the primary objective of this work has been successfully accomplished by producing a functional proof of concept that effectively addresses all the required topics. The developed solution has proven to facilitate the interaction of developers and stakeholders with APIs, leading to notable improvements in all the main areas examined, as evidenced in the findings presented in Chapter 6.6.

7.2 Critical analysis of work carried out

In conclusion, it can be affirmed that the primary objective of this work has been accomplished by successfully implementing a proof of concept that enhances the interoperability of a system with its evolution. Throughout the process, various challenges were encountered in relation to the implementation of the chosen proof of concept solution. However, all the necessary steps towards achieving the main objective were addressed and fulfilled, although with different levels of success.

As demonstrated by the results presented in Chapter 6.6, there is still room for improvement within the studied and implemented solution, particularly concerning the topics of API lifecycle management and schema registry. Consequently, Chapter, 7.3 provides detailed information on the next steps to be pursued to further enhance the solution.

7.3 Future work

In terms of future work, there are aspects that can be improved, including API lifecycle management, API schema registry, and the need for a centralized API hub within the implemented proof of concept.

To enhance API lifecycle management, it is important to explore additional approaches that offer better visibility into the various stages of an API's lifecycle. One potential path to investigate is the chosen solution for the API hub, which may provide additional functionalities in this area, but further research should be conducted to delve deeper into this topic.

Concerning the API schema registry, there is an opportunity to benefit from the existing Protobuf schema definitions. AsyncAPI does support the serialization of Protobuf schemas, and integrating this type of definition into the specification could be beneficial. This integration would reduce the need for manual operations and decrease the likelihood of documentation becoming outdated.

Lastly, it is necessary to study and implement a centralized hub for Flutter Entertainment's APIs. This can be achieved by exploring the utilization of Backstage, an open platform that supports the importation of AsyncAPI schema specification files. Backstage serves as a portal and can effectively function as a centralized hub for APIs.

By addressing these future work aspects, the overall functionality and effectiveness of the solution can be significantly improved, further enhancing the management and interoperability of Flutter Entertainment's system.

Bibliography

(2023). url: https://c4model.com/ (visited on 05/27/2023).

About ACM DL (2022). url: https://dl.acm.org/about (visited on 10/06/2022).

About Google Scholar (2022). url: https://scholar.google.com/intl/en/scholar/about.html (visited on 10/06/2022).

Abukwaik, Hadil, Davide Taibi, and Dieter Rombach (2014). “Interoperability-related architectural problems and solutions in information systems: A scoping study”. In: *Software Architecture*, pp. 308–323. doi: 10.1007/978-3-319-09970-5_27.

Akbulut, Akhan and Harry G. Perros (2019). “Software versioning with microservices through the API Gateway Design Pattern”. In: *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*.

Alsaleh, Saad and Haryani Haron (2016). “The most important functional and non-functional requirements of knowledge sharing system at public academic institutions: A case study”. In: *Lecture Notes on Software Engineering*, pp. 157–161.

Amazon Web Services (2023a). url: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (visited on 01/28/2023).

– (2023b). *Documenting rest apis - amazon API gateway*. url: https://docs.aws.amazon.com/apigateway/latest/developerguide/api-gateway-documenting-api.html (visited on 01/28/2023).

Apache Software Foundation (2022). *Getting started: Apache APISIX® – cloud-native API gateway*. url: https://apisix.apache.org/docs/apisix/getting-started/ (visited on 01/27/2023).

Association for Computing Machinery (2023). *The code affirms an obligation of computing professionals to use their skills for the benefit of society.* url: https://www.acm.org/code-of-ethics (visited on 01/02/2023).

AsyncAPI Initiative (2022). *2.0.0: ASYNCAPI Initiative for Event-driven apis*. url: https://www.asyncapi.com/docs/reference/specification/v2.0.0 (visited on 01/28/2023).

Aziz, Omer et al. (2020). “Research trends in enterprise service bus (ESB) applications: A systematic mapping study”. In: *IEEE Access* 8, pp. 31180–31197. doi: 10.1109/access.2020.2972195.

B-on (2022). url: https://www.b-on.pt/ (visited on 10/06/2022).

Baca, Dejan, Bengt Carlsson, and Lars Lundberg (2008). “Evaluating the cost reduction of static code analysis for software security”. In: *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*. doi: 10.1145/1375696.1375707.

Blip | Blip, a flutter company (Mar. 2022). url: https://blip.pt/about-us/ (visited on 10/06/2022).

Brito, Gleison, Thais Mombach, and Marco Tulio Valente (2019). “Migrating to graphql: A practical assessment”. In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. doi: 10.1109/saner.2019.8667986.

Chappell, David (2008). *What is Application Lifecycle Management?* url: http://www.davidchappell.com/WhatIsALM--Chappell.pdf (visited on 12/08/2022).

Curbera, F. et al. (2002). "Unraveling the web services web: An introduction to soap, WSDL, and Uddi". In: *IEEE Internet Computing* 6.2, pp. 86–93. doi: 10.1109/4236.991449.

Diego, Jesus de (June 2021). *API contract testing, tools and Impressions*. url: https://medium.com/adidoescode/api-contract-testing-tools-and-impressions-1eaa18bc2bda (visited on 01/14/2023).

Exame: E as melhores empresas Para Trabalhar São... (Oct. 2022). url: https://visao.sapo.pt/exame/melhores-empresas-para-trabalhar/2022-10-27-e-as-melhores-empresas-para-trabalhar-sao/ (visited on 10/06/2022).

Fielding, Roy T. and Richard N. Taylor (2000). "Principled design of the modern web architecture". In: *Proceedings of the 22nd international conference on Software engineering - ICSE '00*. doi: 10.1145/337180.337228.

Fischer, Thomas et al. (2019). "RPC based framework for partitioning IOT security software for trusted execution environments". In: *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. doi: 10.1109/iemcon.2019.8936247.

Flutter Entertainment plc (Mar. 2022). *Annual Report and Accounts 2021*. url: https://www.flutter.com/media/dvvn0ith/flutter-entertainment-plc-annual-report-2021.pdf (visited on 02/05/2023).

Flutter Entertainment plc Annual Report and Accounts 2021 (2022). url: https://www.flutter.com/media/dvvn0ith/flutter-entertainment-plc-annual-report-2021.pdf?ftag=MSFd61514f (visited on 10/18/2022).

Garrote, Antonio (Jan. 2022). *ASYNCAPI and openapi: An API modeling approach*. url: https://engineering.salesforce.com/asyncapi-and-openapi-an-api-modeling-approach-db9873695910/ (visited on 01/28/2023).

Granli, William et al. (Aug. 2015). *The driving forces of API Evolution: Proceedings of the 14th International Workshop on principles of software evolution*. url: https://dl.acm.org/doi/pdf/10.1145/2804360.2804364 (visited on 10/21/2022).

IBM (2016). *Innovation in the API economy - ibm.com*. url: https://www.ibm.com/downloads/cas/OXV3LYLO (visited on 10/21/2022).

IEEE Xplore (2022). url: https://ieeexplore.ieee.org/Xplore/home.jsp (visited on 10/06/2022).

Jacot, Allen D. (2009). *JD Edwards Enterpriseone: The complete reference*. McGraw-Hill.

Jain, Jagdeep (2022). *Learn API testing: Norms, practices, and guidelines for building effective Test Automation*. Apress.

Janssen, Marijn, Elsa Estevez, and Tomasz Janowski (2014). "Interoperability in big, open, and linked data-organizational maturity, capabilities, and data portfolios". In: *Computer* 47.10. doi: 10.1109/mc.2014.290.

Jesus Ekie, Yapo, Bassirou Gueye, and Ibrahima Niang (2021). "A comparative analysis of SOAP and rest web service composition based on performance in local and Remote Cloud Environments". In: *The 4th International Conference on Networking, Information Systems amp Security.* doi: 10.1145/3454127.3457621.

Koen, Peter A. et al. (2002). "Fuzzy Front End: Effective Methods, Tools, and Techniques". In.

Koschel, Arne et al. (2019). "Restfulness of apis in the wild". In: *2019 IEEE World Congress on Services (SERVICES)*.

Kreps, Jay, Neha Narkhede, and Jun Rao (June 2011). "Kafka: a Distributed Messaging System for Log Processing". In: url: https://pages.cs.wisc.edu/~akella/CS744/F17/838-CloudPapers/Kafka.pdf.

Landeiro, Mafalda Isabel and Isabel Azevedo (2020). "Analysis of GraphQL performance: a case study". In: *Software Engineering for Agile Application Development*.

Lapierre, Jozée (2000). "Customer-perceived value in industrial contexts". In: *Journal of Business and Industrial Marketing* 15.2/3. doi: 10.1108/08858620010316831.

Lee, Yunhyeok and Yi Liu (2022). "Using refactoring to migrate rest applications to grpc". In: *Proceedings of the ACM Southeast Conference*. doi: 10.1145/3476883.3520220.

Martin-Lopez, Alberto (2020). "AI-driven web API testing". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. doi: 10.1145/3377812.3381388.

Neap, Halil Shevket and Tahir Celik (1999). "Value of a Product: A Definition". In: *International Journal of Value-Based Management* 12.2, pp. 181–191. doi: 10.1023/a:1007718715162.

Newman, Sam (2021). *Building microservices: Designing fine-grained systems*. O'Reilly.

Nicola, Susana, Eduarda Pinto Ferreira, and J. J. Ferreira (2012). "A novel framework for modeling value for the customer, an essay on negotiation". In: *International Journal of Information Technology and Decision Making* 11.03, pp. 661–703. doi: 10.1142/s0219622012500162.

Novak, Jernej, Andrej Krajnc, and Rok Žontar (2010). *Taxonomy of static code analysis tools*. url: https://ieeexplore.ieee.org/document/5533417/ (visited on 01/14/2023).

Pattigulla, Manikanta (2020). *Web api validation*. url: https://www.c-sharpcorner.com/article/learn-about-web-api-validation/ (visited on 01/14/2023).

Peffers, Ken et al. (2006). "The design science research process: A model for producing and presenting information systems research". In: *Proceedings of First International Conference on Design Science Research in Information Systems and Technology DESRIST*.

Pigneur, Yves and Alexander Osterwalder (2010). *Business model generation: A handbook for visionaries, game changers, and Challengers*. John Wiley and Sons.

Porcello, Eve and Alex Banks (2018). *Learning graphql: Declarative data fetching for modern web apps*. O'Reilly.

Puspitasari, N et al. (2021). "Microservice API implementation for E-government service interoperability". In: *Journal of Physics: Conference Series* 1807.1, p. 012005. doi: 10.1088/1742-6596/1807/1/012005.

Quarkus (2023). *Using Apache Kafka with schema registry and avro*. url: https://quarkus.io/guides/kafka-schema-registry-avro (visited on 02/01/2023).

Ramingwong, Lachana (June 2012). "A review of requirements engineering processes, problems and models". In: *International Journal of Engineering Science and Technology*.

Red Hat (June 2020). *What is Application Lifecycle Management (ALM)?* url: https://www.redhat.com/en/topics/devops/what-is-application-lifecycle-management-alm (visited on 12/08/2022).

Reselman, Bob (Apr. 2021). *Using a schema registry to ensure data consistency between microservices*. url: https://www.redhat.com/architect/schema-registry (visited on 02/01/2023).

Rich, Nick (2000). *Value Analysis, Value Engineering*. url: https://www.urenio.org/tools/en/value_analysis.pdf (visited on 02/04/2023).

Richards, Mark (2015). *Software Architecture Patterns*. O'Reilly Media.

Richardson, Leonard, Mike Amundsen, and Sam Ruby (2013). *RESTful Web APIs*. O'Reilly.

Robillard, Martin P. et al. (2013). "Automated API property inference techniques". In: *IEEE Transactions on Software Engineering* 39.5, pp. 613–637. doi: 10.1109/tse.2012.63.

Saaty, Thomas L. (1980). *The Analytic Hierarchy Process: Planning, priority setting, resource allocation*. McGraw-Hill International Book Co.

Saaty, Thomas Lorie (1990). *Decision making for leaders: The Analytic hierarchy process for decisions in a complex world*. RWS Year.

Sohan, S.M., Craig Anslow, and Frank Maurer (2015). "A case study of web api evolution". In: *2015 IEEE World Congress on Services*. doi: 10.1109/services.2015.43.

Strengholt, Piethein (2020). *Data Management at Scale: Best Practices for Enterprise Architecture*. O'Reilly Media.

Tavares, Nírondes A. and Samyr Vale (2013). "A model driven approach for the development of semantic restful web services". In: *Proceedings of International Conference on Information Integration and Web-based Applications and Services - IIWAS '13*. doi: 10.1145/2539150.2539193.

Tüzün, Eray et al. (2019). "Adopting integrated application lifecycle management within a large-scale software company: An Action Research Approach". In: *Journal of Systems and Software* 149, pp. 63–82. doi: 10.1016/j.jss.2018.11.021.

Valle, Pedro Henrique (June 2021). "Architectural decision-making on interoperability in software-intensive systems". In: doi: 10.11606/t.55.2021.tde-23062021-141447.

Vasudevan, Keshav (Mar. 2017). *What is API lifecycle management?* url: https://swagger.io/blog/api-strategy/what-is-api-lifecycle-management/ (visited on 12/08/2022).

Wang, John (Oct. 2019). *Rest API best practices for interoperability*. url: https://medium.com/ringcentral-developers/rest-api-best-practices-for-interoperability-ce2c76d99312 (visited on 01/22/2023).

Wang, Stephanie, Benjamin Hindman, and Ion Stoica (2021). "In reference to RPC: It's Time to Add Distributed Memory". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. doi: 10.1145/3458336.3465302. (Visited on 01/22/2023).

Watson, Robert et al. (2013). "API documentation and software community values". In: *Proceedings of the 31st ACM international conference on Design of communication*. doi: 10.1145/2507065.2507076.

Wen, Ming (Jan. 2022). *Why is Apache APISIX the best API gateway?* url: https://api7.ai/blog/why-is-apache-apisix-the-best-api-gateway (visited on 01/27/2023).

Westerveld, Dave (2021). *API Testing and Development with Postman*. Packt Publishing. isbn: 9781800569201; 1800569203.

Yang, Jinqiu et al. (2018). "Towards extracting web API specifications from documentation". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. doi: 10.1145/3196398.3196411.

Zuo, Xianyu et al. (2020). "An API gateway design strategy optimized for persistence and coupling". In: *Advances in Engineering Software* 148, p. 102878. doi: 10.1016/j.advengsoft.2020.102878.

Appendix A

List of architectural strategies from "Architectural decision-making on interoperability in software-intensive systems"

This Appendix shows the complete list of identified architectural strategies from Valle 2021 study to promote interoperability. This list is organized by the architectural solutions according to the corresponding type of architectural strategy and is represented in Figure A.1.

Approaches	Name of Strategy
Architectural Styles	Microservices
	Service-Oriented Architecture (SOA)
	Layer
	Representational State Transfer (REST)
	File Transfer
	Shared Database
	Remote Procedure Invocation (RPI)
	Messaging
Architectural Patterns	Pipes and Filters &Pipeline
	Scatter/Gather
	Centralized Architecture
	Mediator
	Construct Monitor
	Bus
	Adapter
	Wrapper
	Broker
	Facade
Architectural Tactics	Orchestrate
	Coreography
	Discover Service
	Tailor Interface
Techniques	Business Process Execution Language (BPEL)
	Web Service Level Agreement (WSLA)
	OGSA (Migration)
	Business Process Modeling Notation (BPMN)
	XML Process Defintion Language (XPDL)
	APIs
	Taxonomies
	Ontologies
	Thesaurus
	Web Services Languages
	JSON, YAML, XML
	Web Services Security (WS-Security)
	OGSA-DATA Access
	Integration (OGSA-DAI)
	Simple Object Access Protocol (SOAP)

Figure A.1: Complete list of identified architectural strategies from the literature review “Architectural decision-making on interoperability in software-intensive systems”.

Appendix B

AHP - analysis auxiliary calculations

B.1 API lifecycle management

Table B.1: API lifecycle management - comparison matrix.

	S1	S2	S3	S4
S1	1	3	2	3
S2	1/3	1	1/3	1
S3	1/2	3	1	3
S4	1/3	1	1/3	1
Sum	2,67	8,00	2,67	8,00

Table B.2: API lifecycle management - normalized comparison matrix and relative priority.

	S1	S2	S3	S4	Relative priority
S1	0,3750	0,3750	0,3750	0,3750	0,375000
S2	0,1250	0,1250	0,1250	0,1250	0,125000
S3	0,3750	0,3750	0,3750	0,3750	0,375000
S4	0,1250	0,1250	0,1250	0,1250	0,125000
Sum	1,0000	1,0000	1,0000	1,0000	1,000000

1. **Priority vector matrix (Ax):**

Table B.3: API lifecycle management - Priority vector matrix.

	Resulted value
S1	1,8081
S2	0,4983
S3	1,2751
S4	0,4983

2. **Intermediate vector:**

Table B.4: API lifecycle management - Intermediate vector matrix.

	Resulted value
S1	4,116418
S2	4,028269
S3	4,069038
S4	4,028269

3. **Self Value (**λmax**)**:

$$\lambda max = \frac{4,116418 + 4,028269 + 4,069038 + 4,028269}{4}$$
$$\lambda max = 4,06050$$

4. **Consistency Index (**CI**):**

$$CI = \frac{4,06050 - 4}{4 - 1}$$
$$CI = 0,020166$$

5. **Consistency Ratio (**CR**):**

$$CR = \frac{CI}{IR}$$
$$CR = \frac{0,022407}{0,9}$$
$$CR =$$

B.2 API documentation handling

Table B.5: API documentation handling - comparison matrix.

	S1	**S2**	**S3**	**S4**
S1	1	1/5	3	1/3
S2	5	1	7	3
S3	1/3	1/7	1	1/5
S4	3	1/3	5	1
Sum	9,33	1,68	16,00	4,53

Table B.6: API documentation handling - normalized comparison matrix and relative priority.

	S1	S2	S3	S4	Relative priority
S1	0,1071	0,1193	0,1875	0,0735	0,121873
S2	0,5357	0,5966	0,4375	0,6618	0,557892
S3	0,0357	0,0852	0,0625	0,0441	0,056890
S4	0,3214	0,1989	0,3125	0,2206	0,263345
Sum	1,000	1,000	1,000	1,000	1,000000

1. **Priority vector matrix (Ax):**

Table B.7: API documentation handling - Priority vector matrix.

	Resulted value
S1	0,4919
S2	2,3555
S3	0,2299
S4	1,0994

2. **Intermediate vector:**

Table B.8: API documentation handling - Intermediate vector matrix.

	Resulted value
S1	4,036200
S2	4,222175
S3	4,040829
S4	4,174659

3. **Self Value (λmax)**:

$$\lambda max = \frac{4,036200 + 4,222175 + 4,040829 + 4,174659}{4}$$

$$\lambda max = 4,11847$$

4. **Consistency Index (CI):**

$$CI = \frac{4,11847 - 4}{4 - 1}$$

$$CI = 0,039489$$

5. **Consistency Ratio (CR):**

$$CR = \frac{CI}{IR}$$

$$CR = \frac{0,039489}{0,9}$$

$$CR = 0,043876$$

B.3 API schema registry

Table B.9: API schema registry - comparison matrix.

	S1	S2	S3	S4
S1	1	3	1	5
S2	1/3	1	1/3	3
S3	1	3	1	5
S4	1/5	1/3	1/5	1
Sum	2,53	7,33	2,53	14,00

Table B.10: API schema registry - normalized comparison matrix and relative priority.

	S1	S2	S3	S4	Relative priority
S1	0,3947	0,4091	0,3947	0,3571	0,388927
S2	0,1316	0,1364	0,1316	0,2143	0,153452
S3	0,3947	0,4091	0,3947	0,3571	0,388927
S4	0,0789	0,0455	0,0789	0,0714	0,068694
Sum	1,000	1,000	1,000	1,000	1,000000

1. **Priority vector matrix (Ax):**

Table B.11: API schema regitry - Priority vector matrix.

	Resulted value
S1	1,5817
S2	0,6188
S3	1,5817
S4	0,2754

2. **Intermediate vector:**

Table B.12: API schema registry - Intermediate vector matrix.

	Resulted value
S1	4,066784
S2	4,032665
S3	4,066784
S4	4,009287

3. **Self Value (**λmax**)**:

$$\lambda max = \frac{4,066784 + 4,032665 + 4,066784 + 4,009287}{4}$$
$$\lambda max = 4,04388$$

4. **Consistency Index (**CI**):**

$$CI = \frac{4,04388 - 4}{4 - 1}$$
$$CI = 0,014627$$

5. **Consistency Ratio (**CR**):**

$$CR = \frac{CI}{IR}$$
$$CR = \frac{0,014627}{0,9}$$
$$CR = 0,016252$$

B.4 API versioning

Table B.13: API versioning - comparison matrix.

	S1	S2	S3	S4
S1	1	1/3	1/3	1/5
S2	3	1	3	1/3
S3	3	1/3	1	1/5
S4	5	3	5	1
Sum	12,00	4,67	9,33	1,73

Table B.14: API versioning - normalized comparison matrix and relative priority.

	S1	S2	S3	S4	Relative priority
S1	0,0833	0,0714	0,0357	0,1154	0,076465
S2	0,2500	0,2143	0,3214	0,1923	0,244505
S3	0,2500	0,0714	0,1071	0,1154	0,135989
S4	0,4167	0,6429	0,5357	0,5769	0,543040
Sum	1,000	1,000	1,000	1,000	1,000000

1. **Priority vector matrix (*Ax*):**

Table B.15: API versioning - Priority vector matrix.

	Resulted value
S1	0,3119
S2	1,0629
S3	0,5555
S4	2,3388

2. **Intermediate vector:**

Table B.16: API versioning - Intermediate vector matrix.

	Resulted value
S1	4,079042
S2	4,347066
S3	4,084848
S4	4,306914

3. **Self Value (λmax)**:

$$\lambda max = \frac{4,079042 + 4,347066 + 4,084848 + 4,306914}{4}$$

$$\lambda max = 4,20447$$

4. **Consistency Index (CI):**

$$CI = \frac{4,20447 - 4}{4 - 1}$$

$$CI = 0,068156$$

5. **Consistency Ratio (CR):**

$$CR = \frac{CI}{IR}$$

$$CR = \frac{0,068156}{0,9}$$
$$CR = 0,075729$$

B.5 API testing

Table B.17: API testing - comparison matrix.

	S1	**S2**	**S3**	**S4**
S1	1	1	3	5
S2	1	1	3	5
S3	1/3	1/3	1	3
S4	1/5	1/5	1/3	1
Sum	2,53	2,53	7,33	14,00

Table B.18: API testing - normalized comparison matrix and relative priority.

	S1	**S2**	**S3**	**S4**	**Relative priority**
S1	0,3947	0,3947	0,4091	0,3571	0,388927
S2	0,3947	0,3947	0,4091	0,3571	0,388927
S3	0,1316	0,1316	0,1364	0,2143	0,153452
S4	0,0789	0,0789	0,0455	0,0714	0,068694
Sum	1,000	1,000	1,000	1,000	1,000000

1. **Priority vector matrix (Ax):**

Table B.19: API testing - Priority vector matrix.

	Resulted value
S1	1,5817
S2	1,5817
S3	0,6188
S4	0,2754

2. **Intermediate vector:**

Table B.20: API testing - Intermediate vector matrix.

	Resulted value
S1	4,066784
S2	4,066784
S3	4,032665
S4	4,009287

3. **Self Value (λmax)**:

$$\lambda max = \frac{4,066784 + 4,066784 + 4,032665 + 4,009287}{4}$$
$$\lambda max = 4,04388$$

4. **Consistency Index (CI):**

$$CI = \frac{4,04388 - 4}{4 - 1}$$
$$CI = 0,014627$$

5. **Consistency Ratio (CR):**

$$CR = \frac{CI}{IR}$$
$$CR = \frac{0,014627}{0,9}$$
$$CR = 0,016252$$

B.6 Pricing

Table B.21: Pricing - comparison matrix.

	S1	S2	S3	S4
S1	1	1	5	1
S2	1	1	5	1
S3	1/5	1/5	1	1/5
S4	1	1	5	1
Sum	3,20	3,20	16,00	3,20

Table B.22: Pricing - normalized comparison matrix and relative priority.

	S1	**S2**	**S3**	**S4**	**Relative priority**
S1	0,3125	0,3125	0,3125	0,3125	0,312500
S2	0,3125	0,3125	0,3125	0,3125	0,312500
S3	0,0625	0,0625	0,0625	0,0625	0,062500
S4	0,3125	0,3125	0,3125	0,3125	0,312500
Sum	1,000	1,000	1,000	1,000	1,000000

1. **Priority vector matrix (Ax):**

Table B.23: Pricing - Priority vector matrix.

	Resulted value
S1	1,2500
S2	1,2500
S3	0,2500
S4	1,2500

2. **Intermediate vector:**

Table B.24: Pricing - Intermediate vector matrix.

	Resulted value
S1	4,000000
S2	4,000000
S3	4,000000
S4	4,000000

3. **Self Value (λmax)**:

$$\lambda max = \frac{4,000000 + 4,000000 + 4,000000 + 4,000000}{4}$$
$$\lambda max = 4,000000$$

4. **Consistency Index (CI):**

$$CI = \frac{4,000000 - 4}{4 - 1}$$
$$CI = 0,000000$$

5. **Consistency Ratio (CR):**

$$CR = \frac{CI}{IR}$$

$$CR = \frac{0,000000}{0,9}$$

$$CR = 0,000000$$

Appendix C

AsyncAPI - Generated documentation

The generated HTML documentation by the AsyncAPI schema specification is demonstrated in Figures C.1, C.2, C.3 and C.4.

Figure C.1: AsyncAPI generated documentation - overview.

Operations

PUB `global.exchange.input.events`

Input topic of Ramp Events coming from Global Sportex Mapping flow.

Input topic from where Global Sportex Mapping flow requests Ramp Events to be mapped.

Accepts the following message:

Global Input Events global.input.events

Input payload of a GSM event mapping request.

APPLICATION/JSON

Payload Expand all Object uid: global.input.events.payload

apiVersion	String
action	String
id	String
eventId	String
sportId	String
hwm Expand all	Object
event Expand all	Object

Additional properties are allowed.

Headers Expand all Object

ID	String The provider id for the given payload.
CORRELATION_ID	String The correlation id of all messages related to input date.
RAMP_SPORT	Integer The Ramp Sport id. As this is the Tennis flow, it should be "13"

Examples

Payload


```
{
  "apiVersion": "string",
  "action": "string",
  "id": "string",
  "eventId": "string",
  "sportId": "string",
  "hwm": {
    "token": "string",
    "sequence": "string"
  },
  "event": {
    "competition": {
      "id": "string",
      "name": "string"
    },
    "name": "string",
    "startTime": "string",
    "sort": "string",
    "teams": [
      {
        "name": "string"
      }
    ]
  }
}
```


This example has been generated automatically.

Headers

Figure C.2: AsyncAPI generated documentation - operations.

Servers

`{dc}-mfs{vmNum}-{env}.{env}.betfair:{port}` KAFKA 2.0.1 MFS-DEV

[QA] Kafka cluster from where TMT topology get the input data.

URL Variables ^ Expand all

dc
required

String
The datacenter that the cluster belongs to.
Default value: "ie1"
Allowed values: "ie1" "ie2"

vmNum
required

String
The virtual machine identification. In this case, the cluster has 3 machines.
Default value: "001"
Allowed values: "001" "002" "003"

env
required

String
The development environment of the cluster.
Default value: "qa"
Allowed values: "qa" "nxt"

port
required

String
The port that must be used to connect to the cluster.
Default value: "9092"
Allowed values: "9092"

Security:

SECURITY.PROTOCOL: PLAINTEXT

#env:qa #kind:remote

`{dc}-mfs{vmNum}-prd.prd.betfair:{port}` KAFKA 2.0.1 MFS-PRD

[PRD] Kafka cluster from where TMT topology get the input data.

URL Variables > Expand all

Security:

SECURITY.PROTOCOL: PLAINTEXT

Figure C.3: AsyncAPI generated documentation - servers.

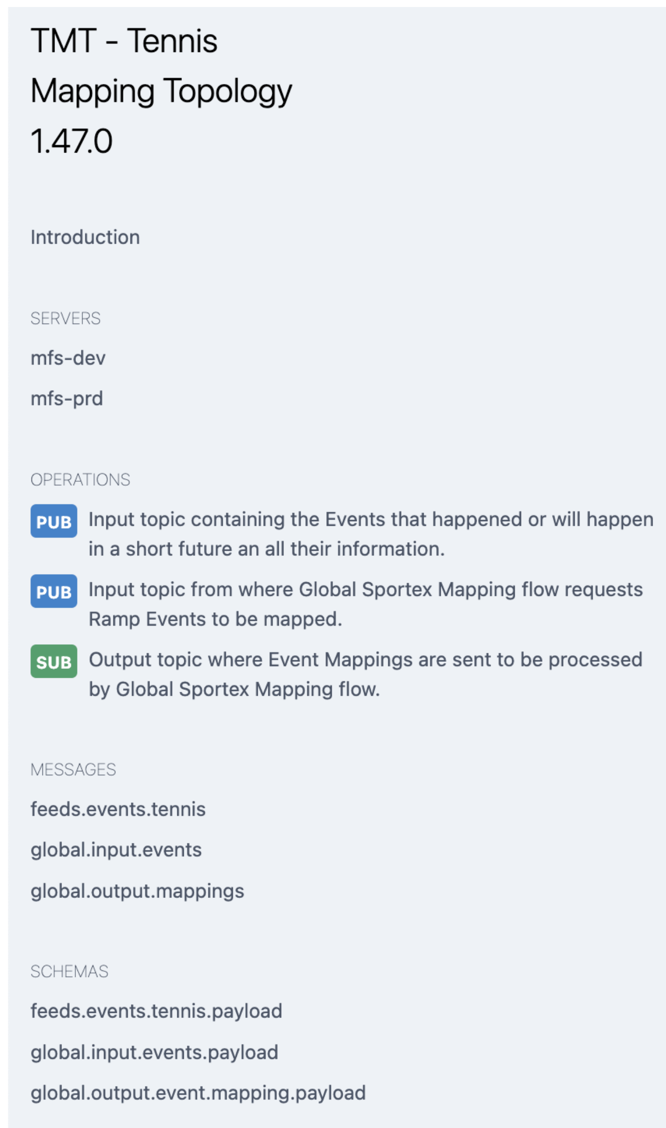
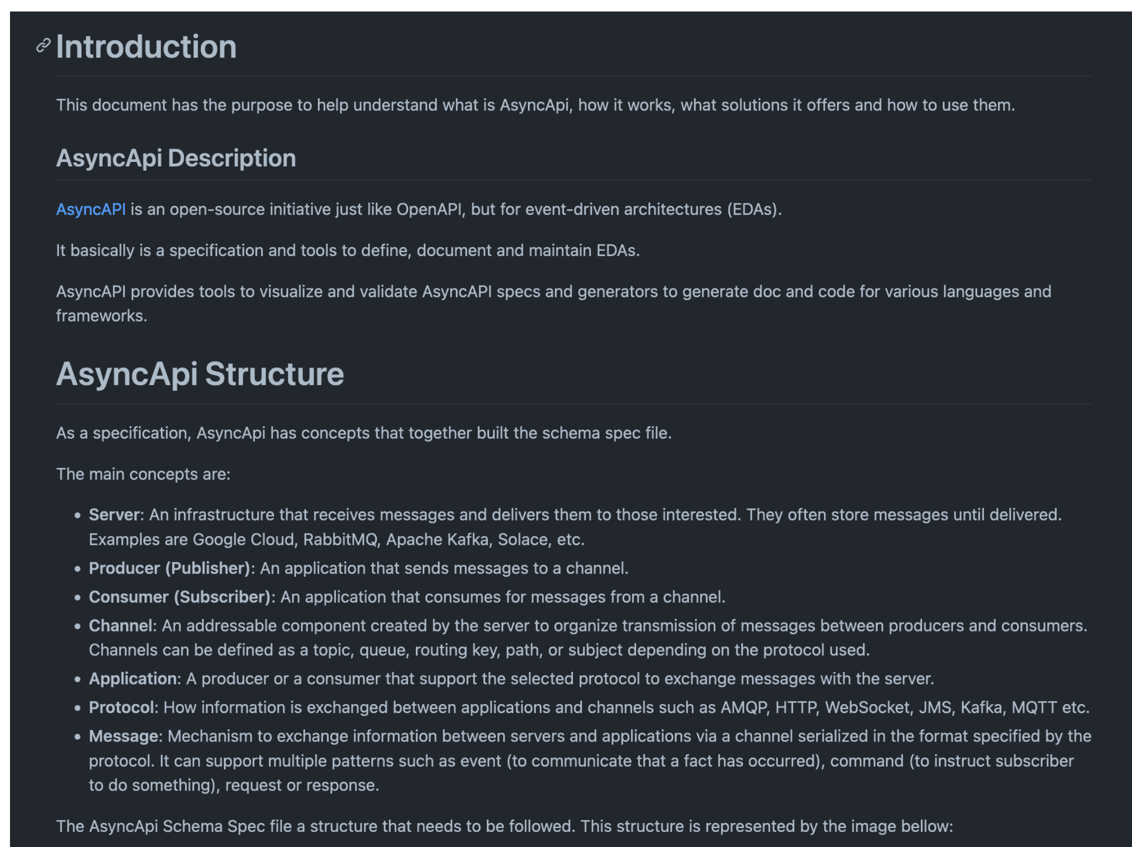


Figure C.4: AsyncAPI generated documentation - sidebar.

Appendix D

AsyncAPI - how to guide

The produced guide on how to handle AsyncAPI schema specification and best practices can be visualized in Figures D.1, D.2, D.3 and D.4.



Introduction

This document has the purpose to help understand what is AsyncApi, how it works, what solutions it offers and how to use them.

AsyncApi Description

AsyncAPI is an open-source initiative just like OpenAPI, but for event-driven architectures (EDAs).

It basically is a specification and tools to define, document and maintain EDAs.

AsyncAPI provides tools to visualize and validate AsyncAPI specs and generators to generate doc and code for various languages and frameworks.

AsyncApi Structure

As a specification, AsyncApi has concepts that together built the schema spec file.

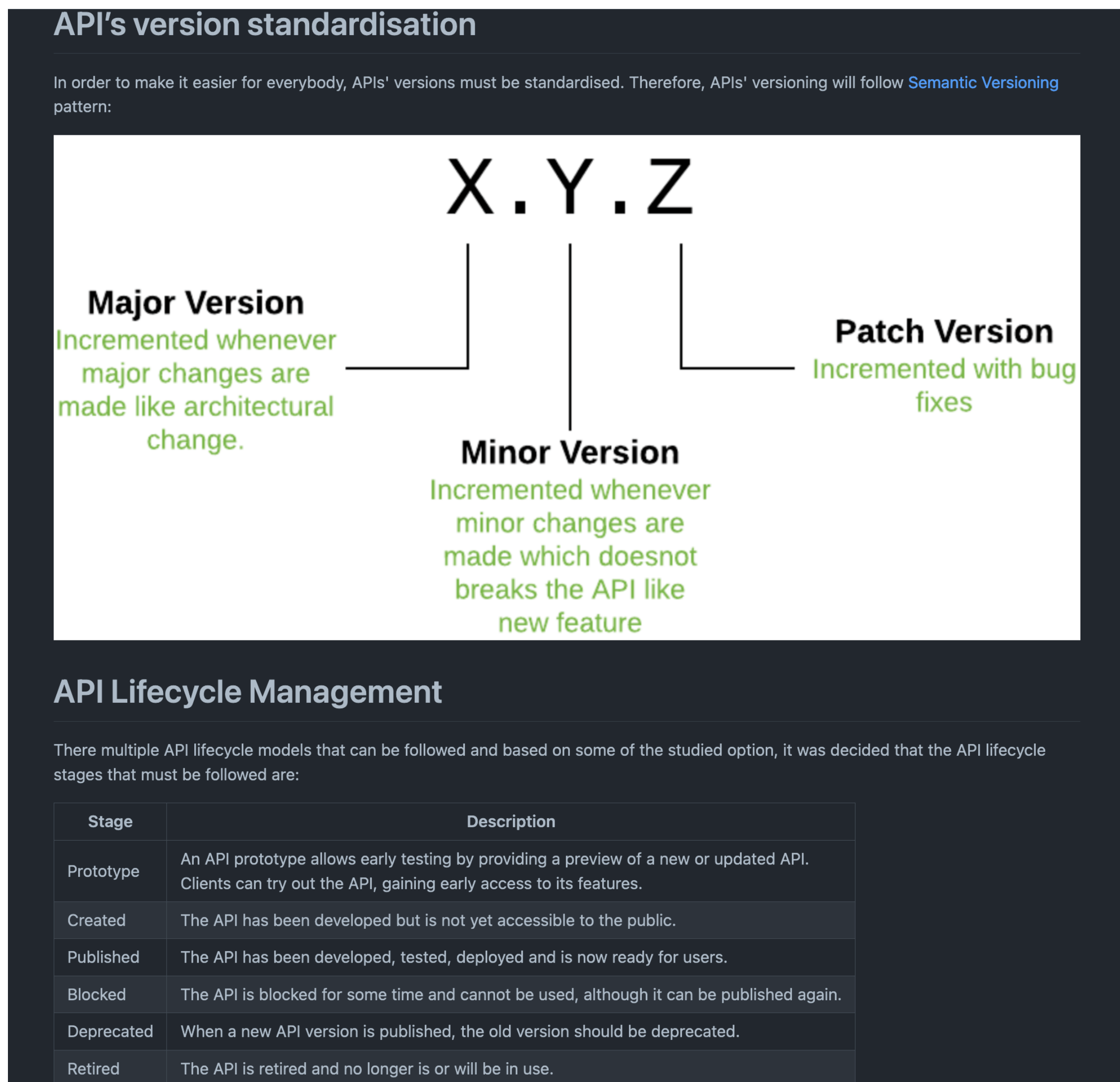
The main concepts are:

- **Server**: An infrastructure that receives messages and delivers them to those interested. They often store messages until delivered. Examples are Google Cloud, RabbitMQ, Apache Kafka, Solace, etc.
- **Producer (Publisher)**: An application that sends messages to a channel.
- **Consumer (Subscriber)**: An application that consumes for messages from a channel.
- **Channel**: An addressable component created by the server to organize transmission of messages between producers and consumers. Channels can be defined as a topic, queue, routing key, path, or subject depending on the protocol used.
- **Application**: A producer or a consumer that support the selected protocol to exchange messages with the server.
- **Protocol**: How information is exchanged between applications and channels such as AMQP, HTTP, WebSocket, JMS, Kafka, MQTT etc.
- **Message**: Mechanism to exchange information between servers and applications via a channel serialized in the format specified by the protocol. It can support multiple patterns such as event (to communicate that a fact has occurred), command (to instruct subscriber to do something), request or response.

The AsyncApi Schema Spec file a structure that needs to be followed. This structure is represented by the image bellow:

Figure D.1: AsyncAPI guide - part 1.

Figure D.2: AsyncAPI guide - part 2.



API's version standardisation

In order to make it easier for everybody, APIs' versions must be standardised. Therefore, APIs' versioning will follow Semantic Versioning pattern:

API Lifecycle Management

There multiple API lifecycle models that can be followed and based on some of the studied option, it was decided that the API lifecycle stages that must be followed are:

Stage	Description
Prototype	An API prototype allows early testing by providing a preview of a new or updated API. Clients can try out the API, gaining early access to its features.
Created	The API has been developed but is not yet accessible to the public.
Published	The API has been developed, tested, deployed and is now ready for users.
Blocked	The API is blocked for some time and cannot be used, although it can be published again.
Deprecated	When a new API version is published, the old version should be deprecated.
Retired	The API is retired and no longer is or will be in use.

Figure D.3: AsyncAPI guide - part 3.

Schema Registry

In order to centralize all APIs' schemas in a unit place, it was decided to create a repo that will have all AsyncApi payload's definitions. This repo is the flutter-schema-registry.

This way it is easy to know every APIs' schema, and we can also reduce the size of the AsyncApis' Schema Spec files.

So, if any change needs to be done, a PR should be open on this repository.

How to Generate Doc

To generate doc based on the AsyncApi Schema Spec file we can use AsyncApi's Generators.

Based on this, we could generators to generate both code and documentation, but focusing on documentation, there are the possibility to generate documentation in a MarkDown or HTML files:

1. First you'll need to install @asyncapi/cli and @asyncapi/generator
2. Then install the template you want to use (Suggestion @asyncapi/html-template and @asyncapi/markdown-template)
3. After that you are ready to go and can execute the generation commands:

```
asyncapi generate fromTemplate asyncapi_live_data_topology.yml @asyncapi/html-template
asyncapi generate fromTemplate asyncapi_event_management_topology.yml @asyncapi/markdown-template
```

As the commands are quick long and are even more parameters available, there is a makefile that to:

1. Validate: `make validate`
2. Generate HTML doc (output directory: *asyncapi/generated-doc/html*): `make generate_html`
3. Generate MarkDown doc (output directory: *asyncapi/generated-doc/md*): `make generate_md`
4. Delete all generated doc: `make delete_all_generated_file`

Figure D.4: AsyncAPI guide - part 4.

Appendix E

Questionnaire for the experimentation and evaluation

The structure of the questionnaire done for the experimentation and evaluation phase can be viewed in Figures E.1, E.2, E.3 and E.4.

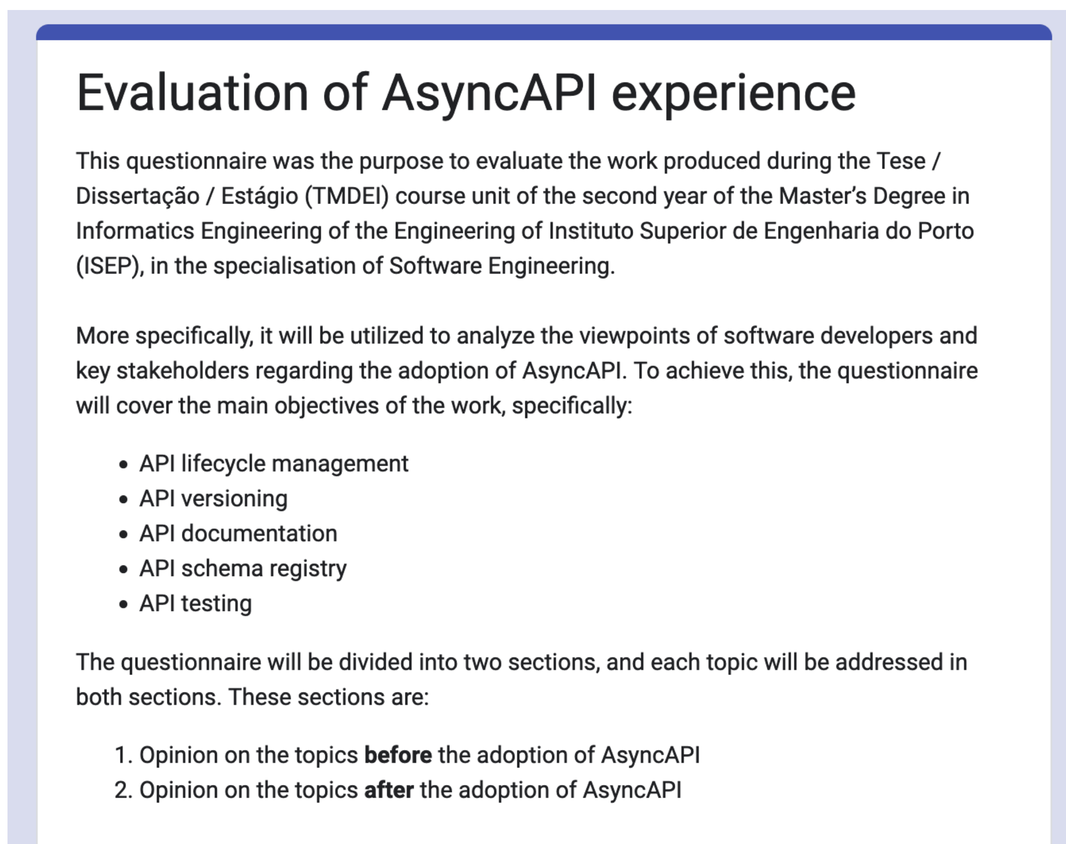


Figure E.1: Questionnaire - initial page.

Opinion on the topics **before** the adoption of AsyncAPI

In this section you should evaluate the 5 topics based on the experience **before** the adoption of AsyncAPI.
For this, you should use a scale of 1 to 10 where:

- 1 means that the topic is not being addressed at all
- 5 the topic is being perfectly addressed

How do you classify the approach taken to address each topics? *

	0	1	2	3	4	5
API lifecycle management	○	○	○	○	○	○
API versioning	○	○	○	○	○	○
API documentation	○	○	○	○	○	○
API schema registry	○	○	○	○	○	○
API testing	○	○	○	○	○	○

Figure E.2: Questionnaire - section before AsyncAPI.

Opinion on the topics **after** the adoption of AsyncAPI

In this section you should evaluate the 5 topics based on the experience **after** the adoption of AsyncAPI.
For this, you should use a scale of 1 to 10 where:

- 1 means that the topic is not being addressed at all
- 10 the topic is being perfectly addressed

How do you classify the approach taken to address each topics? *

	0	1	2	3	4	5
API lifecycle management	○	○	○	○	○	○
API versioning	○	○	○	○	○	○
API documentation	○	○	○	○	○	○
API schema registry	○	○	○	○	○	○
API testing	○	○	○	○	○	○

Figure E.3: Questionnaire - section after AsyncAPI.

Overview on the AsyncAPI developed solution

This sections intends to gather an evaluation the usability of the developed solution.
For this, you should use a scale of 0 to 5 where:

- 0 - the solution is not usable at all
- 5 - the solution is usable for anyone with zero background on AsyncAPI

How do you classify the usability of the solution? *

0	1	2	3	4	5
○	○	○	○	○	○

Figure E.4: Questionnaire - section for solution usability.