COMPREHENSIVE TESTING FRAMEWORK FOR MICROSERVICES

Vajiramal Vilochane Vidyarathne

(179355D)

M.Sc. in Computer Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2021

COMPREHENSIVE TESTING FRAMEWORK FOR MICROSERVICES

Vajiramal Vilochane Vidyarathne

(179355D)

This dissertation submitted in partial fulfilment of the requirements for the Degree of MSc in Computer Science specializing in Software Architecture

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March 2021

DECLARATION

I declare that this is my own work and this MSc Thesis Project Report does not incorporate without acknowledgment any material previously submitted for a Degree or Diploma in any other University or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgment is made in the text.

Also, I hereby grant to the University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works.

---------- ----------

B.K.V. V.Vidyarathne Date

I certify that the declaration above by the candidate is true to the best of my knowledge and that this project report is acceptable for evaluation for the MSc Thesis (CS6997).

---------- ----------

Dr. Indika Perera Date

ABSTRACT

An emerging trend has begun with the introduction of microservices architecture, transforming monolithic applications into microservices. These services are focused on smaller independent autonomous services, which encourage development, deployment, scale, and maintain each service independently with improved parallel development among multiple autonomous development teams. Unlike the monolith microservices, architecture introduces additional testing challenges.

Microservices integration testing, a crucial process that ensures the communication paths are correct while interacting with other services. Because of this unique architecture, testing integration in isolation impose a challenge due to many reasons, such as each microservice is developed and maintained by individual autonomous teams, unavailability or instability of service due to parallel continuous rapid development, deployment and difficulty in setting up all the services and seed the necessary data for testing in a local or a remote development environment. This process requires effort to initialize and maintain the resource-intensive environment. Currently, available testing approaches or tools have particular limitations. Such approaches or tools require extra effort to create, initialize and maintain with the rapidly changing unstable services and requirements. Gradually with the introduction of new services as the number of services is greater this process turns in to a development overhead.

The main objective of this research is to overcome the integration testing challenges in REST-based microservices, via a service virtualization solution based on widely adapted Open API specification. The proposed implemented solution was evaluated against the existing services. The solution is proven to mitigate the identified integration testing challenges in microservices and successfully emulates the tested producer services via the virtualization of its Open API specification. The solution facilitates reproducing the edge cases and failures, such is difficult to produce in the real services, and this helps to reduce the development time significantly. The solution can evolve with the services due to the OAS of the respective service reflects the producer services' additions or its changes. Because of this reason, unlike the existing approaches or tools, this solution does not require any maintenance via mocking, simulation by the record and replay requests or as a third-party dependency package or a library to the producer and consumer source code to reflect the service changes or additions. This report concludes by mentioning research contributions, limitations, and future works of the implemented solution.

Keywords: monolithic, microservice, REST, integration, testing, development, architecture, OAS.

ACKNOWLEDGEMENTS

I would first like to express my sincere gratitude to my research supervisor Dr. Indika Perea for the guidance, knowledge, suggestions, and encouragement throughout the research which steered for the successful completion of this research.

Secondly, I would like to thank my family members, friends, and peers for the support and encouragement for the research work.

Finally, I would like to thank my current workplace CMS (PVT) Ltd, and my colleagues for their support and encouragement.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION 1

1.1 BACKGROUND 1

1.2 MONOLITHIC ARCHITECTURE 2

1.3 MICROSERVICES ARCHITECTURE 3

1.4 COLLECTION OF SERVICES WORK TOGETHER AS A SYSTEM 7

1.5 PROBLEM STATEMENT 8

1.6 MOTIVATION 9

1.7 OBJECTIVES 10

CHAPTER 2 - LITERATURE REVIEW 11

2.1 MICROSERVICES TESTING 11

2.1.1 Unit tests 12

2.1.2 Sociable unit tests 12

2.1.3 Solitary unit tests 12

2.1.4 Integration tests 12

2.1.5 Gateway integration tests 13

2.1.6 Persistence integration tests 13

2.1.7 Component tests 13

2.1.8 Contract tests 13

2.1.9 End to end tests 14

2.1.10 Test Pyramid 14

2.2 MICROSERVICES TESTING CHALLENGES 15

2.2.1 Shared testing environment 15

2.2.2 Test kits 16

2.2.3 Test doubles 16

2.2.4 Service virtualization 16

2.3 CURRENTLY AVAILABLE TOOLS 17

2.3.1 PACT 17

2.3.2 Wire mock 17

2.3.3 Hooverfly 17

CHAPTER 3 - METHODOLOGY 18

3.1 PROPOSED TESTING FRAMEWORK 18

3.2 OPEN API SPECIFICATION STRUCTURE 19

3.2.1 Open API specification as code annotations 20

3.2.2 Paths 21

3.2.3 Parameters 22

3.2.4 Request body 23

3.2.5 Response 24

3.2.6 Testing isolated microservices using the framework *25*
3.2.7 REST API *27*
3.2.8 Request handler *27*
3.2.9 Response generator *28*
3.2.10 Report generator *28*

CHAPTER 4 - DESIGN AND IMPLEMENTATION 29
4.1 THE SCOPE OF THE IMPLEMENTATION 29
4.2 TEST FRAMEWORK IMPLEMENTATION 31
4.3 PRODUCER REGISTRATION AND CAPTURE OPEN API SPECIFICATION 36
4.4 PRODUCER SERVICE VIRTUALIZER 37
4.4.1 Request parameter validation *38*
4.4.2 Request body validation *41*
4.5 RESPONSE GENERATOR 48
4.6 VIEW PRODUCER OPEN API SPECIFICATION 52
4.7 REPORTING 54

CHAPTER 5 - EVALUATION 55
5.1 CTFM VS. EXISTING TOOLS OR TECHNOLOGIES 55
5.2 MINIMUM SERVER REQUIREMENTS AND CONFIGURATIONS 56
5.3 CONFIGURATIONS FOR WRITTEN INTEGRATION TESTS 58
5.4 REAL SERVICES VS. CTFM VIRTUALIZED SERVICES 62
5.4.1 Test results interpretation *71*
5.5 PERFORMANCE TEST 81

CHAPTER 6 - CONCLUSION 84
6.1 RESEARCH CONTRIBUTION 85
6.2 RESEARCH LIMITATIONS 86
6.3 FUTURE WORK 87

LIST OF FIGURES

Figure 1.1: Sample monolithic application layout *2*

Figure 1.2: Sample microservice internal structure *3*

Figure 1.3: Microservice communication between external services and external data stores *5*

Figure 1.4: System is formed by collection of microservices *7*

Figure 2.1: Test Pyramid *14*

Figure 3.1The high-level structure of the proposed testing framework *18*

Figure 3.2: Open API specification main structure *19*

Figure 3.3: Open API specification as code annotations *20*

Figure 3.4: Paths definitions of an Open API Specification *21*

Figure 3.5: Parameters of a request *22*

Figure 3.6: Request body schema *23*

Figure 3.7: Sample response segment *24*

Figure 3.8: Testing microservices in isolation *25*

Figure 3.9: Request activity flow *26*

Figure 4.1: CTFM abstract architecture *32*

Figure 4.2: composer.json *33*

Figure 4.3: package.json *34*

Figure 4.4: Framework folder structure *35*

Figure 4.5: Adding a new producer configuration *36*

Figure 4.6: Producer virtualize class *37*

Figure 4.7: Open API parameter schema object *38*

Figure 4.8: Schema analyzer request parameter validator *39*

Figure 4.9: Request parameter validation error response *40*

Figure 4.10: Request body as a reference *41*

Figure 4.11: Request body definition as a object *42*

Figure 4.12: Request complex payload structure definition *44*

Figure 4.13: Formation of validation rules for a given specification *45*

Figure 4.14: Request validation rule overridden example *47*

Figure 4.15: Open API response specification *48*

Figure 4.16: CTFM generated response according to response specification *49*

Figure 4.17: Expected error response only instructed with the status code *50*

Figure 4.18 Expected response with the status and overridden response *51*
Figure 4.19: Configuration view to access the specification of a producer *52*
Figure 4.20: Open API specification view of a producer *53*
Figure 4.21 User interface of the reporting module *54*
Figure 5.1 Hooverfly producer test with simulation capture mode *58*
Figure 5.2 Hooverfly consumer test configured for a specific simulation *59*
Figure 5.3 Consumer test written to test with CTFM *60*
Figure 5.4 Groovy response assertion for /card/${token}/transaction *70*
Figure 5.5: Producer services test response times *71*
Figure 5.6: Producer services test results complete summary *71*
Figure 5.7: CTFM virtualized services test response times *72*
Figure 5.8: CTFM virtualized producer services test results complete summary *72*
Figure 5.9: CTFM virtualized test results summary graphical presentation *73*
Figure 5.10: Response assertion failure due to unavailable response definition *74*
Figure 5.11: Response assertion failure due to unavailable response definition *75*
Figure 5.12: Response assertion failure due to missing response definition *76*
Figure 5.13: Response assertion failure due wrong response definition *77*
Figure 5.14: Tests failure composition *78*
Figure 5.15: CTFM results with corrected producer specification *79*
Figure 5.16: Average response time for a single user *80*
Figure 5.17: Average response time for a hundred users *81*
Figure 5.18: Average response time for a ten users *81*
Figure 5.19: Performance results summary graph *82*
Figure 6.1 composer.json *2*
Figure 6.2 Package.json *4*
Figure 6.3 Dockerfile *5*
Figure 6.4 Docker compose file *6*
Figure 6.5 Make file *7*

LIST OF TABLES

Table 4.1 Header to override validations *46*

Table 4.2: Expected response request headers with only status code *50*

Table 4.3: Expected response request headers with json schema *51*

Table 5.1 CTFM vs existing tools or technologies *55*

Table 5.2 Testing framework minimum server requirements *57*

Table 5.3: Selected services for testing *70*

Table 5.4: CTFM Virtualized test results summary *74*

Table 5.5: Testing resources and environment *81*

Table 5.6: Performance results summary *83*

APPENDICES

APPENDIX I: FRAMEWORK PACKAGES AND LIBRARIES 1

APPENDIX II: CTFM DOCKER CONFIGURATION 5

LIST OF ABBREVIATIONS AND SYMBOLS

OAS	Open API specification
API	Application programming interface
REST	Representational state transfer
HTTP	Hypertext Transfer Protocol
Groovy	Java syntax compatible object oriented programming language
PHP	Hypertext Preprocessor
CPU	Central Processing Unit
RAM	Random access memory
HDD	Hard disk drive
XML	Extensible Markup Language
JSON	JavaScript Object Notation
NPM	Node package manager
GraphQL	Open source data query and manipulation language
CTFM	Comprehensive testing framework for microservices

CHAPTER 1 - INTRODUCTION

This chapter covers the background of the research study, explains the problem statement, the motivation for the research and the research objectives.

1.1 Background

Microservice architecture has captured the attention of software architects all over the world with its popularity. Over the years, it has been accommodated and has evolved in many ways in the industry by migrating from a monolithic architecture to more focused smaller services [1]. This transformation has many benefits over monolithic architecture. These services can be rapidly developed, scaled, maintained, and continuously deployed independently and parallelly [9], Though this could be a simpler process at the initiation with fewer microservices over the time when the services become mature and the system becomes more fine-grained as the number of services increases along with the introduction of new services to cater the evolving business requirements, which leads to many complexities in this maturing architecture such as higher memory consumption, operational complexity, managing integrations, mechanisms for handling partial failures, cyclic dependencies, testing complexities, such issues are not easy to trace or mitigate in advance [2].

Microservices testing is a crucial process that ensures the correct orchestration and the expected functionality of the services. Currently available testing approaches and tools are cumbersome it’s not easy to initialize and keep the testing tools up to date with continuously changing unstable services and requirements, when the number of services is greater this is nearly impossible, developers face challenges especially when it comes to, integration testing service in isolation, it’s not feasible to set up all the services and seed the necessary data for testing in a local or remote resource-intensive development environment and some services might not be available at the project initiation. To mitigate or overcome these problems requires the refining or reshaping of the testing strategies [6].

1.2 Monolithic Architecture

A monolithic application is designed and built as a single-tiered software application usually, system user interface and data access layers which are combined into a single platform program [4]. Enterprise system applications are built into three parts, a database layer, a client-side User interface, and a server-side application. The server-side application handles HTTP requests from clients, which may also contain an API for third-party communications, and the application also might integrate with other applications via web services or message brokers. The server-side application executes defined business logic, which results in database operations and populates HTML views or responses and sent to requesting client. If there are any alterations made to the system developers must build, test and deploy the updated version of the application, but the testing process is less complex compared with microservices.

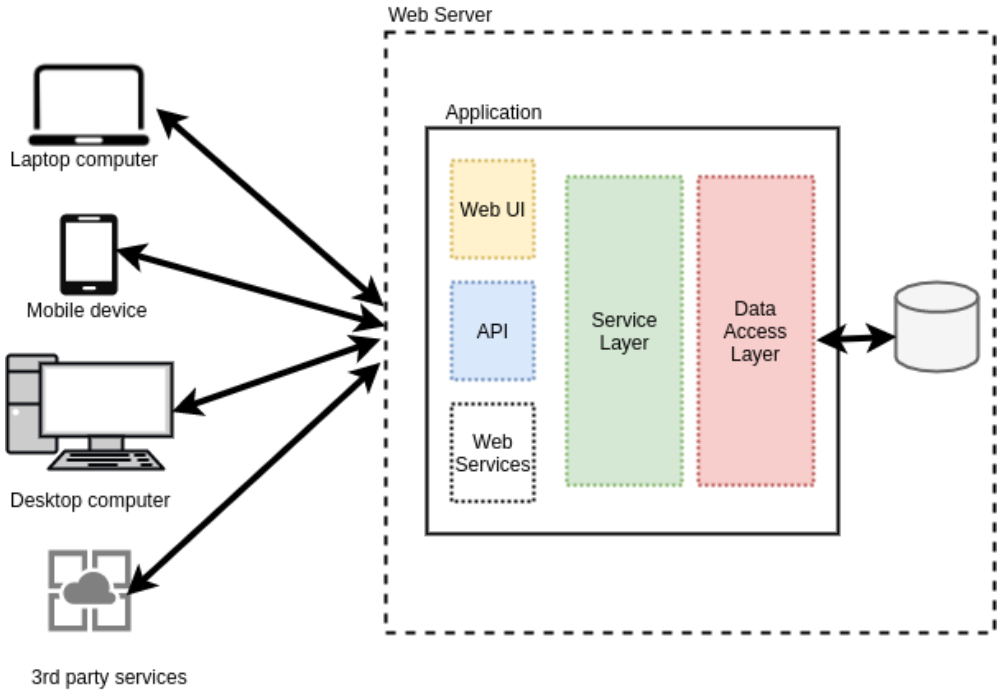


Figure 1.1: Sample monolithic application layout

1.3 Microservices Architecture

When applying the single responsibility principle at an architectural level over a monolithic application, identified components could be decomposed, and transformed into cohesive stand-alone microservices [1]. The motivation for this approach provides many benefits over a monolith, smaller more focused fine-grained independent services, which could be independently and parallelly developed, continuous delivery of new features and updates within a minimum lead time [8][3]. Microservices also encourages services to be developed using different scripting or programming languages and platforms [2]. Scalability and architectural level flexibility is higher than a monolith system.

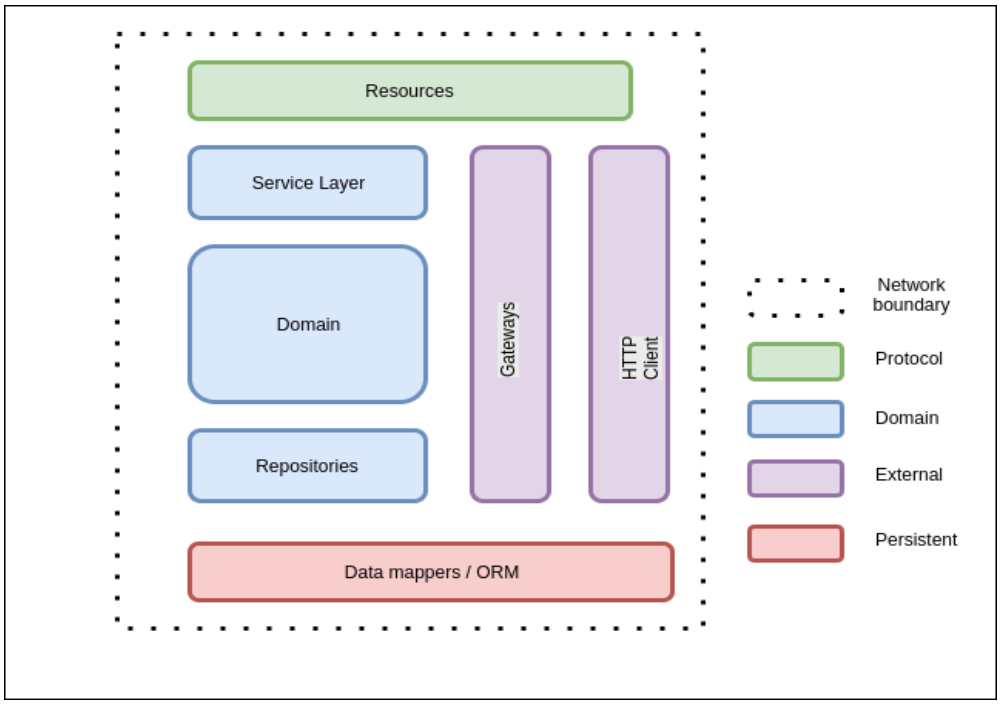


Figure 1.2: Sample microservice internal structure

The above figure displays a sample of a frequent microservices internal structure and its arrangements of protocol, domain, external and persistent layers. When designing and creating tests, which must be lightweight and should cover each layer and the layers in between.

Resources - This is a mapper in between for exposed application protocols by the service and the messages to domain objects. Typically these are thin and responsible for request validation and responses against the based on the manipulations of the business transaction [6].

Domain - Is wrapped around the services, repositories, and domain layers. Most of the service logic was implemented in this domain model which represents the business domain. Such services harmonize over multiple domain activities, repositories processes on, as a collection of multiple domain entities and which is the persistent layer [6].

External - This layer defines the logic which requires communicating with other collaborative external services. Messages are enclosed by the gateway when communicating with other external services. Often, the request-response cycle is handled by a client who is capable of grasping the hidden protocol. When there is an aggregator service across the other services resources, such a service must be capable of persisting the objects from the domain, between the requests. This scenario can be handled by data mappers depending on the persistent requirement complexity, repositories use a dedicated set of domain objects, which carries the logic to archive this objective [6].

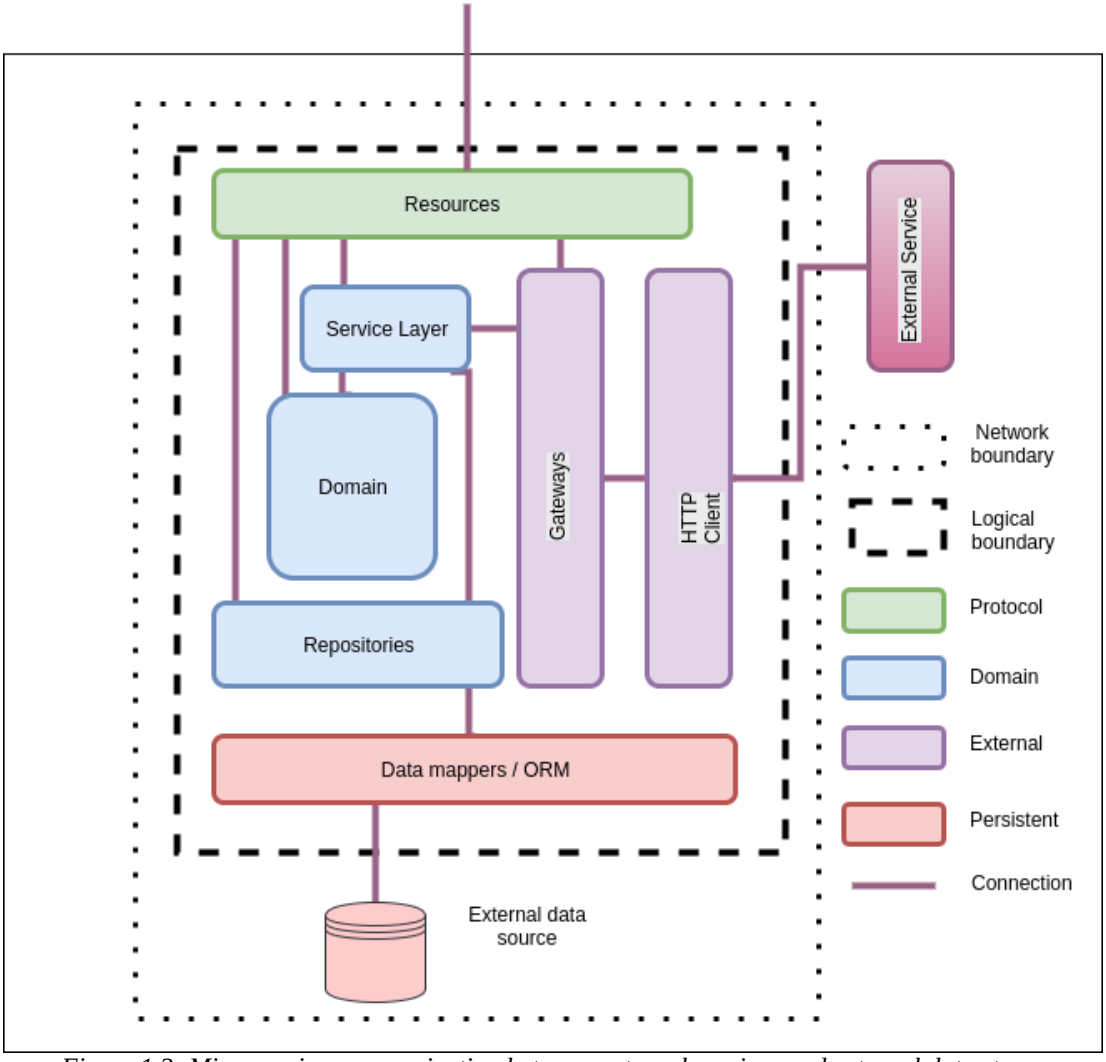


Figure 1.3: Microservice communication between external services and external data stores

The above figure displays how a request is handled and how a response is formed by passing messages to each relevant module. Some requests might require communication with repositories, gateways, and other services. To achieve this purpose module connection should be loosely connected. Lightweight focussed tests should cover these communications and should provide quicker feedback.

The request is intercepted by the resource and checks for validations, valid requests are processed in the domain. Also, a resource may authorize a service, to collaborate with many modules to complete a business transaction, otherwise, it may directly communicate with the particular module. The system should also be able to recover

from any communication breakdown from external services, which may be due to unavailability of a remote module or components, responsibility to handle such errors are passed on to gateways where the error handling logic is implemented. To reduce the unnecessary API requests and latency more, coarse-grained communications are conducted against the external services. The external data store is more logically coupled to a microservice than an external service. There is a risk that still between these connections that network latency could occur because the data store resides over a network boundary [6].

These additional network partitions directly impact the testing strategy. These tests may tend to fail due to many reasons like latency, unavailability of service, etc. Usually, these test execution times are longer.

1.4 Collection of services work together as a system

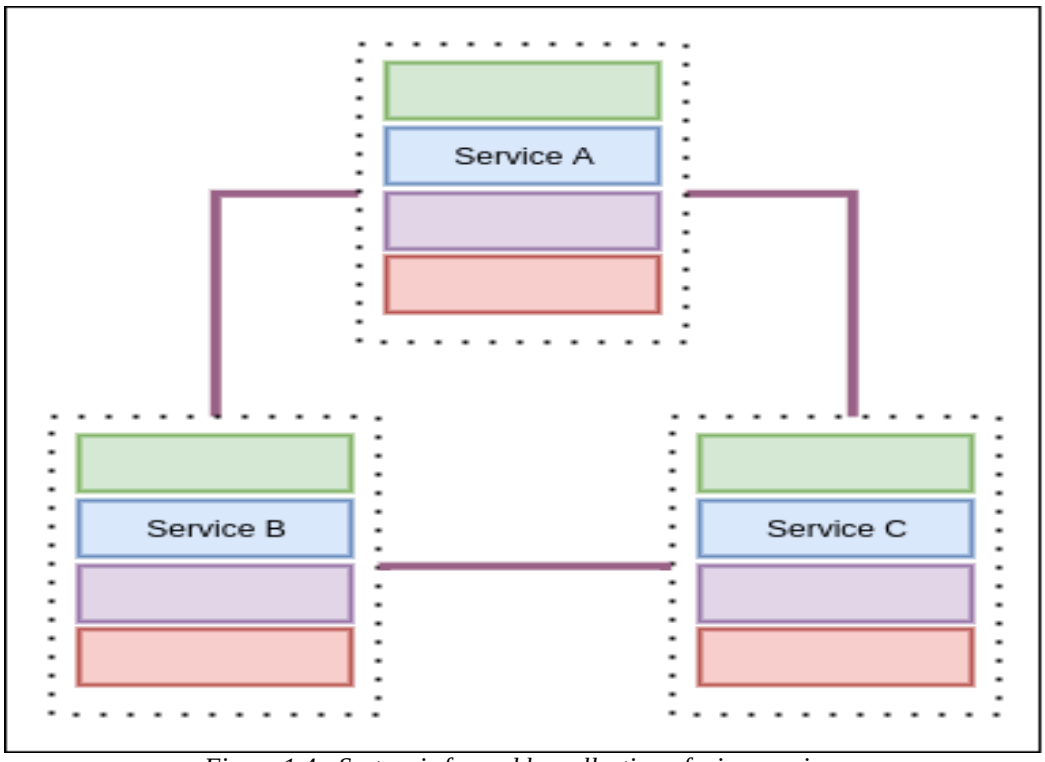


Figure 1.4: System is formed by collection of microservices

The above figure displays how the microservices work as a collection to form a whole system. A larger business request is collaboratively processed by relevant multiple microservices. Interchanging messages could be formed by using JSON, XML, etc. Usually, a larger business request, might spread across multiple services which might tend to fail, but using circuit breakers, system timeouts, and bulkheads this can be handled [8].

Another testing concern with external service testing is the availability and the changes of the services, which the consumer service development team might not be aware of.

1.5 Problem Statement

The main research problem focused on this research is to overcome the integration testing challenges of the complex microservices architecture. Integration tests are a must in test automation for the microservices and it carries out a major responsibility which ensures the communication paths are correct while interacting with other services. Because of this unique architecture testing integrations in isolation impose a challenge due to many reasons such as each microservice is developed and maintained by individual autonomous teams, unavailability or instability of service due to rapid parallel continuous development, deployment, with challenges and limitations when testing in a local environment [16][17][19].

The most commonly used techniques or approaches for integration testing are.

- Shared remote testing environment – Where a developer or tester can point to the remote services. This approach is costly, required real stable services to be available, and takes a considerable amount of time for feedback, higher accuracy needs effort in maintaining the environment, this approach is not feasible in the early development stages.
- Test kits – Much closer to the real services but scaled-down versions based on the development requirement, mostly physical memory intensive to improve faster feedback, frequent deployments, and build dependencies on implementation, needs attention and effort in creating, initializing, and maintaining. Accuracy is closer to the respective real services.
- Test doubles – Dummy or fake objects, Mocks, Spies, stubs fall under this category, These are configured or written in a predetermined way to behave as the real implementation, Needs attention and effort in maintaining, accuracy depends on the configuration of the respective technique [6][16].
- Service virtualization – This technique emulates a specific dependant component or a service that belongs to a component-based application. The amount of attention and the maintaining complexity, effort, and accuracy depends on the adapting tools and approaches [19].

As pointed out above the pros and cons of these techniques and the amount of time and effort varies. Which requires to initialize or keeping the environments up to date or maintaining according to the changing requirements and continuous development and deployments. This research study will focus on producing a service virtualization testing framework capable of automatically emulate responses to mitigate the testing challenges of the microservices.

1.6 **Motivation**

As pointed out in section 1.5, when it comes to testing microservices, doesn't bring out a pleasant experience or a smooth flow to both developers and testers, at any stage of the system development [25]. This could directly lead to inefficiency and increase costs of the development and infrastructure. This is mainly because of the weaknesses of the currently available testing tools, which are unable to mitigate testing challenges. Developers spend a considerable amount of time and effort facilitating the integration tests with the test kits, test doubles, or with currently available service virtualization tools, all of these require continuous maintenance and effort against the rapidly evolving services and requirements. To overcome these problems or challenges [6][27] requires a tool that requires very little effort in initiation, maintenance, cost-effective and efficient, should evolve with the services or adapt the growth of the services without any major effort, modifications or input and should be capable of automatically emulate the expected responses and provide faster feedbacks for the consuming services which the developers or testers can virtualize incomplete, unavailable or unstable microservices on the local or remote development environment, regardless of the development stage of the producing or consuming microservices.

1.7 Objectives

The main objective of this research is to mitigate the above-mentioned testing challenges in testing REST/RESTful microservices by creating a comprehensive integration testing framework. To fulfil the objective following requirements should be satisfied.

- Gather knowledge on the currently available techniques, tools, and solutions and the weaknesses.
- Study and analyze related literature for the research problem and possible solutions.
- Decide and finalize the potential methodology and suitable technologies to solve the research problem.
- Develop a comprehensive integration testing framework as proposed in the solution.

CHAPTER 2 - LITERATURE REVIEW

This chapter presents the important topics and background studies related the microservices testing and challenges.

2.1 Microservices testing

Monolithic architecture [4] is difficult to scale in distinct axes, less architecturally flexible, strict to one main programming language for all the components, must be deployed as a whole application even it is a minor change, because of these limitations the software industry shifted towards microservices architecture [8] which overcome these limitations and provides many benefits over the monolithic application. A monolithic application is broken into more focused fine-grained smaller microservices [1] which are developed, scaled, maintained, and continuously delivered and deployed independently by autonomous teams [9].

Adapting microservices introduces challenges, microservices are heavily based on remote dependencies over the network boundaries and unlike monoliths very less dependent on internally available components. These changes must be adopted by the testing strategies and the testing environment. More efforts are put into the orchestration of the systems and to test contracts and the correct communication between microservices. Testing strategies of monolithic application architecture cannot be directly applied to microservices architecture and testing strategies must be refined and reshaped to provide good overall test coverage on the microservices which facilitates continuous delivery and effective testing [9]. To fulfil the testing requirements of this microservices architecture, testing layers can be divided into the following [6].

2.1.1 Unit tests

Unit tests are the most focussed direct tests which test a smaller part of the application, it determines whether the test section behaves as expected, and these tests are mostly written at the class level or around several related classes. Unit tests can be divided further into two categories based on isolation or non-isolation from its collaborators [11].

2.1.2 Sociable unit tests

These types of tests test the behaviour of the modules by monitoring the state changes in the modules. Tests are carried out as black-box testing.

2.1.3 Solitary unit tests

These types of tests test an object for its interaction and collaboration and its dependencies, which are most likely to be replaced by mock objects.
Unit tests are carried out by isolating core system modules, these tests don't cover the moving parts of the system, which works together to form a complete service or any remote dependency interactions. To ensure the module interactions with the remote dependencies more coarse-grained testing is required.

2.1.4 Integration tests

These types of tests test collaborative modules as a subsystem to verify whether the collaborating services, communication paths are working correctly while interacting with other services. It mostly verifies the interactions between the integration layers of the services and the integrating external components. Automated tests are written to verify that modules' interaction with an external component is sufficient.

2.1.5 Gateway integration tests

These types of tests, verify any protocol-related errors, found errors will be verified at the finest granularity possible.

2.1.6 Persistence integration tests

These types of tests, verify that the schema structure resides in the code, compatible with the one in the data store. Unit tests and integration tests can cover the logic implemented in separate modules which forms up microservices, but this does not ensure the behaviour of microservices collaboration to full fill a business requirement.

2.1.7 Component tests

These types of tests test a single microservice for its complete functionality. If any external service interactions are needed, those services are mocked. To reduce the complexity of the component behaviour, it's isolated from its peers. This isolation helps to maintain better control over the testing environment.

2.1.8 Contract tests

These types of tests, verify the agreed contracts are satisfied with other integrating services. A contract is formed based on the consumer component requirements, hence there’s a tendency that providers and consumers evolve, the provider must continue to satisfy the consumer contracts. Component behaviour is not tested under contract tests, it verifies whether the inputs and outputs are matched or contains the required attributes as agreed with the provider and also checks for the response latency and throughput.

2.1.9 End to end tests

These types of tests, verify as a whole system whether it satisfies the business requirements without considering the component architecture. To run these tests the whole system is spun up and black box tests are carried out via public interfaces for integrating services. For a microservice behaviour that involves many moving parts, these tests provide coverage for the additional gaps between the services, which assures message passing between services are valid and assures any other network infrastructure like load balancers, proxies and firewalls are configured correctly.

2.1.10 Test Pyramid

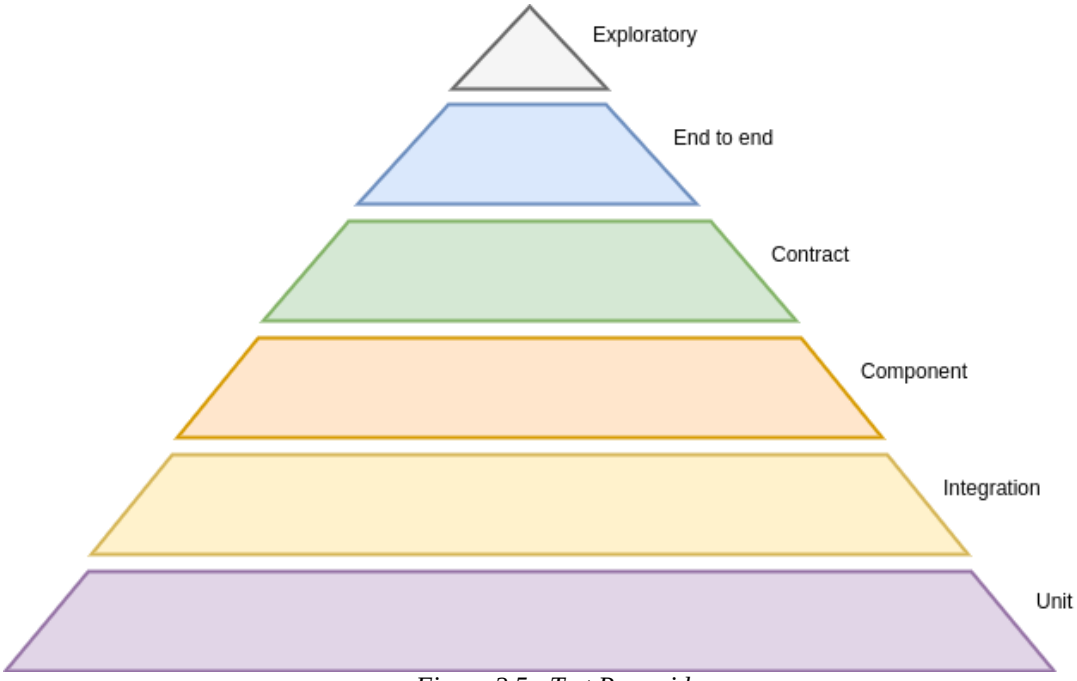


Figure 2.5: Test Pyramid

The above figure displays the testing pyramid which explains the characteristics of the testing strategies. From top to bottom tests are more focused in scope and at each granularity number of tests to be written becomes higher, time to create, execute and feedback times are lower, and also easy maintainability. Exploratory tests are carried out by the development team manually in a way that test strategies are incapable to cover. Usually, these tests are very time-consuming and not direct, which could easily

end up with failure to find any exploratory errors. The test pyramid provides a good understanding of how the test coverage should be at each granularity level.

2.2 Microservices testing challenges

To continue testing a software application with functional, integration, and performance testing it should be at least in a nearly completed state. Testing microservices developed by distinct autonomous development teams, all the necessary services must be completed and then deployed to a testing environment then testing can be continued once all the systems are integrated [23][25][27]. This approach is nearly impossible to adapt in a rapidly changing development environment.

Unlike monoliths, microservices cannot function independently, to complete a business request which might spread across several microservices and in the process exchange messages in between and to complete the request [22]. When it comes to testing a microservice in isolation the main challenge is the availability of the collaborating services, and also such services could be written in several different programming languages, spinning up all the necessary services in a local development environment is very difficult and time-consuming to set up and requires higher resource consumption and resource availability[3][28]. Following is a list of popular alternative techniques for testing microservices.

2.2.1 Shared testing environment

The most expensive and labour-intensive approach where the real services are served in a remote environment is shared among the developers and testers. This approach is only feasible when the services are in a stable and complete state. The accuracy is very high, feedbacks are considerably faster, requires heavy efforts in initializing and maintaining the environment [43].

2.2.2 Test kits

This approach is very similar to the shared environment but a scaled-down version of the real services, the scaling down varies based on the requirements how real the services should be based on that components or parts are removed or mocked within the services. Test kits are less expensive and less maintenance required than the shared environment but the resource consumption and the resource availability might be high compared to the shared environment because, this approach is highly memory intensive to provide faster feedback, accuracy is very closer to the shared environment [42].

2.2.3 Test doubles

For application software testing purposes, any type of replacement of the real implementation object could be called a test double. Such test doubles can be grouped as follows based on Their function, application, and behavior [16][17].

- Mocks: The expected behavior and functionality are preconfigured it'll throw an exception if the configuration is not satisfied.
- Stubs: Pre-programmed with expected behavior with is limited functionality,
- Spies: This is similar to stubs but it records information regarding behavior.
- Dummy: Sole purpose is to pass as a parameter in the called methods
- Fake: These are actual working implementations that are only used in the testing environment and not suitable for the production environment.

2.2.4 Service virtualization

Service virtualization is an approach used for testing component-based heterogeneous applications. A specific component or a service can be emulated via service virtualization, which allows making the unavailable dependant services available for the system under test [17] [19].

2.3 Currently available tools

This section describes the currently available popular tools and the limitations of those.

2.3.1 PACT

“PACT is a code first consumer-driven contract testing tool” [7] for the integration tests which can be used by the developers and testers to test HTTP services. With contract testing, the necessary parts of the requests that consumed by a consumer can be tested. PACT requires extra effort in writing contract tests and maintaining and adapting changes of the services this also including learning and writing PACT tests by the developers or testers [37][38].

2.3.2 Wire mock

Wire mock can be introduced as a mock server or a service virtualization tool. Requires efforts in understanding, configuring, and maintaining the Wire Mock server. Service changes need to be reflected on the mock server [20].

2.3.3 Hooverfly

Hooverfly claims to be the lightest API simulation tool, where the developers can use it as a proxy server or a web server, where real requests can be recorded and replayed or via configured simulations. Same as the wire mock when services changes and efforts need to be put in when updating the simulations and understanding, configuring, and maintaining the tool [21].

CHAPTER 3 - METHODOLOGY

This chapter describes the proposed solution based on the Open API specification and its structure, how the OAS adapted in the code base of a microservice, and the approach for the service virtualization.

3.1 Proposed testing framework

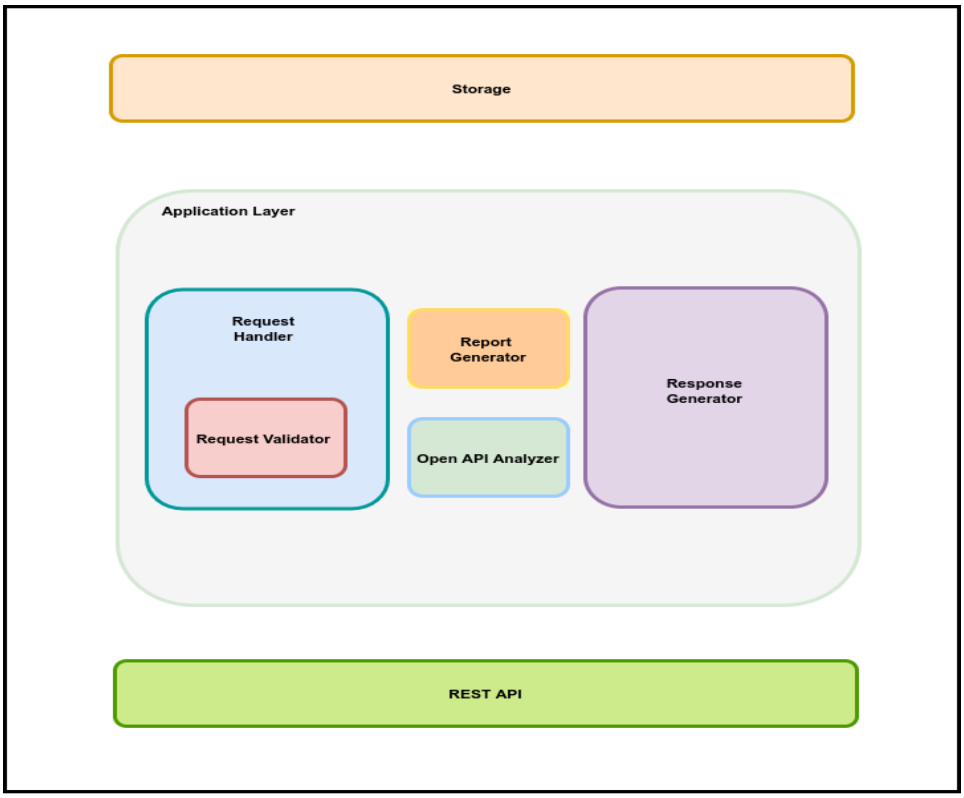


Figure 3.6The high-level structure of the proposed testing framework

The above diagram depicts the main components of the proposed testing framework. The main focus of this solution is to overcome the testing challenges in microservice [36] most effectively and efficiently, this will shine in where less effort in setting up the local or remote environment and less resource consumption and will hardly require any effort in maintaining and adapting the changes with the evolving microservices [45]. The platform-independent solution proposed under this topic will be

capable of responding to consumer requests by emulating responses of a producer based on its OpenAPI specification by virtualized producer [19][39].

3.2 Open API specification structure

The testing framework's main challenge is to identify the main components and the available services of the given producers' Open API specification schema to define and virtualize the producer services for the consumers. Below is an insight into the main segments of the Open API specifications.

Figure 3.7: Open API specification main structure

The above figure depicts the main structure of a OAS, which contains the version and the requests categorized by the paths and the HTTP verbs. Under those as the main structure as the figure depicts we can see the request specification.

3.2.1 Open API specification as code annotations

Almost all the programming or scripting languages supports Open API annotations in the code which is used to auto generate the specification for the services. With the annotation it helps to maintain and evolve the specification with the service changes adaptation.

```
/**
 * @OA\Post(
 * path="/card/{token}/activate",
 * summary="Activate card",
 * description="This will activate a card",
 * operationId="card-activation",
 * security={ {"authorization": {}} },
 * tags={"Card"},
 * @OA\Parameter(
 *    description="Card public token",
 *    in="path",
 *    name="token",
 *    required=true,
 *    example="Efs23234",
 *    @OA\Schema(
 *       type="string",
 *    )
 * ),
 * @OA\RequestBody(
 *    required=true,
 *    @OA\JsonContent(
 *       required={"pan_last_digits"},
 *       @OA\Property(property="pan_last_digits", type="integer"),
 *       @OA\Property(
 *            property="cardholder",
 *            type="object",
 *            @OA\Property(property="title", type="string"),
 *            @OA\Property(property="first_name", type="string"),
 *            @OA\Property(property="last_name", type="string"),
 *            @OA\Property(property="dob", type="date"),
 *            @OA\Property(property="address_line_1", type="string"),
 *            @OA\Property(property="address_line_2", type="string"),
 *            @OA\Property(property="address_line_3", type="string"),
 *            @OA\Property(property="post_code", type="string"),
 *            @OA\Property(property="city", type="string"),
 *            @OA\Property(property="country_code", type="string", example="BE"),
 *       ),
 *    ),
 * ),
```

Figure 3.8: Open API specification as code annotations

This figure depicts a sample adaptation of OAS in the codebase. Developers can specify the request properties and bound by using the annotations. This helps to maintain the OAS parallelly with the changes with services in the codebase. Usually, with the review cycles, developers verify the annotations are correct and up to date.

3.2.2 Paths

Paths section defines each REST endpoint supported by a specific service and segmented by the HTTP method.

```
{
    "openapi": "3.0.0",
    "info": {
        "title": "Issuing API",
        "version": "0.0.1"
    },
    "paths": {
        "/card/{token}/activate": {
            "post": {
                "tags": [
                    "Card"
                ],
                "summary": "Activate card",
                "description": "This will activate a card",
                "operationId": "card-activation",
                "parameters": [
                    {
                        "name": "token",
                        "in": "path",
                        "description": "Card public token",
                        "required": true,
                        "schema": {
                            "type": "string"
                        },
                        "example": "Efs23234"
                    }
```

Figure 3.9: Paths definitions of an Open API Specification

3.2.3 Parameters

Service can contain three types of parameters, header, path, and query parameters each contain its very own schema definition.

```
"/partner/{partnerId}/distributor/": {
    "post": {
        "tags": [
            "Distributor"
        ],
        "summary": "Add a distributor",
        "description": "This will add a new distributor",
        "operationId": "add-distributor",
        "parameters": [
            {
                "name": "Operation-id",
                "in": "header",
                "required": true,
                "schema": {
                    "type": "string",
                    "format": "uuid"
                }
            },
            {
                "name": "partnerId",
                "in": "path",
                "description": "Partner id",
                "required": true,
                "schema": {
                    "type": "integer",
                    "format": "int64"
                },
                "example": "5435"
            },
            {
                "name": "card_brand_ids",
                "in": "query",
                "description": "Card brand ids",
                "required": true,
                "schema": {
                    "type": "array",
                    "items": {
                        "type": "integer"
                    }
                }
            }
        ],
```

Figure 3.10: Parameters of a request

The above figure depicts a sample request parameter schema of the OAS specification request.

3.2.4 Request body

If a request contains a request body, this schema describes the media type and the body content and properties of an endpoint.

Figure 3.11: Request body schema

The above figure depicts a sample request body schema definition of a OAS request.

3.2.5 Response

The response object will contain the potential responses categorized with the HTTP status code and its specific schema object which describes how the response is formed.

Figure 3.12: Sample response segment

The above diagram depicts the potential response schema of a request which is categorized by its HTTP status code.

3.2.6 Testing isolated microservices using the framework

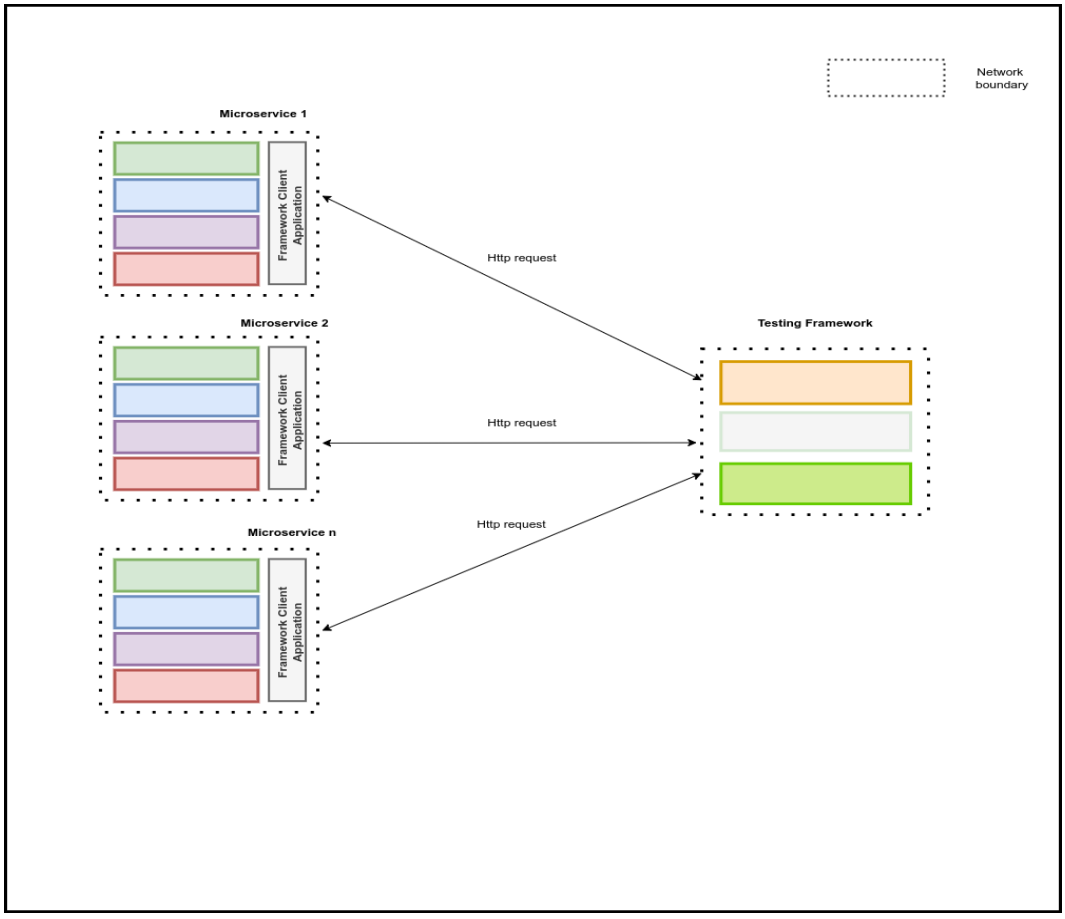


Figure 3.13: Testing microservices in isolation

The above diagram depicts consuming microservices are under test in isolation [47][48] by using the testing framework. The testing framework will be feeded with the generated OpenAPI specification of each microservice, these specifications are usually generated from already available services or prepared when decided on what services to be developed in the initial stages of the project development. The test framework client application will proxy the consumer microservice when the integration tests are called, the test framework should be running in the background ready to accept the requests at this point the following workflow will be executed.

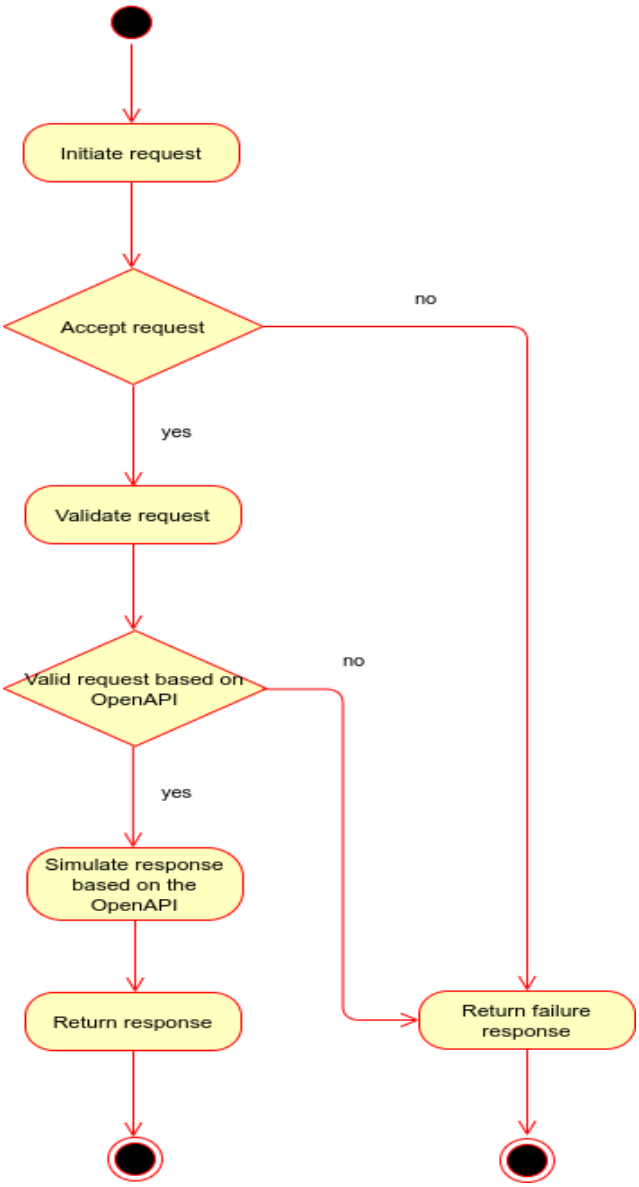


Figure 3.14: Request activity flow

The above diagram depicts the abstract service request activity flow of the CTFM. Initially, the CTFM will check if the services are defined in the specification if not it'll throw an error, else it'll proceed to the next stage and will check whether the request fulfils the OAS output is generated accordingly.

3.2.7 REST API

Incoming and outgoing HTTP request pairs will be handled by the REST API which is the exposed component or interface of the testing framework for the interaction of isolated microservices. For the initial stage, this API will support content-type “application/json” which is the commonly used content type for APIs.

3.2.8 Request handler

Request handler will accept the valid requests through the REST API which request are defined in the Producer Open API specification, such requests will be further validated by the request validator, which will check the request against the requesting producer services published specification whether it’s a supported request and for the defined parameters key, value data type schema. If the request is invalid it’ll be rejected with a detailed error response and it'll be logged for reporting purposes. The accepted requests will be passed on to the response generator for further processing.

3.2.9 Response generator

The response generator will accept the validated requests from the request handler. Which will initially check for any available recorded responses for the respective request, if available it'll be compared with the latest saved requesting response structure of the producing microservice and it'll align and merge the recorded response with the latest available version of the respective API, which will add or remove request elements such as attributes or objects based on its saved Open API Specification and accordingly the response will be prepared by the Open API Specification Analyser which will then initialize the request-response structure based on the written Open API Specification. Further, it'll log identified obsolete missing attributes of the request-response for reporting purposes. One of the other key features of this component is to generate a mock response based on the written Open API specification of the respective response if no recorded responses are found.

3.2.10 Report generator

This is one of the key components of the testing framework. Which will record all the necessary information starting from the REST API, request handler, response generator. The recorded information will be categorized under information, notice, warning, error, debug. This service will provide feedback on the ongoing process, which will help to monitor and take necessary actions easily by the developer.

CHAPTER 4 - DESIGN AND IMPLEMENTATION

This chapter describes the implementation scope, the system design and architecture including the technologies and concepts of the system. The latter part explains the implementation challenges of the edge cases and how to overcome such challenges, also presents the main user interfaces and its functionality of the framework.

4.1 The scope of the implementation

The main focus of this research is to overcome the integration testing challenges via a microservice virtualization solution which is cost effective, less time-consuming to set up and maintain with minimum configurations which will evolve with the micro-services [32]. The implementation will produce a proof of concept by following the proposed approach and solution under the Chapter-03 Methodology. The solution will satisfy the following scope of requirements as a proof of concept.

- Operating system agnostic and will support virtualization of REST/ RESTful API microservices developed adopting OpenAPI specification.
- Supports request body validations for the following request schema definitions for HTTP methods POST, PUT and PATCH.
    - required
    - minimum
    - maximum
    - maxLength
    - minLength
    - maxItems
    - minItems
    - uniqueItems
    - enum,
    - oneOf, AnyOf
    - pattern

- Emulate consistent request and responses within the defined request OpenAPI specification
- User-friendly responsive interfaces to register the producer microservices and to capture its OpenAPI specification should be able to use by any user without prior experience.
- Generates report for each test cycle, where the user can find in detail information of the test run.
- Only English is supported at this stage.
- Standard error response formats.

4.2 Test framework implementation

The testing framework should be capable to virtualize REST/RESTful based producer microservices [41] and interact with consumer microservice or microservices within or beyond the network boundaries [46], also should provide a User interface for the end-users to interact with the framework enabling them to monitor the service interactions in a local or remote environment.

The CTFM will cater the above mentioned main requirements with the adaptation of REST and MVC architectures, including the repository pattern. The proposed testing framework will be a web application, the reason being for this choice is, the framework should be capable of communicating with microservices over HTTP/HTTPS protocol on a remote or local network and also should facilitate an graphical user interface for the end user interactions.

The framework will be implemented using open source technologies, back-end based on the Laravel framework 8, as the web user interface the front-end library ReactJs and MongoDB for the persistent storage. The framework will expose a REST API to expose the virtualized services to consumer services. This framework application will be capable of self-hosting by itself or be able to support containerization with docker [44] [45]. Following will be the main architectural diagram of the CTFM followed by the dependencies of the front end and the back end packages, libraries and technologies, complete dependencies will be available under the appendix.

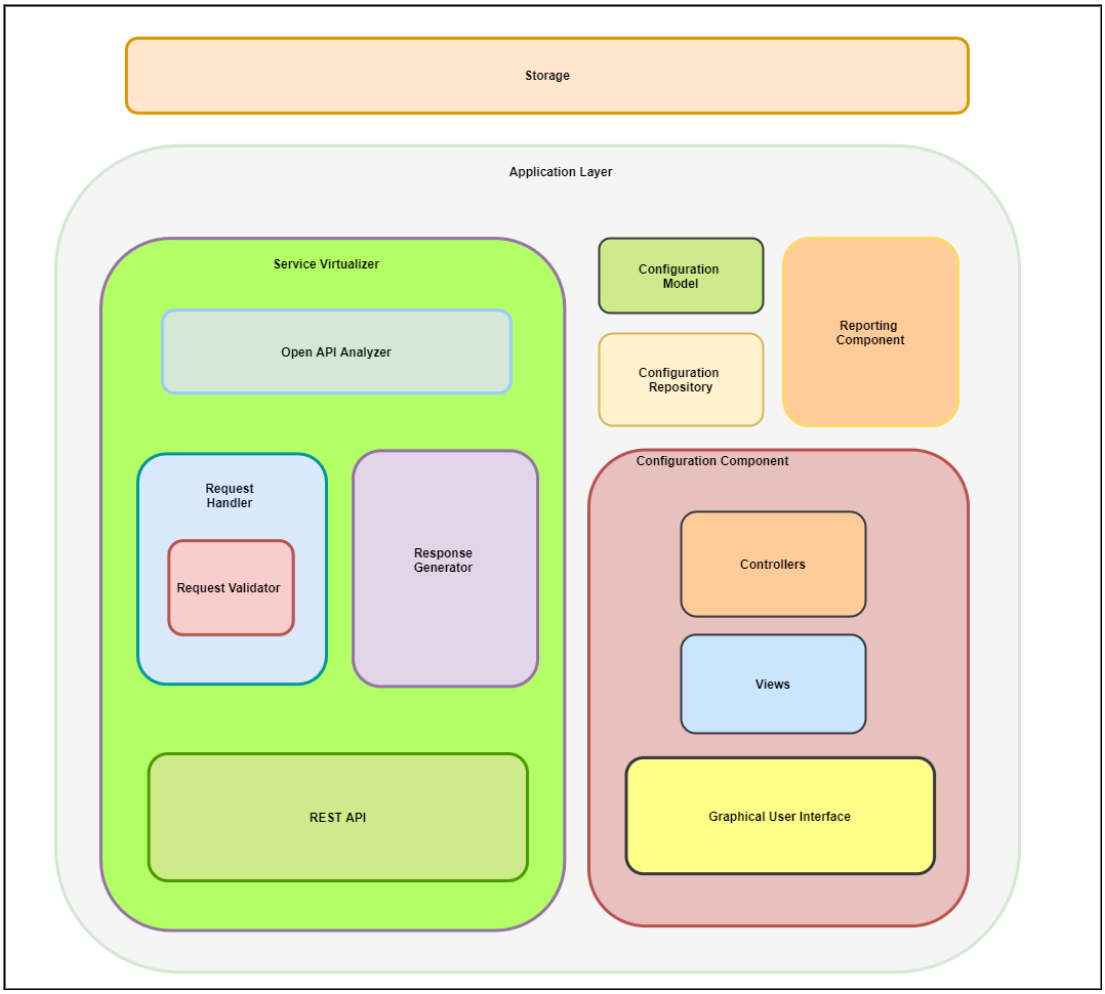


Figure 4.15: CTFM abstract architecture

The above diagram depicts the abstract architecture of the CTFM. The framework composed of a REST API interface which will expose the virtualized the registered producer services' OAS for the consumer's integration [49]. Web graphical user interface is facilitated for the end user to register producer services and its OAS, interfaces provided to view specific producer reporting and to view the OAS specification documentation. The CTFM main components and its functionality is explained in the latter parts of this chapter.

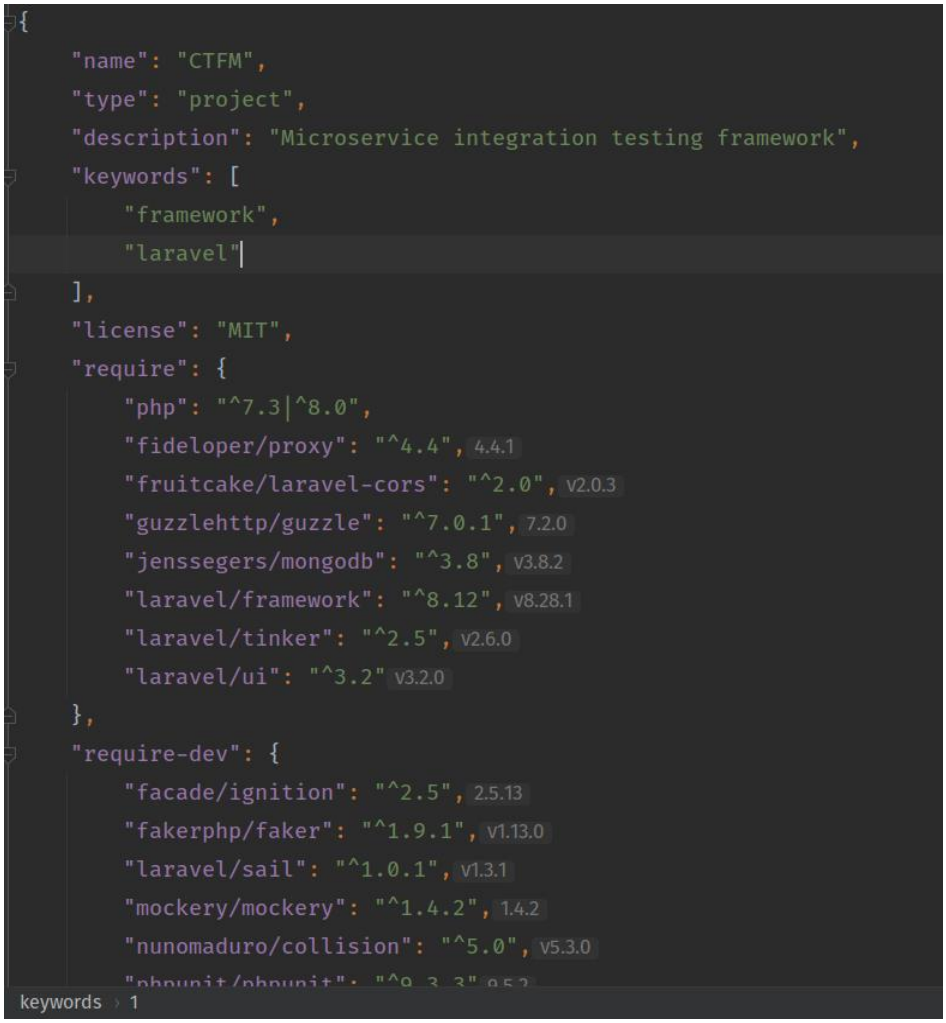


Figure 4.16: composer.json

The above diagram depicts the composer.json of the framework which manages the dependencies of packages, libraries and technologies for the backend.

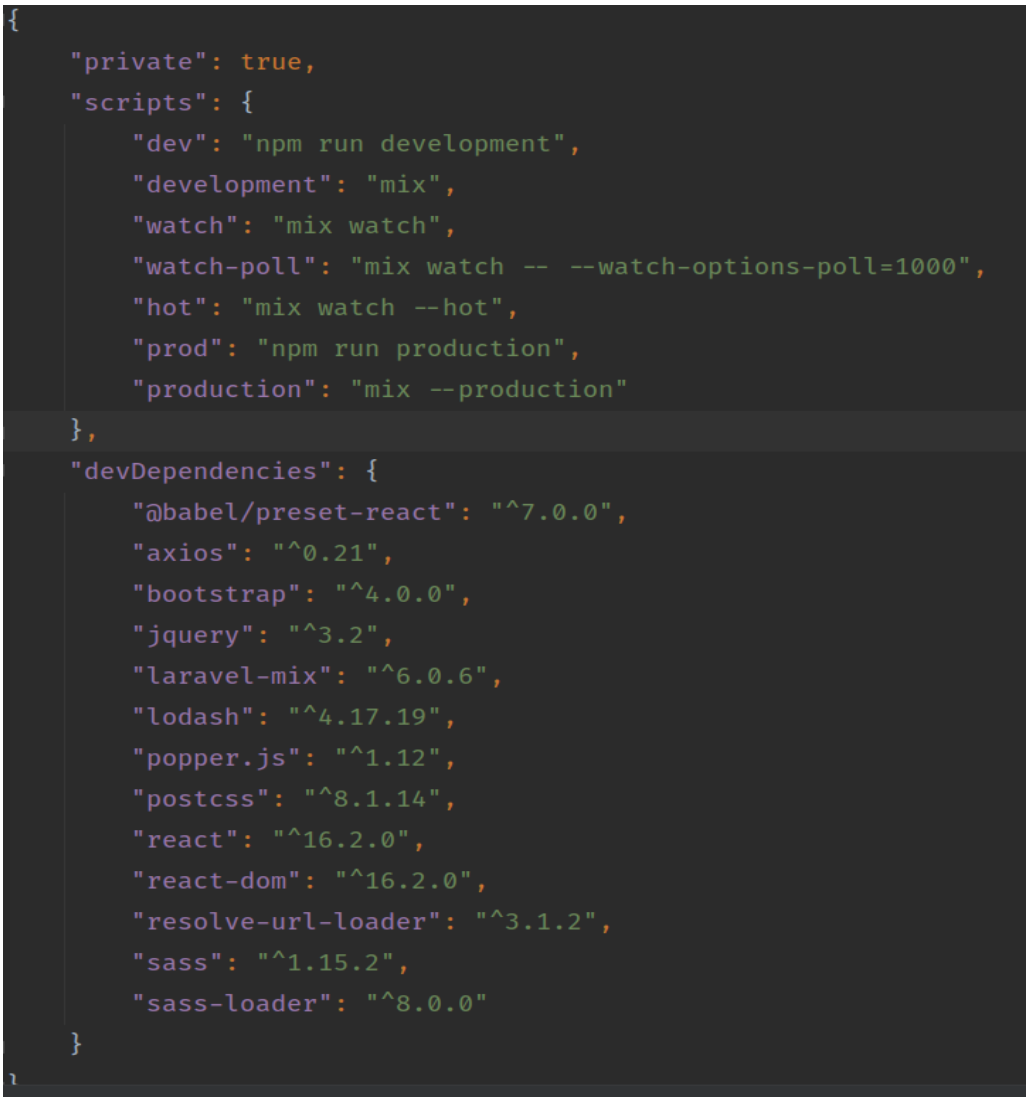


Figure 4.17: package.json

The above diagram depicts the package.json of the framework which manages the dependencies of packages, libraries and technologies for the frontend.

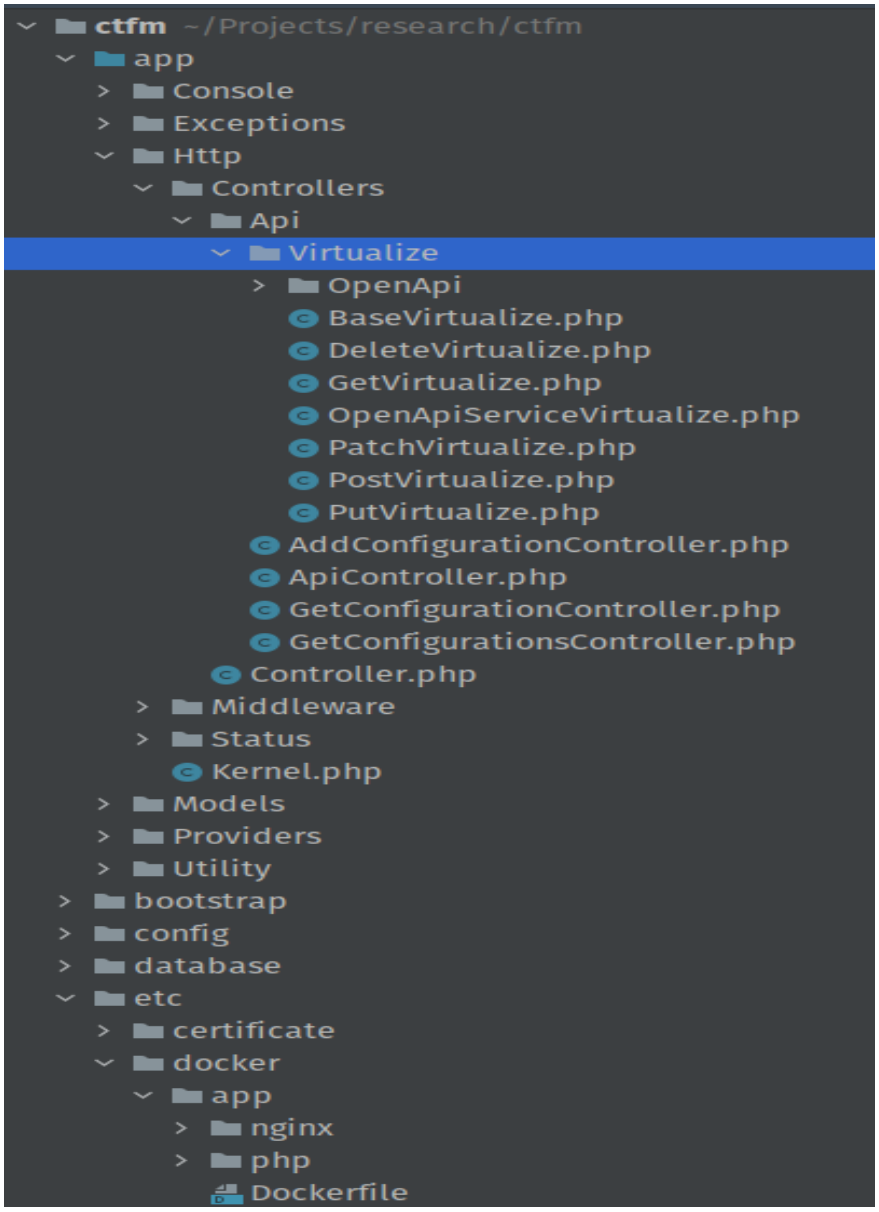


Figure 4.18: Framework folder structure

The above figure depicts the framework high level folder structure.

4.3 Producer registration and capture Open API Specification

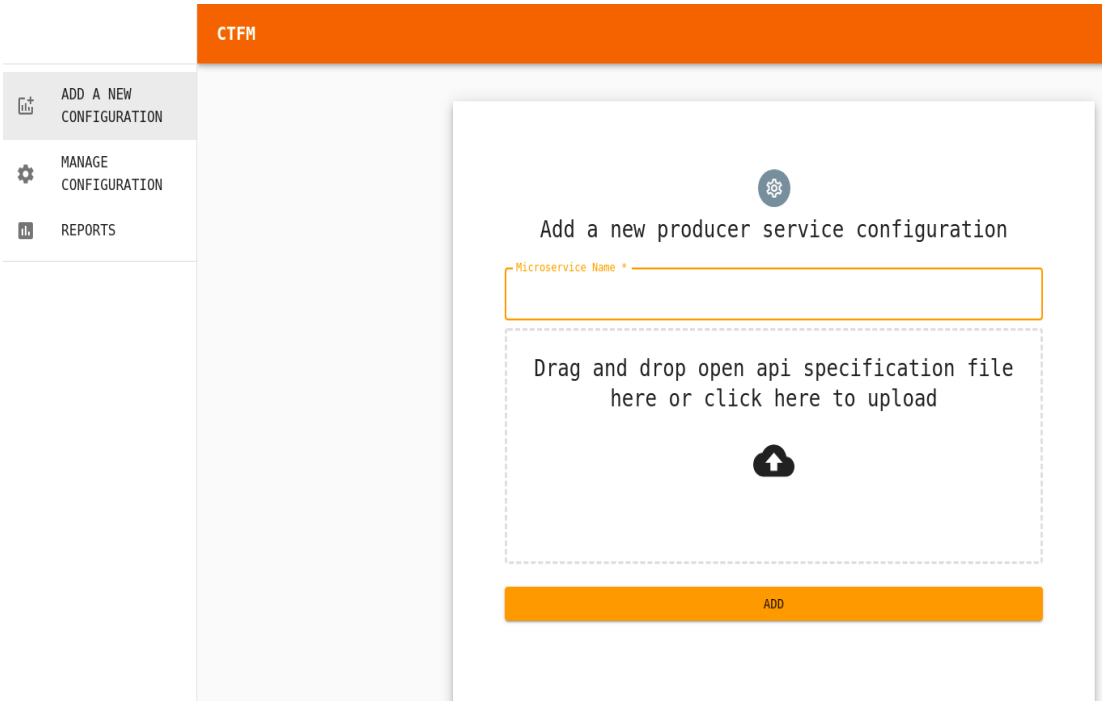
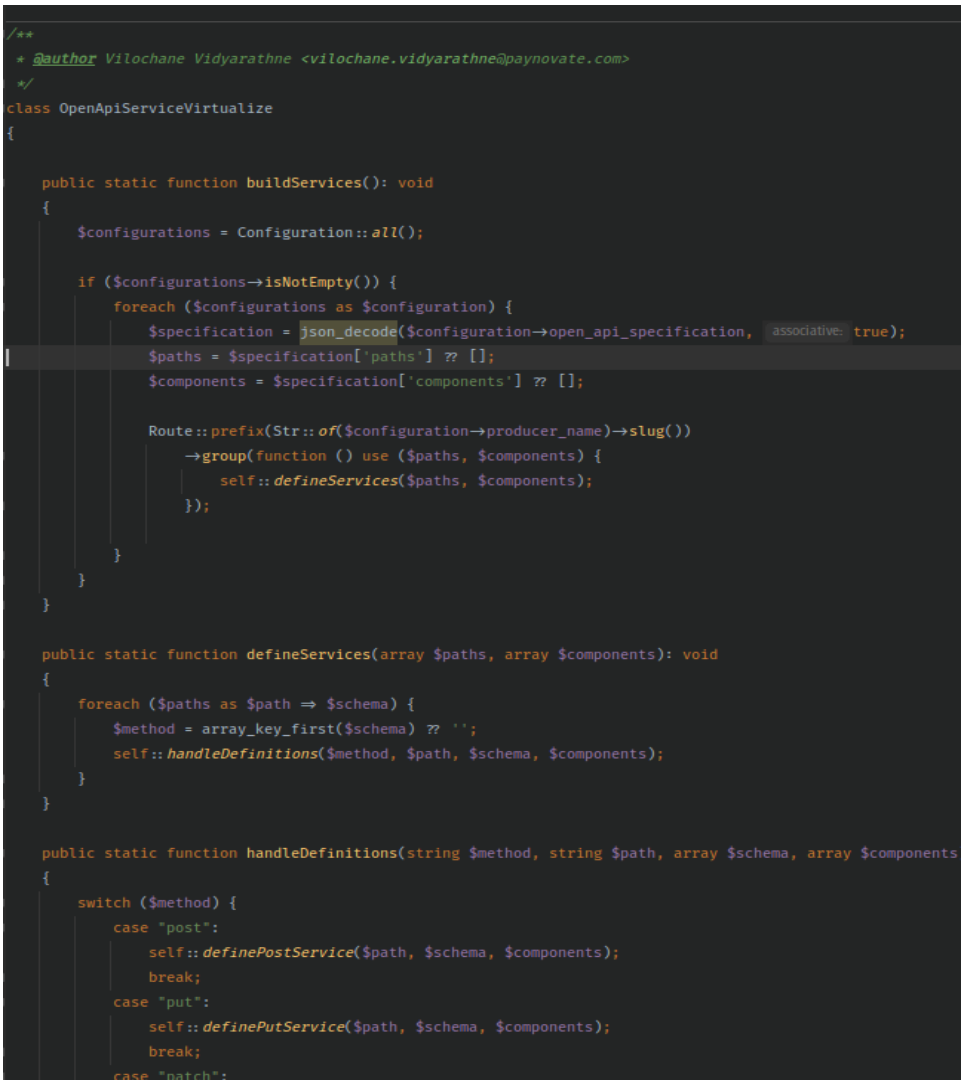


Figure 4.19: Adding a new producer configuration

Users can add new producer configurations via this form. It'll request the name of the producer which will be used to prefix the URL of the testing framework to avoid route conflicts and which will be used by the consumer services to interact and the OpenAPI specification of the respective producer. All the information will be saved in the MongoDB persistent storage.

4.4 Producer service virtualizer

This is one of the most critical components of the testing framework, which should load the stored configurations up on bootstrapping the testing framework and should prepare and define the services within the bounds of the Open API specification of each respective producing microservice. This component will segment each service into HTTP methods, GET/PUT/PATCH/POST/DELETE from the stored specification, and will analyze each method with the Open API analyzer and will decide how to request structure, responses, paths, and the validations should be handled.



```php
/**
 * @author Vilochane Vidyarathne <vilochane.vidyarathne@paynovate.com>
 */
class OpenApiServiceVirtualize
{

    public static function buildServices(): void
    {
        $configurations = Configuration::all();

        if ($configurations->isNotEmpty()) {
            foreach ($configurations as $configuration) {
                $specification = json_decode($configuration->open_api_specification, associative: true);
                $paths = $specification['paths'] ?? [];
                $components = $specification['components'] ?? [];

                Route::prefix(Str::of($configuration->producer_name)->slug())
                    ->group(function () use ($paths, $components) {
                        self::defineServices($paths, $components);
                    });

            }
        }
    }

    public static function defineServices(array $paths, array $components): void
    {
        foreach ($paths as $path => $schema) {
            $method = array_key_first($schema) ?? '';
            self::handleDefinitions($method, $path, $schema, $components);
        }
    }

    public static function handleDefinitions(string $method, string $path, array $schema, array $components
    {
        switch ($method) {
            case "post":
                self::definePostService($path, $schema, $components);
                break;
            case "put":
                self::definePutService($path, $schema, $components);
                break;
            case "patch":
```

Figure 4.20: Producer virtualize class

4.4.1 Request parameter validation

REST API Request could contain header parameters, path parameters, and query parameters. Below is an Open API request parameter schema.

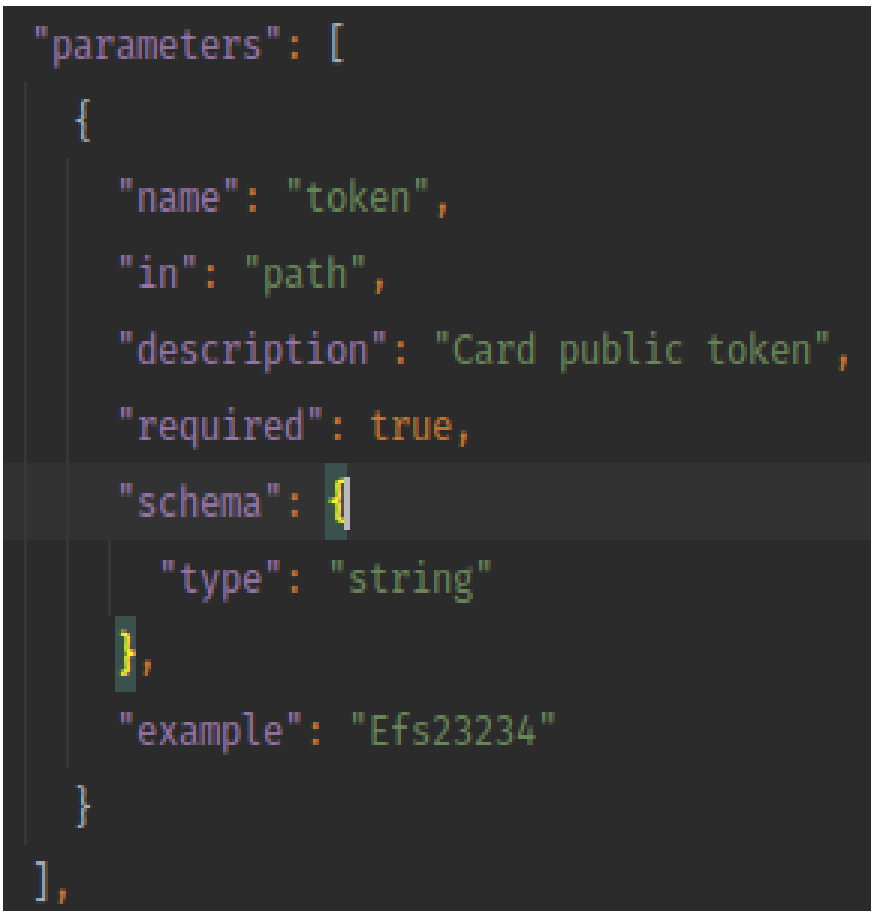


Figure 4.21: Open API parameter schema object

The above figure depicts the OAS parameter schema definition object.

The testing framework should validate all the incoming parameters to make sure the consumer sends the correct type of parameters. Below is the Schema analyzer class parameter validation segment which will validate the parameters against its schema object.

```php
public static function validateRequestParameters(
    string $path,
    array $requestParams,
    array $schema
): \Illuminate\Contracts\Validation\Validator {
    $rules = [];
    $parametersSchema = $schema['parameters'] ?? [];

    $mappedParameters = self::extractPathParameters($path, $requestParams);

    foreach ($parametersSchema as $parameterSchema) {
        $parameterName = $parameterSchema['name'];
        if (!isset($parameterName) || !isset($parameterSchema['schema'])) {
            continue;
        }

        $requiredParameters = [];
        if (isset($parameterSchema['required']) && $parameterSchema['required']) {
            $requiredParameters[$parameterName] = [$parameterName];
        }

        $rules[$parameterName] = self::buildValidations($parameterName, $requiredParameters,
            $parameterSchema['schema'])[$parameterName];
    }

    if (empty($rules)) {
        return Validator::make([], []);
    }

    return Validator::make($mappedParameters, $rules);
}
```

Figure 4.22: Schema analyzer request parameter validator

Below figure depicts request parameter validation based on the OAS parameter definition by the framework.

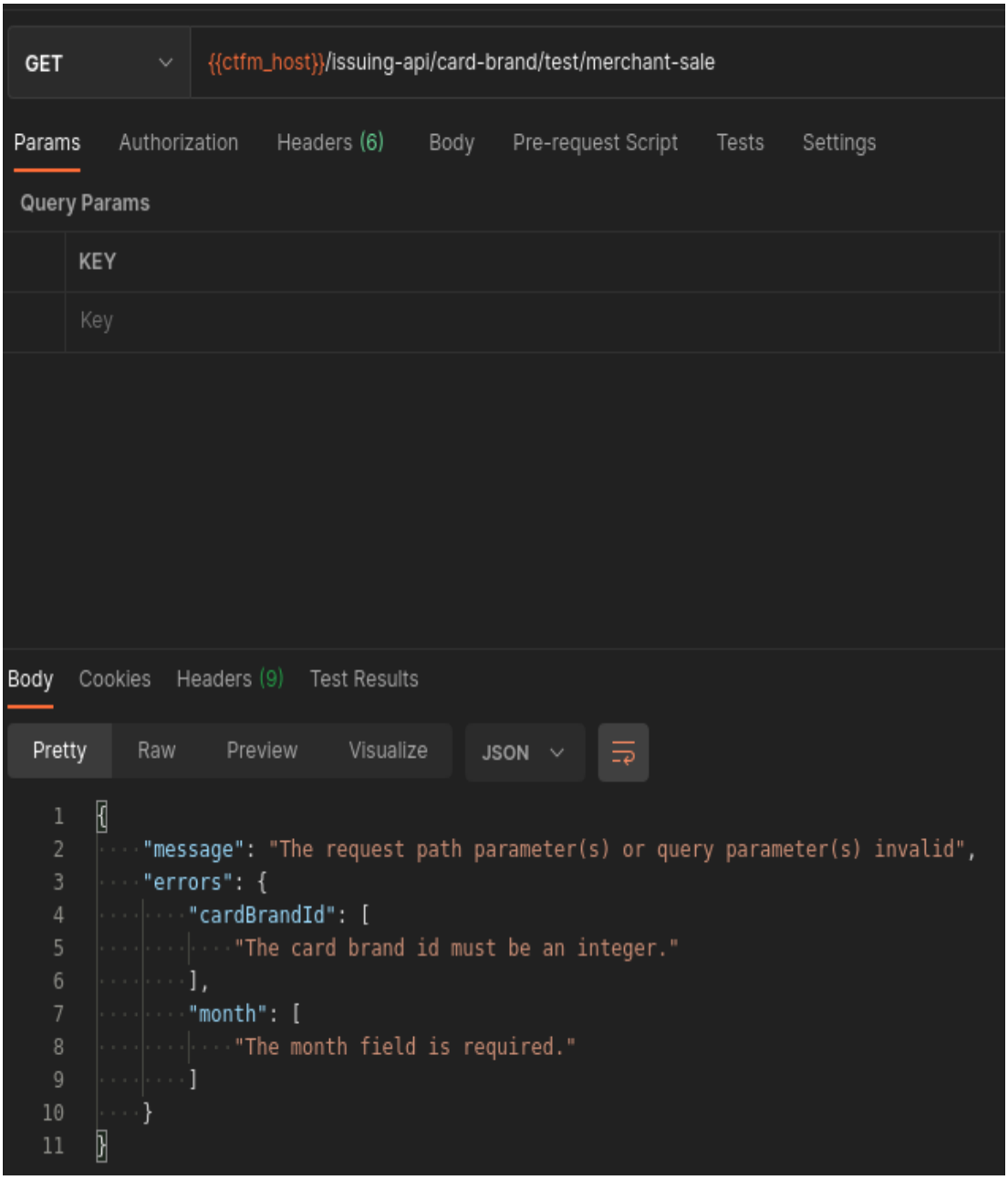


Figure 4.23: Request parameter validation error response

4.4.2 Request body validation

This is one of the complex components of the testing framework. If a request contains a request body, the test framework should validate the request payload with the defined schema of each respective payload property. There are several ways of schema definitions as shown below.

4.4.2.1 As a reference object

```
"requestBody": {
  "description": "Card order information",
  "required": true,
  "content": {
    "application/json": {
      "schema": {
        "required": [
          "reference",
          "quantity",
          "distributor_id",
          "delivery"
        ],
        "$ref": "#/components/schemas/CardOrder",
        "type": "object"
      }
    }
  }
```

Figure 4.24: Request body as a reference

4.4.2.2 As a object

Figure 4.25: Request body definition as a object

The request body can contain the following data types and complex types either as objects or arrays which also could contain primitive data types or complex types.

- integer
- number
  - int32 (signed)
  - int64 (signed)
- string
  - byte
  - binary
  - date
  - date-time
  - enum
  - email
  - uuid
- boolean
- object
- array

Below is a sample of a complex data structure of defined Open API schema, for a request payload the validator should build the validation based on the defined schema. It will initially take into account the required segment of the schema and then primitive data types and complex data types and each of its specified schema and formats. It should also take into account if any optional parameters are present it should validate the optional parameters accordingly if present.

Figure 4.26: Request complex payload structure definition

Below is a formation of the validation rules as per the Laravel framework validation rules definitions, for the above requests' Open API specification.

```
array:14 [
  "payments" => "required|array"
  "payments.*.method_id" => "required|integer"
  "payments.*.amount" => "required|numeric"
  "tokens" => "required|array"
  "tokens.*.token" => "integer"
  "tokens.*.amount" => "numeric"
  "fields" => "array|max:10|min:1"
  "fields.*.field_id" => "required"
  "fields.*.value" => "required"
  "additional_products" => "array"
  "additional_products.*.product_id" => "integer"
  "additional_products.*.quantity" => "integer"
  "channel" => "required|integer"
  "desk" => "required|integer"
]
```

Figure 4.27: Formation of validation rules for a given specification

4.4.2.3 Handling validations requires logic or database interactions

One of the limitations found out with the testing framework is unable to validate unique or existence and dependant validations which require database interactions or logic, to mitigate this problem, the framework will receive an instruction header that will instruct the framework to virtualize that respective error, for a specific test. Following are available validations. Complete guide available for these validations in Laravel documentation [40].

- unique
- exists
- required_if
- required_unless
- exclude_if
- exclude_unless

Below is an example of how to override the request validation when required by the user for a specific validation error. The header should be formed as a JSON and the properties same way as in Figure 4.12

Header	value
ctfm-validation	{"fields.*.field_id": "exists"}
content-type	application/json

Table 4.1 Header to override validations

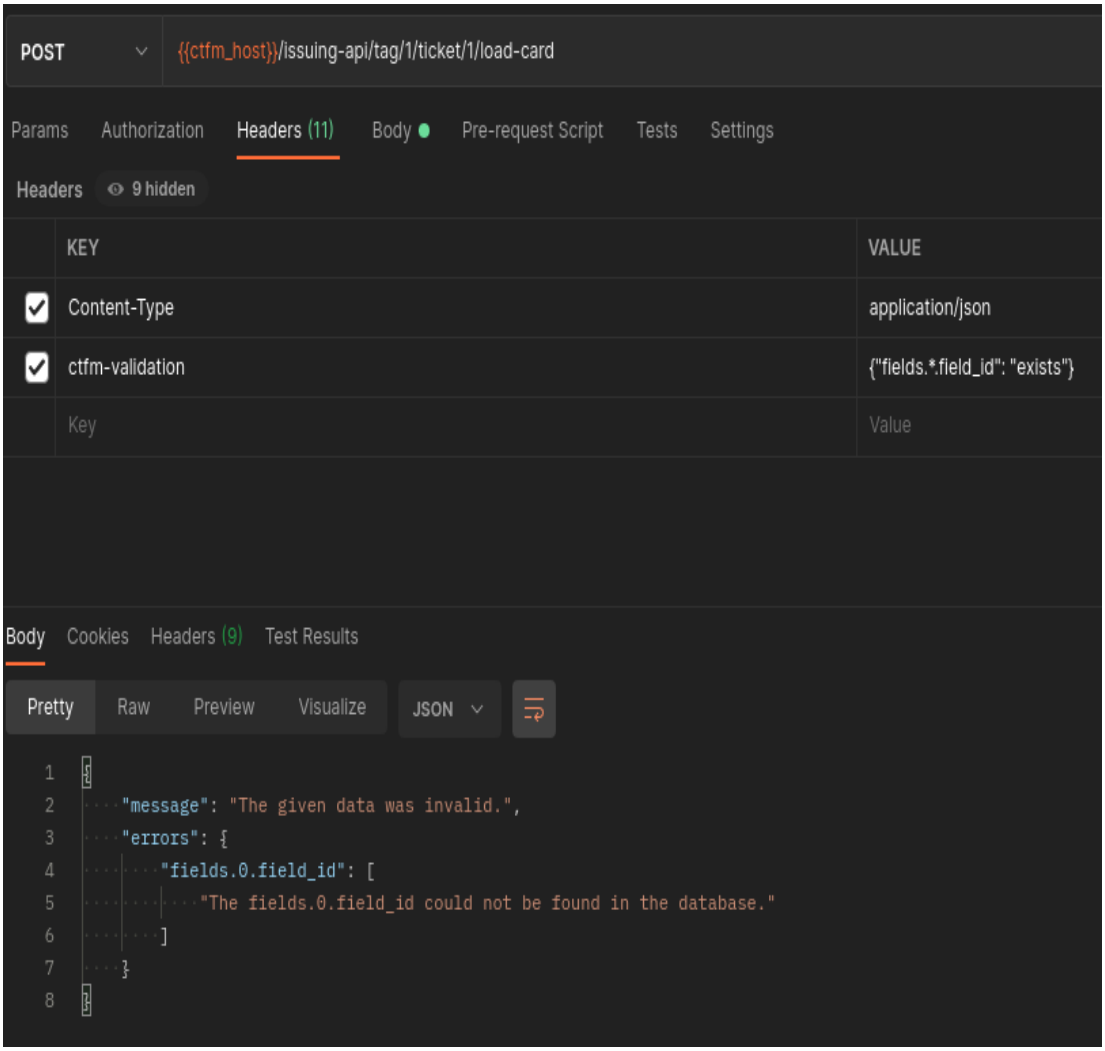


Figure 4.28: Request validation rule overridden example

The above diagram depicts an example of a request validation edge case which requires database interactions, since the CTFM doesn't have such this example shows how to produce such validation with CTFM.

4.5 Response generator

Open API responses are defined with the response object and it's mandatory to have at least one response for each respective request. The schema document will categorize the responses with the HTTP status codes. The below figure depicts the schema definitions for the request potential responses.

Figure 4.29: Open API response specification

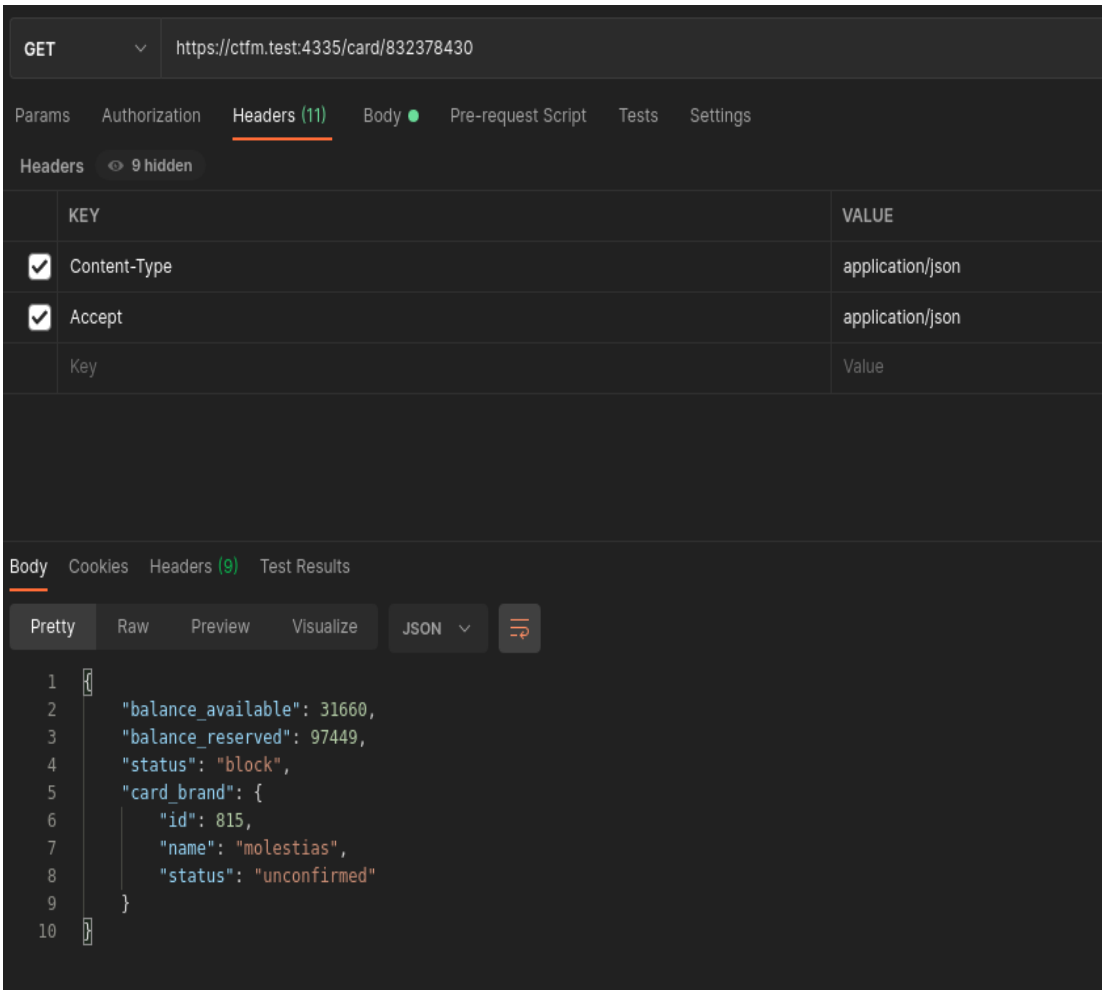


Figure 4.30: CTFM generated response according to response specification

By default, the first status is selected and its response is automatically generated by the framework according to the defined schema bounds, when there could be scenarios like edge cases or failures where the user would expect a specific response or a value or values apart from the auto generated response. In these situations, the user can override the expected value or values of a response by instructing the framework from the special instruction header formed as a JSON schema, if any values to be changed or if only just the status code like the below examples. If the expected responses were not found or any request-response values not found in the defined producer response schema specification framework would complain about that error and only one status could be requested for a specific request at a time.

Header	value
ctfm-response	401
content-type	application/json

Table 4.2: Expected response request headers with only status code

The below figure depicts how to instruct the CTFM to send a response with only the status code without overriding the response values.

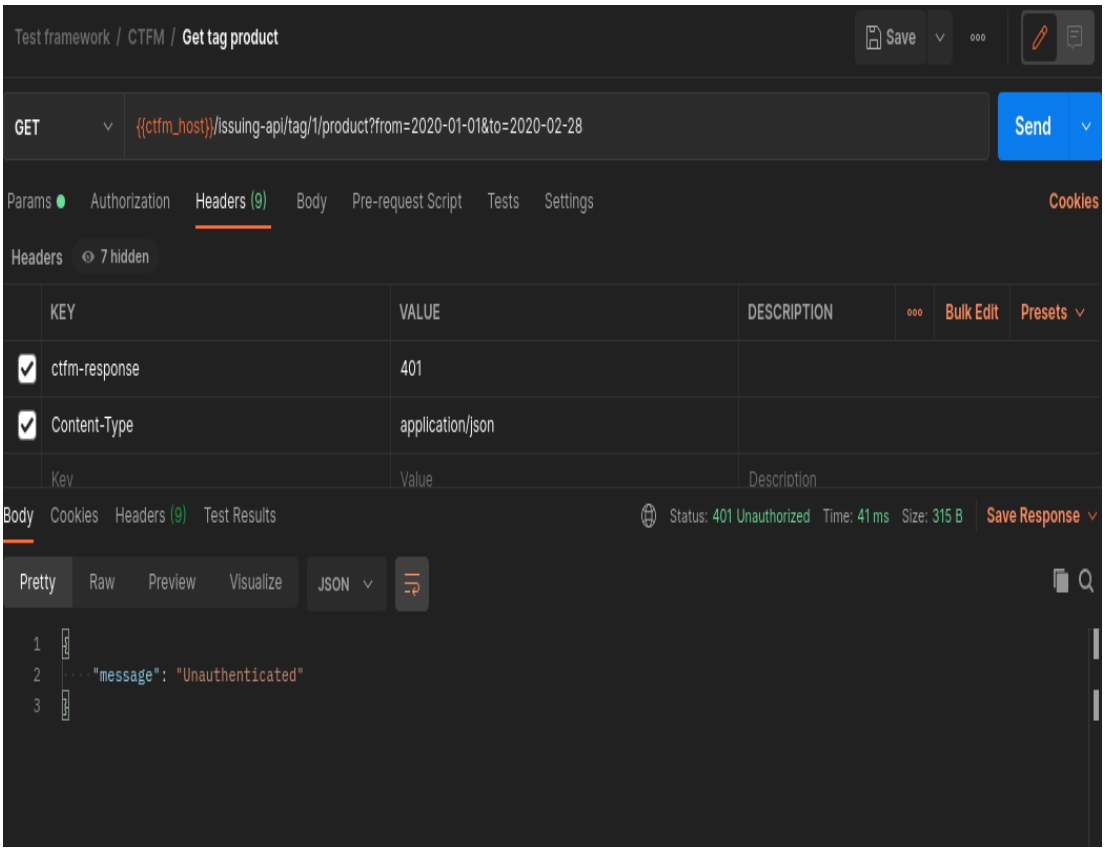


Figure 4.31: Expected error response only instructed with the status code

Header	value
ctfm-response	{"sta- tus":"401","schema":{"properties":{"message":{"type":"string","value":"T his is a modified expected message"}}}}
content-type	application/json

Table 4.3: Expected response request headers with json schema

The below figure depicts how to instruct the CTFM to send a response with the status code overriding the response values. Users can send the producer response schema of the respective response by the response property or properties adding the key named "value" and the value should be overridden for that response should be formed the in JSON instruction header as in Table 4.2.

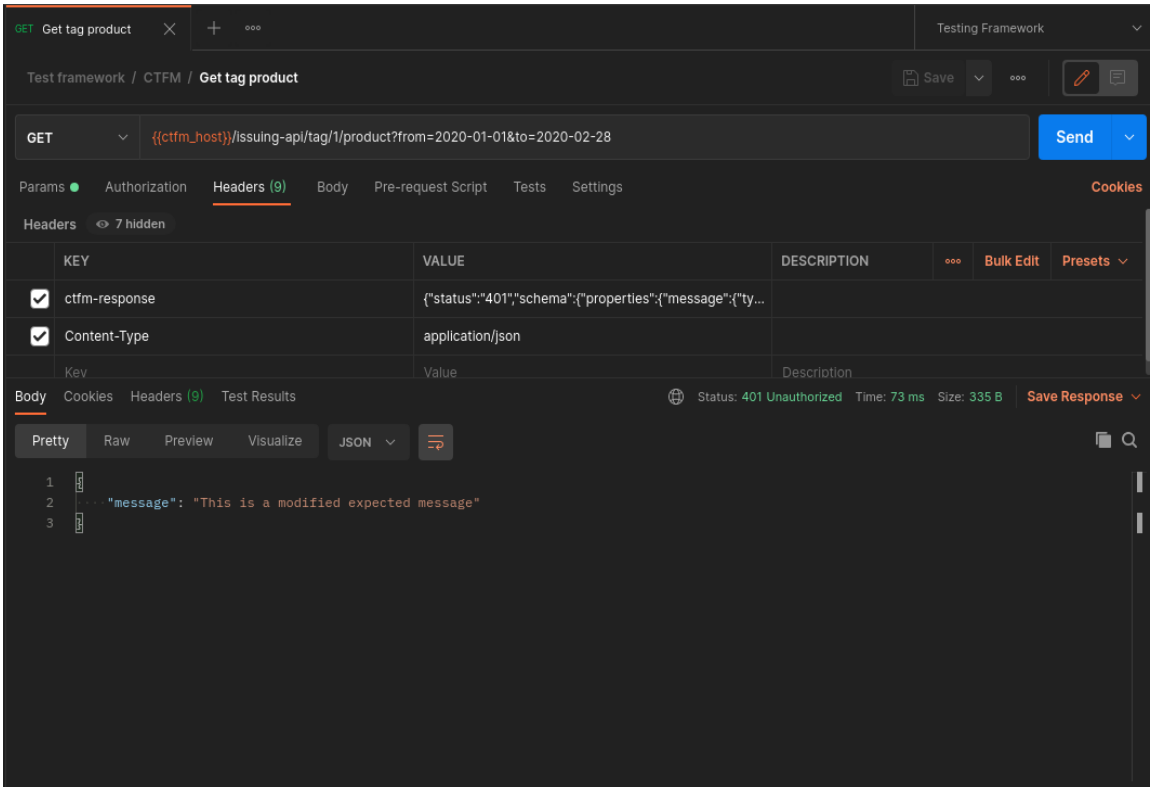


Figure 4.32 Expected response with the status and overridden response

4.6 View producer Open API specification

System will provide a link to view the Open API specification or a producer, for the user reference. Below shows how to access the configurations and the specification view.

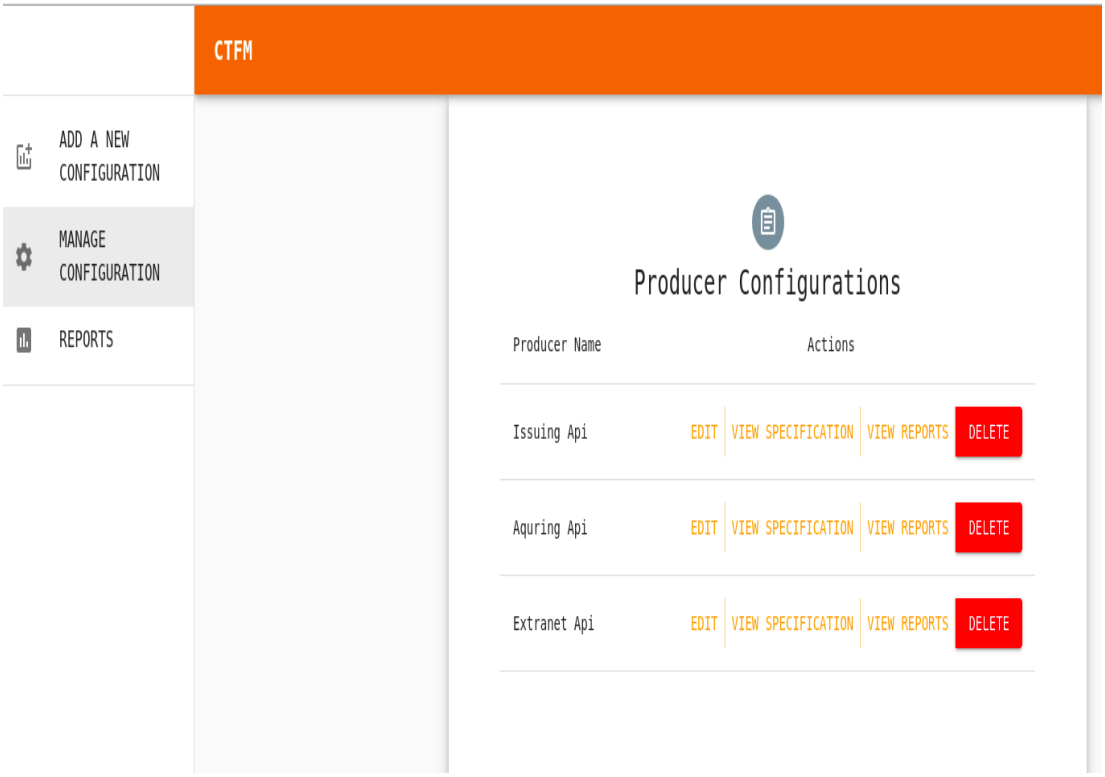


Figure 4.33: Configuration view to access the specification of a producer

This figure depicts the user interface for the mange producer configurations, which shows the configured producers and its available actions.

Below depicts the Open API specification documentation available for a registered producer with the CTFM.

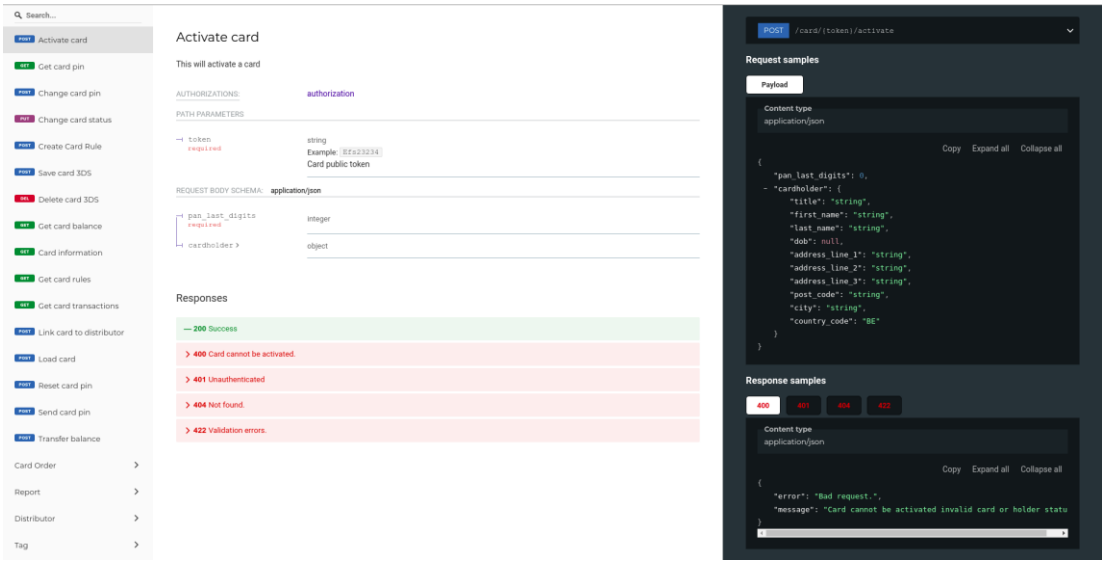


Figure 4.34: Open API specification view of a producer

4.7 **Reporting**

CTFM will provide reports of a specific producer for the latest ten test cycles. It'll categorize the errors under four main categories which are,

- Unavailable service
- Invalid service parameters
- Invalid request body schema
- Invalid request-response schema

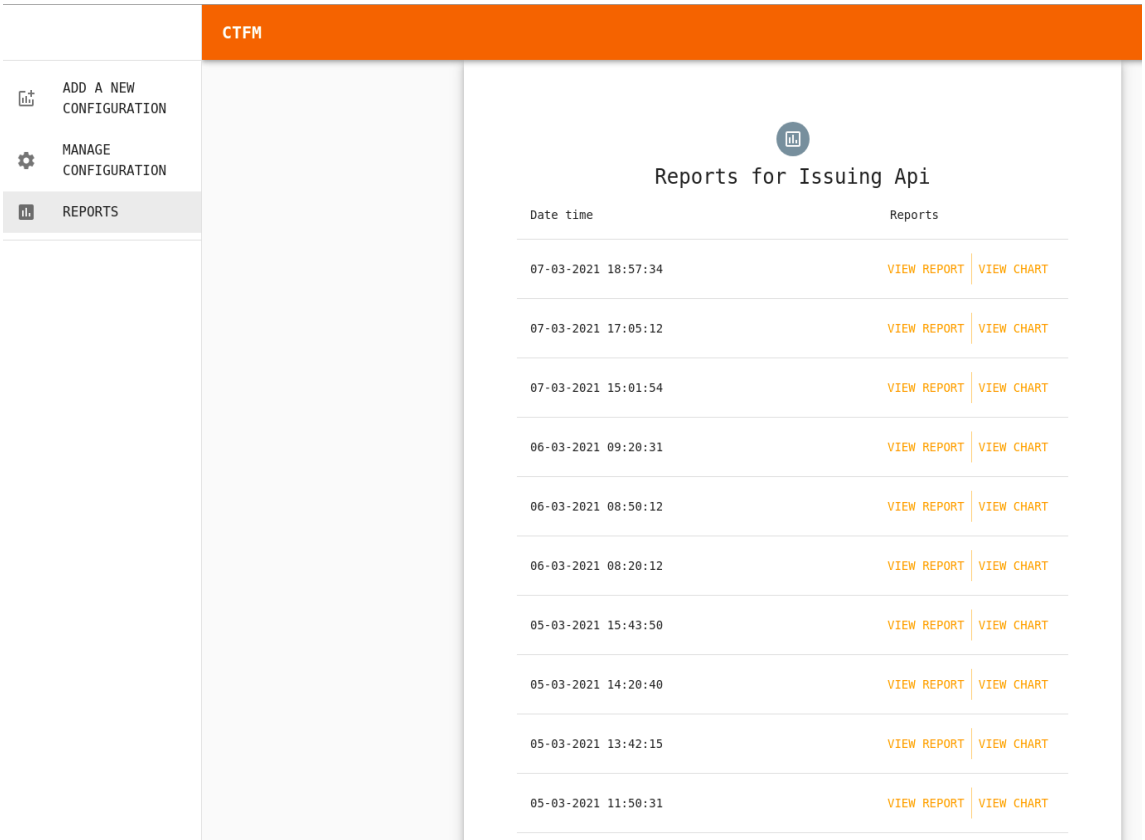


Figure 4.35 User interface of the reporting module

The above figure depicts the available reports for a specific producer on test cycle

CHAPTER 5 - EVALUATION

Under this section the testing framework is evaluated, to verify the POC would satisfy the scope of requirements under section 4.1 and the correct virtualization of the microservices as defined in its Open API specification.

5.1 CTFM vs. existing tools or technologies

Under this section, the CTFM will be compared with the existing tools or technologies

	CTFM	Hooverfly	WireMock	PACT
Could be applied at any development stage	Yes	No	No	No
Mandatory to be included as a third-party package or as a dependency into the source code	No	Yes	Yes	Yes
Capability to emulate services based on dynamic request inputs, without any pre-configured test cases	Yes	No	No	No
Capability to evolve or adapt the producer service changes without any developer input or refactoring in test cases.	Yes	No	No	No
Programming or scripting language support	Any	Limited	Limited	Limited
Record and replay requests	Not required	Yes	Yes	Yes

Table 5.1 CTFM vs. existing tools or technologies

The CTFM is flexible and easy to use out of the box at any development stage. Unlike the other tools, CTFM does not require to be pre-configured or pre-recorded with the potential test cases. Capability to evolve or adapt the producer service changes without any developer inputs or refactoring, targeting any test cases. Capability to emulate an unlimited number of valid responses based on the producer schema for any valid dynamic service request inputs. Programming language agnostic and is not required to be added as a third-party library or dependency into the producer or consumer source code.

5.2 Minimum server requirements and configurations

This section evaluates the frameworks' developer friendlessness and ability to run on any currently available Operating system, initialization configurations, and minimum requirements for the hardware and software resources. The testing framework is a web application that is capable of self-hosting by itself via running the Laravel artisan command “php artisan serve” in the terminal on any local environment which should satisfy the requirements in table 5.1 or the application is capable of running on any web server via the provided ”docker” and ”docker-compose” configurations and with the simple ”Make” commands, which will allow initializing the CTFM on any server with any currently available operating system.

Hardware	Software/Libraries/Packages/Extensions
1GB RAM	Operating system- Linux/macOs/Windows
10GB HDD	PHP >= 7.4
Intel/AMD processor >= 1.6GHz	BCMath PHP Extension
	Ctype PHP Extension
	Fileinfo PHP Extension
	JSON PHP Extension

	Mbstring PHP Extension
	OpenSSL PHP Extension
	PDO PHP Extension
	Tokenizer PHP Extension
	XML PHP Extension
	Composer
	NPM
	MongoDB >=3.*
	Docker
	Docker compose
	Make
	nginx

Table 5.2Testing framework minimum server requirements

Docker and docker-compose are tools that take the hassle away from a user, installing software, packages, or libraries into their operating system or on a server to host a web application, but this requires some knowledge in docker and docker-compose, to a user without having any prior knowledge in docker can use the testing framework with simple Make commands. This improves the developer-friendliness of the CTFM.

5.3 Configurations for written integration tests

This section evaluates and shows how the CTFM can be used for the existing written integration tests and explaining the limitations of the existing approaches. As an example tests written using Hooverfly is taken and guides how to run the tests with CTFM. A user can start using the framework after hosting it and by registering the required producer or producer configurations as in section 4.4, which requires its valid Open API specification. Then the integration tests can be carried out with the CTFM without any additional library or package or configurations to the consumer services. These also removes the burden of managing additional packages [33], unlike the tools like WireMock[20] or Hooverfly[21], where the user have to include its packages or libraries either to mock integration tests or record and replay test by adding a proxy to listen to server traffic as the webserver, this process is cumbersome and requires lots of time and energy and not possible to use the same integration tests with the real services unless modifying the integration tests. Below are few examples.

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = { Application.class }, webEnvironment =
WebEnvironment.DEFINED_PORT)
@FixMethodOrder(MethodSorters.NAME_ASCENDING)


public class AccountApiTest {


   protected Logger logger = Logger.getLogger(AccountApiFullTest.class.getName());


   @Autowired
   TestRestTemplate template;


   @ClassRule
   public static HoverflyRule hoverflyRule = HoverflyRule
         .inCaptureOrSimulationMode("accounts.json",
         HoverflyConfig.configs().proxyLocalHost()).printSimulationData();


   @Test
   public void addAccountTest() {
      Account a = new Account("A-35522334", 300, 222);
      ResponseEntity<Account> r = template.postForEntity("/account", a, Account.class);
      Assert.assertNotNull(r.getBody().getId());
      logger.info("New: " + r.getBody().getId());
   }
```

Figure 5.36 Hooverfly producer test with simulation capture mode

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
@FixMethodOrder(MethodSorters.NAME_ASCENDING)



public class ConsumerControllerTest {


   @Autowired
   TestRestTemplate template;


   @ClassRule
   public static HoverflyRule hoverflyRule = HoverflyRule
         .inSimulationMode(dsl(service("account-
service:2332").get(startsWith("/account/consumer/"))
               .willReturn(success("[{\"id\": 222,\"number\":\"A-35522334\"}]",
"application/json"))))
         .printSimulationData();


   @Test
   public void addConsumerTest() {
      Consumer consumer = new Consumer("A-35522334", "Chase Thomas",
CustomerType.OWN);
      consumer = template.postForObject("/consumer", consumer, Consumer.class);
   }


   @Test
   public void findConsumerWithAccounts() {
      Consumer consumer = template.getForObject("/consumer/account/{accountNumber}",
Consumer.class, "A-35522334");
      Consumer consumer = template.getForObject("/consumer/{id}", Consumer.class,
consumer.getId());
      Assert.assertTrue(cc.getAccounts().size() > 0);
   }
}
```

Figure 5.37 Hooverfly consumer test configured for a specific simulation

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.DEFINED_PORT)
@FixMethodOrder(MethodSorters.NAME_ASCENDING)



public class ConsumerControllerTest {


   @Autowired
   TestRestTemplate template;



   @Test
   public void addCustomerTest() {
      Consumer consumer = new Consumer("A-35522334", "Chase Thomas",
CustomerType.OWN);
      consumer = template.postForObject("/consumer", consumer, Consumer.class);
   }


   @Test
   public void findCustomerWithAccounts() {
      Consumer consumer = template.getForObject("/consumer/account/{accountNumber}",
Consumer.class, "A-35522334");
      Consumer consumer = template.getForObject("/consumer/{id}", Consumer.class,
consumer.getId());
      Assert.assertTrue(cc.getAccounts().size() > 0);
   }
```

Figure 5.38 Consumer test written to test with CTFM

To test with CTFM doesn't require any additional configuration or packages or libraries or the need to record and replay requests. The integration tests can be run against the CTFM framework with the producer service URL pointed to CTFM producer proxy URL under consumer service URL configurations. If the user wants to test any edge cases or failures can follow the steps under Chapter 4 to produce such cases.

5.4 Real services vs. CTFM virtualized services

Under this section, the POC is evaluated against real microservices. The main objective of the evaluation is to verify the correct virtualization of the producer services. The producer service REST API will be tested for its functionality, behaviour, service availability and acceptance, request and response assertions, consistency, and for runtime errors [47].

CTFM will be configured for the respective producers' Open API specification. The same set of tests will be executed against the POC which should behave similarly to the producer and should send correct and consistent auto generated responses based on the producer OAS. During the tests authentication and authorization is not taken into consideration [35]. The following table describes the producer services, tests are created and carried out using the Apache Jmeter API the testing tool.

	Microservice	Card		
	REST services			
	HTTP method	**Path and parameters**	**JSON Payload**	**HTTP status: JSON response**
1	POST	/card/{token}/activate	{ "pan_last_digits": integer, "cardholder": { "title": "string", "first_name": "string", "last_name": "string", "dob": null, "address_line_1": "string", "address_line_2": "string", "address_line_3": "string", "post_code": "string", "city": "string", "country_code": "BE" }	200: {} 400: { "error": "Bad request.", "message": "Card cannot be activated invalid card" } 401:{ "message": "Unauthenticated" } 404: { "error": "Not found.", "message": "Card not found." }

			}	422:{
				"message": "The given data was invalid.",
				"errors": {}
				}

				500:{
				"error": "GPS error.",
				"message": "Unavailable"
				}

2	GET	/card/{token}/pin		200 : {
				"card_pin": "string"
				}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

3	PUT	/card/{token}/pin	{
			"pin": integer,
			"confirm_pin": integer
			}
				200:{}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": " Card not found."
				}

4	PATCH	/card/{token}/status	{
			"status": "active"
			}
				200: {}

				400: {
				"error": "Bad request.",
				"message": "Card status cannot be changed"
				}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

				422:{ "message": "The given data was invalid.",

				"errors": {} }

				500:{
				"error": "GPS error.",
				"message": "Unavailable"
				}

5	POST	/card/{token}/3ds	{
			"mobile": "string"
			}
				200: {}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

				422:{
				"message": "The given data was invalid.",
				"errors": {}
				}

6	DELETE	/card/{token}/3ds		200: {}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

7	GET	/card/{token}		200:[
				{
				"balance_available": 0,
				"balance_reserved": 0,
				"expiration_date": null,
				"is_loadable": true,
				"status": "pending",
				"card_brand": {
				"id": 0,
				"name": "string",
				"status": "uncon-firmed",
				"sta-tus_changed_date": “date”
				}
				}
				]

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

8	GET	/card/{token}/transaction		200:{
				"meta": {
				"total": 0,
				"per_page": 0,
				"current_page": 0,
				"from": 0,
				"to": 0
				},
				"data": [
				{
				"ehi_details": "string",
				"ehi_status": "string",
				"acquirer_id": 0,
				"auth_code": "string",
				"bill_amount": 0,
				"bill_ccy": "string",
				"fx_pad": "string",
				"fee_fixed": "string",
				"fee_rate": "string",
				"mcc_code": "string",
				"mcc_description": "string",
				"mcc_pad": "string",
				"merchant_id": "string",
				"merchant_name": "string",
				"note": "string",
				"pos_terminal": "string",
				"token": "string",

				"transaction_id": "string", "transaction_link": "string", "transaction_amount": "string", "transaction_ccy": "string", "transaction_country": "string", "transaction_description": "string", "transaction_gps_date": null, "transaction_status_code": "string", "transaction_type": "string" }] }

				401:{
				"message": "Unauthen-ticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

9	PATCH	/card/{token}/distributor	{
			"distributor_id": “integer”
			}
				204:{}

				401:{
				"message": "Unauthen-ticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

10	POST	card/{token}/transfer-balance	{
			"receiving_card_token": "string",
			"currency_code": "string",
			"description": "string"
			}
				200:{}

				401:{
				"message": "Unauthen-ticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

11	POST	card/{token}/pin/send	{
			"delivery_phone": "string"
			}
				200:{}

				401:{
				"message": "Unauthenticated"
				}

				404: {
				"error": "Not found.",
				"message": "Card not found."
				}

Table 5.3: Selected services for testing

5.4.1 Test results interpretation

Forty test cases were created for the producer services functionality each test case will evaluate, requests' for expected response code and the response path assertions based on request path parameters or request body parameters. Following is a sample groovy response assertion written for specific test case for Apache Jmeter.

```
import groovy.json.JsonSlurper;


def failureMessage = "";
def jsonResponse = null;


JsonSlurper JSON = new JsonSlurper ();
try {
   jsonResponse = JSON.parseText(prev.getResponseDataAsString());
   } catch (Exception e) {
   failureMessage += "Invalid JSON.\n"
}
if(!"200".equals(prev.getResponseCode())){
failureMessage += "Expected <response code> [200] but we got instead [" + prev.getResponseCode() + "]\n\n" ;
}
if (!jsonResponse.keySet().containsAll(["data","meta"] )) {
        failureMessage += "The json config primary key element has a wrong structure.\n\n";
}
def dataTransaction = jsonResponse.data[0];
if (!dataTransaction.keySet().containsAll(["note","transaction_amount", "merchant_id", "pos_terminal",
"transaction_country", "mcc_code", "fee_fixed", "transaction_id", "transaction_status_code", "ehi_details", "bill_amount",
"transaction_link", "merchant_name", "bill_ccy", "transaction_description", "transaction_type", "auth_code", "fee_rate", "token", "fx_pad",
"ehi_status", "mcc_description", "transaction_ccy", "acquirer_id", "mcc_pad"] )) {
        failureMessage += "The json config child key element has a wrong structure.\n\n";
}
if (failureMessage?.trim()) {
AssertionResult.setFailureMessage(failureMessage);
    AssertionResult.setFailure(true);
}
```

Figure 5.39 Groovy response assertion for /card/${token}/transaction

5.4.1.1 Producer services test results

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput	Received K...	Sent KB/sec
404 POST /card/378433069/transfer-balance	1	1188	1188	1188	1188	1188	1188	1188	0.00%	50.5/min	0.54	1.16
200 POST card/436247292/pin/send	1	1180	1180	1180	1180	1180	1180	1180	0.00%	50.8/min	0.53	1.10
401 POST card/436247292/pin/send	1	1166	1166	1166	1166	1166	1166	1166	0.00%	51.5/min	0.57	1.13
404 POST card/436247292/pin/send	1	1176	1176	1176	1176	1176	1176	1176	0.00%	51.0/min	0.58	1.12
422 POST card/436247292/pin/send	1	1175	1175	1175	1175	1175	1175	1175	0.00%	51.1/min	0.64	1.10
TOTAL	40	1250	1188	1209	2182	2212	680	2212	0.00%	47.9/min	0.58	1.01

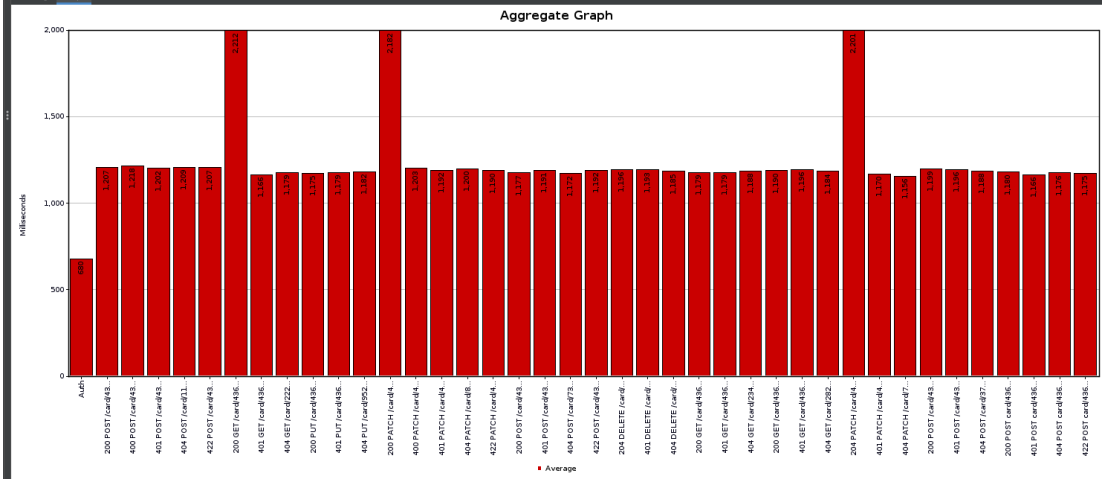


Figure 5.40: Producer services test response times

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
401 POST /card/436247292/activate	1	1202	1202	1202	0.00	0.00%	49.9/min	0.52	1.09	639.0
404 POST /card/111111111/activate	1	1209	1209	1209	0.00	0.00%	49.6/min	0.53	1.08	661.0
422 POST /card/436247292/activate	1	1207	1207	1207	0.00	0.00%	49.7/min	0.59	1.08	734.0
200 GET /card/436247292/pin	1	2212	2212	2212	0.00	0.00%	27.1/min	0.29	0.56	657.0
401 GET /card/436247292/pin	1	1166	1166	1166	0.00	0.00%	51.5/min	0.54	1.07	639.0
404 GET /card/222222222/pin	1	1179	1179	1179	0.00	0.00%	50.9/min	0.55	1.06	661.0
200 PUT /card/436247292/pin	1	1175	1175	1175	0.00	0.00%	51.1/min	0.53	1.11	641.0
401 PUT /card/436247292/pin	1	1179	1179	1179	0.00	0.00%	50.9/min	0.56	1.12	675.0
404 PUT /card/952295944/pin	1	1182	1182	1182	0.00	0.00%	50.8/min	0.58	1.12	697.0
200 PATCH /card/436247292/status	1	2182	2182	2182	0.00	0.00%	27.5/min	0.27	0.59	604.0
400 PATCH /card/436247292/status	1	1203	1203	1203	0.00	0.00%	49.9/min	0.54	1.08	665.0
401 PATCH /card/436247292/status	1	1192	1192	1192	0.00	0.00%	50.3/min	0.52	1.09	639.0
404 PATCH /card/825150591/status	1	1200	1200	1200	0.00	0.00%	50.0/min	0.54	1.08	661.0
422 PATCH /card/436247292/status	1	1190	1190	1190	0.00	0.00%	50.4/min	0.59	1.07	716.0
200 POST /card/436247292/3ds	1	1177	1177	1177	0.00	0.00%	51.0/min	0.50	1.09	604.0
401 POST /card/436247292/3ds	1	1191	1191	1191	0.00	0.00%	50.4/min	0.52	1.09	639.0
404 POST /card/730078801/3ds	1	1172	1172	1172	0.00	0.00%	51.2/min	0.55	1.11	661.0
422 POST /card/436247292/3ds	1	1192	1192	1192	0.00	0.00%	50.3/min	0.59	1.07	716.0
204 DELETE /card/436247292/3ds	1	1196	1196	1196	0.00	0.00%	50.2/min	0.44	1.05	540.0
401 DELETE /card/436247292/3ds	1	1193	1193	1193	0.00	0.00%	50.3/min	0.52	1.07	639.0
404 DELETE /card/100595448/3ds	1	1185	1185	1185	0.00	0.00%	50.6/min	0.54	1.07	661.0
200 GET /card/436247292	1	1179	1179	1179	0.00	0.00%	50.9/min	0.61	1.04	738.0
401 GET /card/436247292	1	1179	1179	1179	0.00	0.00%	50.9/min	0.53	1.06	639.0
404 GET /card/234956818	1	1188	1188	1188	0.00	0.00%	50.5/min	0.54	1.05	661.0
200 GET /card/436247292/transaction	1	1190	1190	1190	0.00	0.00%	50.4/min	1.92	1.04	2334.0
401 GET /card/436247292/transaction	1	1196	1196	1196	0.00	0.00%	50.2/min	0.52	1.05	639.0
404 GET /card/282353035/transaction	1	1184	1184	1184	0.00	0.00%	50.7/min	0.55	1.06	661.0
204 PATCH /card/436247292/distributor	1	2201	2201	2201	0.00	0.00%	27.3/min	0.24	0.58	540.0
401 PATCH /card/436247292/distributor	1	1170	1170	1170	0.00	0.00%	51.3/min	0.53	1.12	639.0
404 PATCH /card/732989652/distributor	1	1156	1156	1156	0.00	0.00%	51.9/min	0.56	1.13	661.0
200 POST /card/436247292/transfer-balance	1	1199	1199	1199	0.00	0.00%	50.0/min	0.49	1.13	604.0
401 POST /card/436247292/transfer-balance	1	1196	1196	1196	0.00	0.00%	50.2/min	0.52	1.15	639.0
404 POST /card/378433069/transfer-balance	1	1188	1188	1188	0.00	0.00%	50.5/min	0.54	1.16	661.0
200 POST card/436247292/pin/send	1	1180	1180	1180	0.00	0.00%	50.8/min	0.53	1.10	641.0
401 POST card/436247292/pin/send	1	1166	1166	1166	0.00	0.00%	51.5/min	0.57	1.13	675.0
404 POST card/436247292/pin/send	1	1176	1176	1176	0.00	0.00%	51.0/min	0.58	1.12	697.0
422 POST card/436247292/pin/send	1	1175	1175	1175	0.00	0.00%	51.1/min	0.64	1.10	768.0
TOTAL	40	1250	680	2212	281.61	0.00%	47.9/min	0.58	1.01	747.8

Figure 5.41: Producer services test results complete summary

The producer service database is seeded for the test cases and the tests are executed to verify the requests' parameters, validations, and response structure. The producer completed the tests with an average response time of 1.2 seconds and without any assertion failures.

5.4.1.2 CTFM virtualized producer services test results

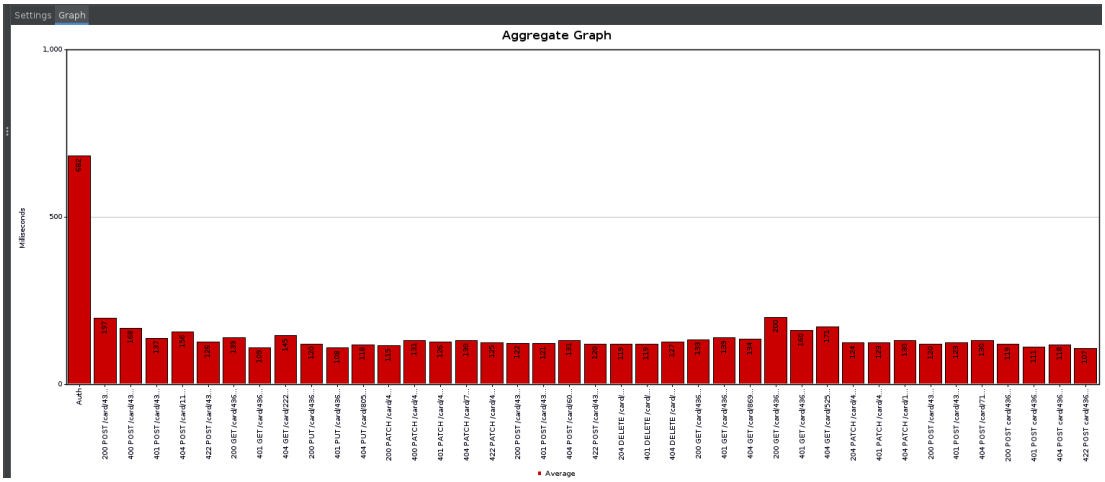


Figure 5.42: CTFM virtualized services test response times

Label ↓	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
422 POST /card/436247292/3ds	1	120	120	120	0.00	0.00%	8.3/sec	3.22	10.57	396.0
422 PATCH /card/436247292/status	1	125	125	125	0.00	0.00%	8.0/sec	3.09	10.16	396.0
404 PUT /card/80514640/pin	1	118	118	118	0.00	0.00%	8.5/sec	2.95	11.19	357.0
404 POST card/436247292/pin/send	1	118	118	118	0.00	0.00%	8.5/sec	2.86	11.12	346.0
404 POST /card/719742693/transfer-balance	1	130	130	130	0.00	0.00%	7.7/sec	2.55	10.58	340.0
404 POST /card/608767109/3ds	1	131	131	131	0.00	0.00%	7.6/sec	2.55	9.91	342.0
404 POST /card/111111111/activate	1	156	156	156	0.00	0.00%	6.4/sec	2.23	8.34	357.0
404 PATCH /card/743859609/status	1	130	130	130	0.00	0.00%	7.7/sec	2.55	9.97	340.0
404 PATCH /card/106430908/distributor	1	130	130	130	0.00	0.00%	7.7/sec	2.58	10.02	343.0
404 GET /card/869599725	1	134	134	134	0.00	100.00%	7.5/sec	3.05	9.27	419.0
404 GET /card/525618109/transaction	1	171	171	171	0.00	0.00%	5.8/sec	21.54	7.33	3771.0
404 GET /card/222222222/pin	1	145	145	145	0.00	0.00%	6.9/sec	2.40	8.59	357.0
404 DELETE /card/390392612/3ds	1	127	127	127	0.00	0.00%	7.9/sec	2.67	9.98	347.0
401 PUT /card/436247292/pin	1	108	108	108	0.00	0.00%	9.3/sec	2.96	12.23	327.0
401 POST card/436247292/pin/send	1	111	111	111	0.00	0.00%	9.0/sec	2.88	11.82	327.0
401 POST /card/436247292/transfer-balance	1	123	123	123	0.00	0.00%	8.1/sec	2.56	11.18	322.0
401 POST /card/436247292/activate	1	137	137	137	0.00	0.00%	7.3/sec	2.30	9.50	322.0
401 POST /card/436247292/3ds	1	121	121	121	0.00	0.00%	8.3/sec	2.60	10.73	322.0
401 PATCH /card/436247292/status	1	126	126	126	0.00	0.00%	7.9/sec	2.50	10.28	322.0
401 PATCH /card/436247292/distributor	1	123	123	123	0.00	0.00%	8.1/sec	2.56	10.59	322.0
401 GET /card/436247292/transaction	1	160	160	160	0.00	100.00%	6.2/sec	12.70	7.84	2080.0
401 GET /card/436247292/pin	1	109	109	109	0.00	0.00%	9.2/sec	2.88	11.43	322.0
401 GET /card/436247292	1	139	139	139	0.00	0.00%	7.2/sec	2.26	8.94	322.0
401 DELETE /card/436247292/3ds	1	119	119	119	0.00	0.00%	8.4/sec	2.64	10.65	322.0
400 POST /card/436247292/activate	1	168	168	168	0.00	0.00%	6.0/sec	2.24	7.75	385.0
400 PATCH /card/436247292/status	1	131	131	131	0.00	0.00%	7.6/sec	2.68	9.89	360.0
204 PATCH /card/436247292/distributor	1	124	124	124	0.00	100.00%	8.1/sec	2.24	10.35	284.0
204 DELETE /card/436247292/3ds	1	119	119	119	0.00	100.00%	8.4/sec	2.33	10.49	284.0
200 PUT /card/436247292/pin	1	120	120	120	0.00	0.00%	8.3/sec	2.35	10.84	289.0
200 POST card/436247292/pin/send	1	119	119	119	0.00	0.00%	8.4/sec	2.37	10.87	289.0
200 POST /card/436247292/transfer-balance	1	120	120	120	0.00	0.00%	8.3/sec	2.31	11.30	284.0
200 POST /card/436247292/activate	1	197	197	197	0.00	0.00%	5.1/sec	1.41	6.51	284.0
200 POST /card/436247292/3ds	1	122	122	122	0.00	0.00%	8.2/sec	2.27	10.48	284.0
200 PATCH /card/436247292/status	1	115	115	115	0.00	0.00%	8.7/sec	2.41	11.10	284.0
200 GET /card/436247292/transaction	1	200	200	200	0.00	0.00%	5.0/sec	19.03	6.17	3897.0
200 GET /card/436247292/pin	1	139	139	139	0.00	0.00%	7.2/sec	2.23	8.86	318.0
200 GET /card/436247292	1	133	133	133	0.00	0.00%	7.5/sec	3.02	9.19	411.0
TOTAL	40	145	107	682	88.40	10.00%	6.7/sec	4.04	8.54	613.8

Figure 5.43: CTFM virtualized producer services test results complete summary

Category	No. of requests	%
Success	36	90%
Failure	4	10%
Total	40	

Table 5.4: CTFM Virtualized test results summary

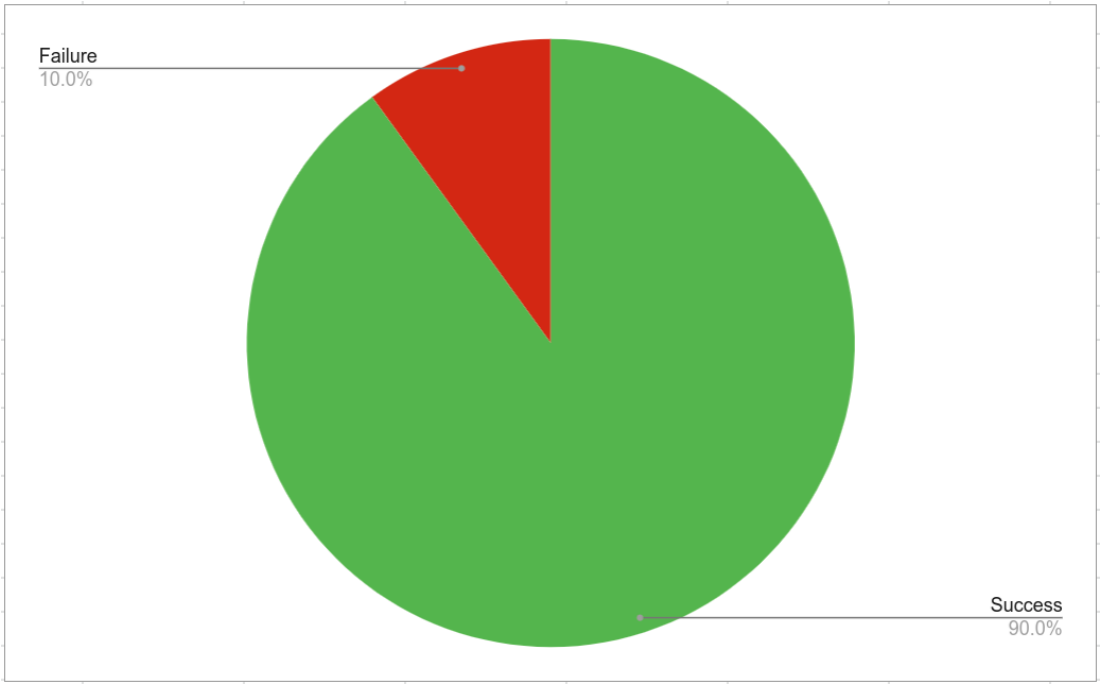


Figure 5.44: CTFM virtualized test results summary graphical presentation

The CTFM was able to successfully virtualize 36/40 requests test cases with an average response time of 145ms. The observed failures are further investigated for the potential causes. The further analysis of the failures are as follows.

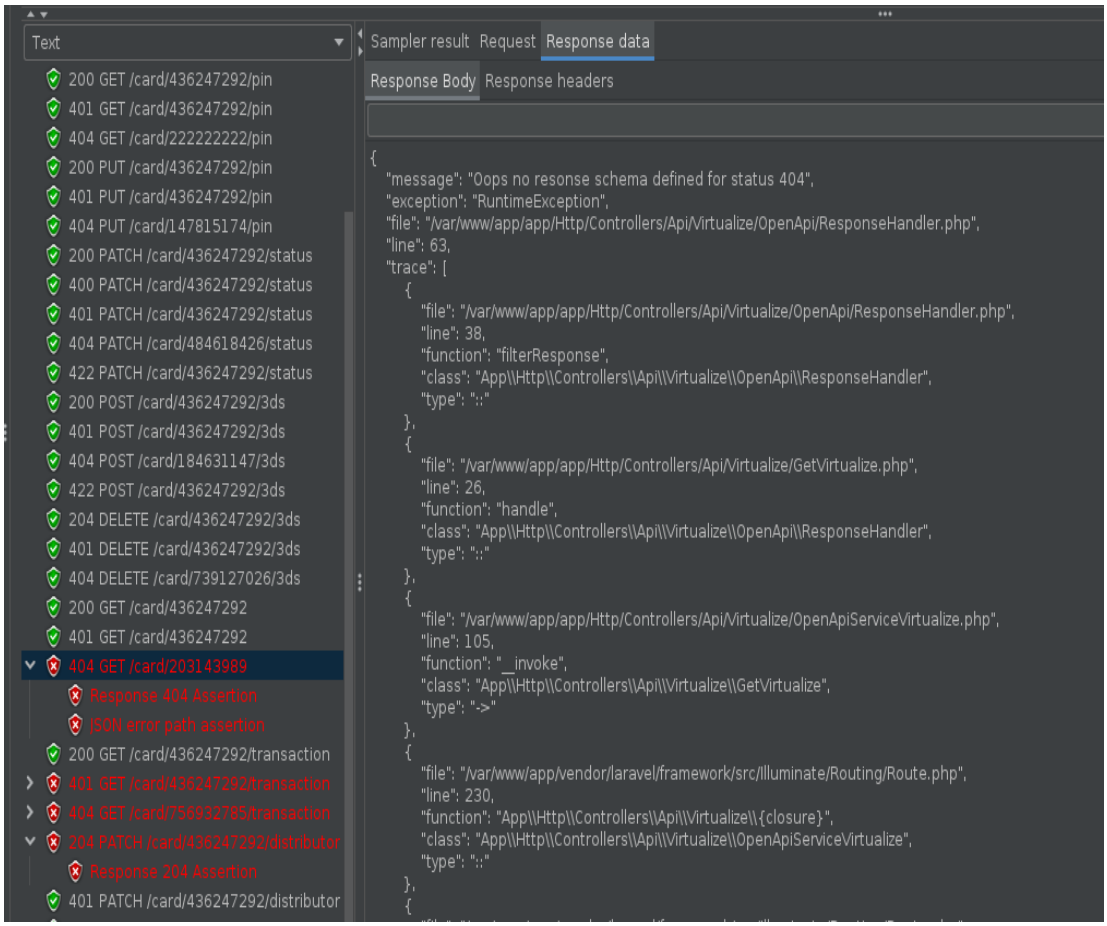


Figure 5.45: Response assertion failure due to unavailable response definition

Under this test case the response error status 404 is expected with an error response format but the CTFM is complaining that the expected error is not defined in the request GET *card*/{token} response schema. This is due to the developer's mistake lacking to document the expected responses in the request-response specification.

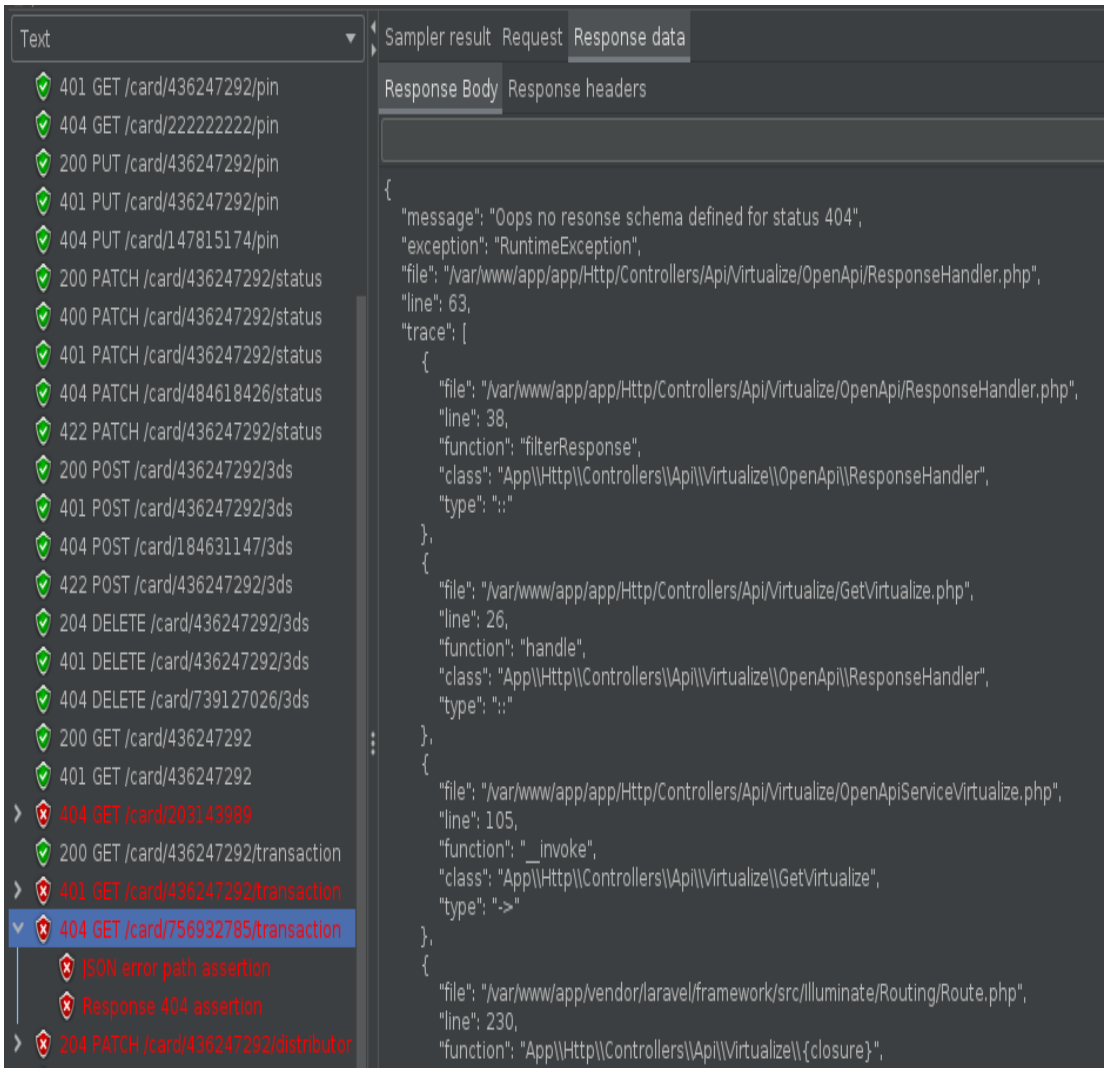


Figure 5.46: Response assertion failure due to unavailable response definition

Under this test case expects error status 404 with an error response format also in this scenario the CTFM is complaining that the expected error is not defined in the request GET *card*/{token}/transaction response schema. This scenario is also due to the developer's lack of documenting the request-response specification.

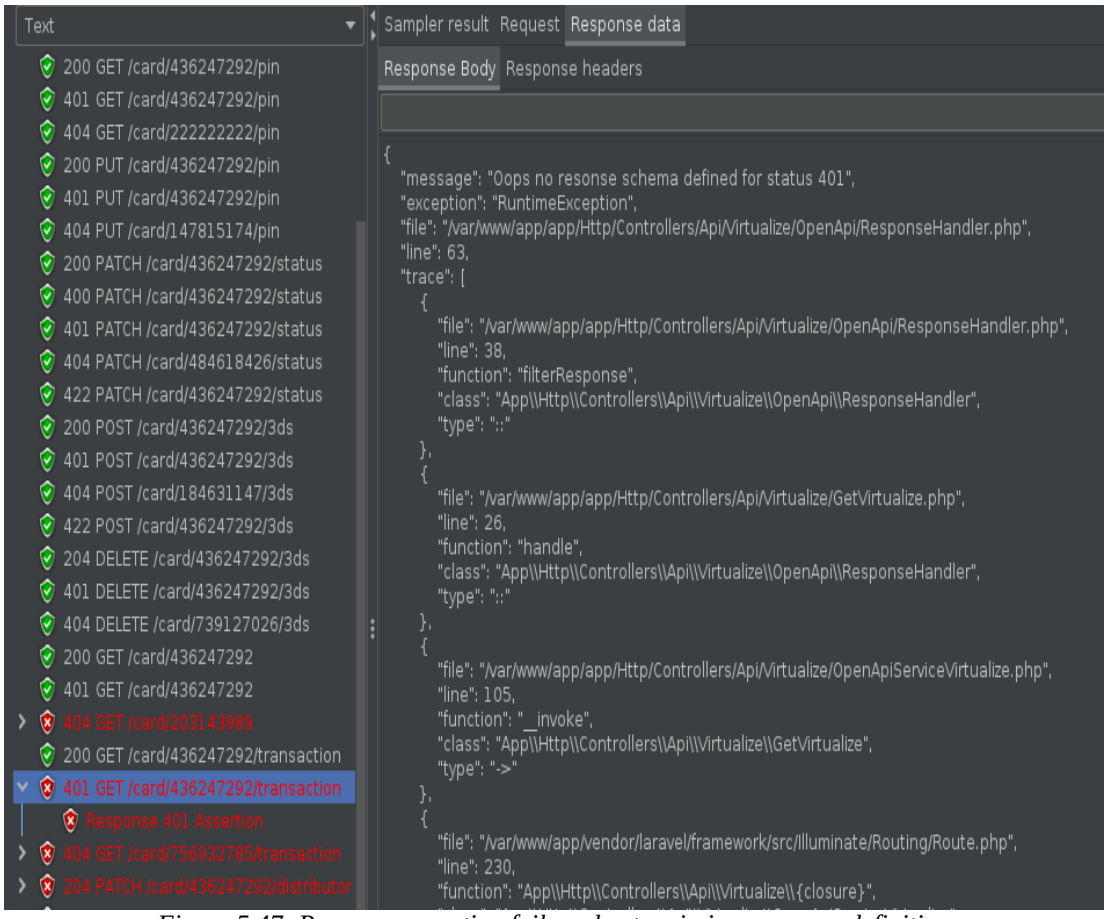


Figure 5.47: Response assertion failure due to missing response definition

Under this test case the expected response status is 401 with and error response format. In this case also CTFM complaining about unavailable response schema for status 404.

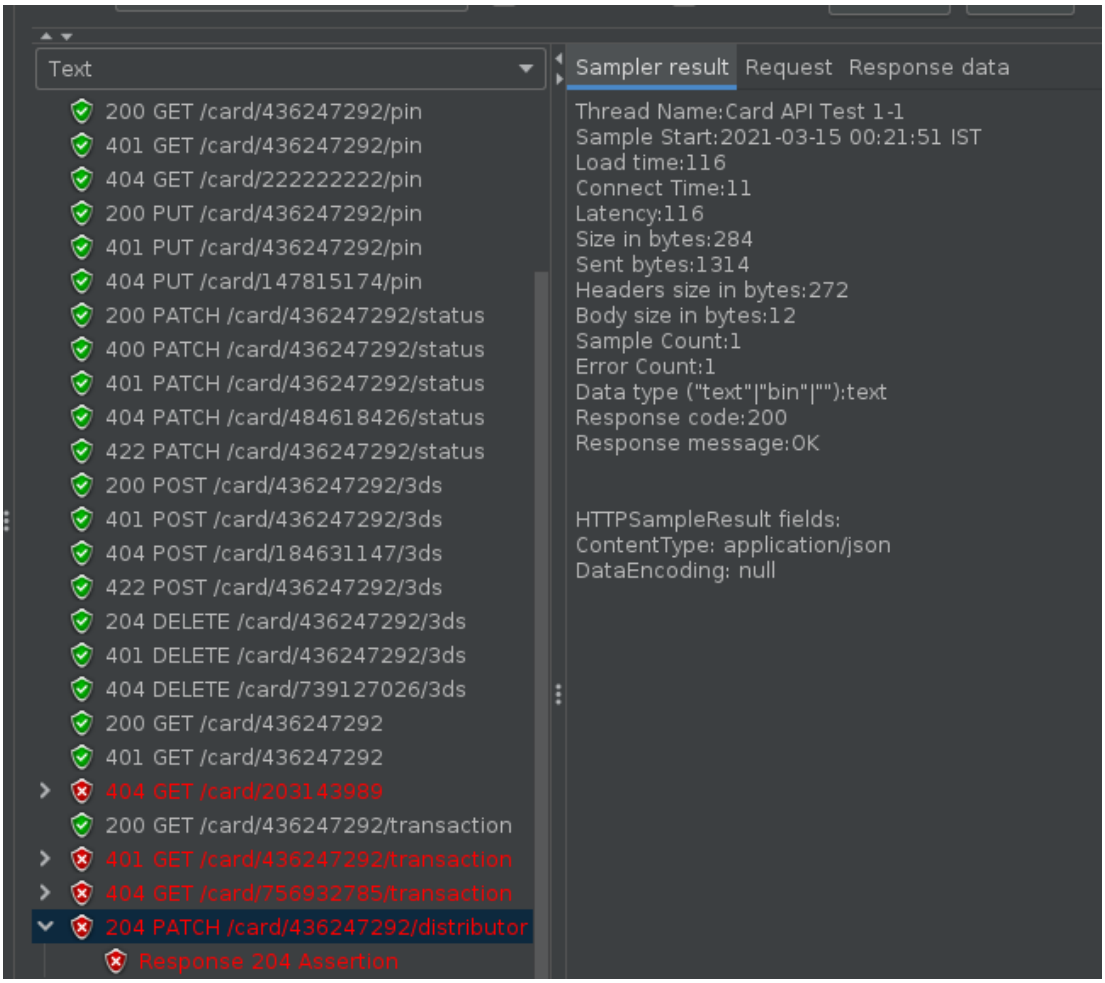


Figure 5.48: Response assertion failure due wrong response definition

Under this test case the expectation is status 204 but in the request potential response definitions it has documented status 200 with no content. This is caused due to a developer mistake, has defined invalid status for the success response.

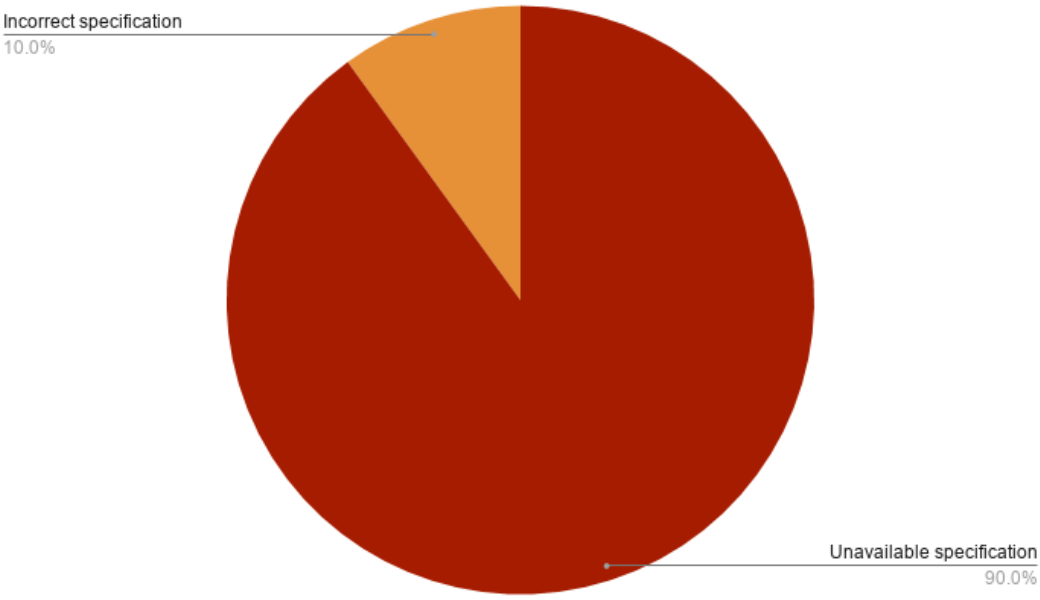


Figure 5.49: Tests failure composition

In a summary, the POC system performed as expected during the test cycles. It accepted and returned consistent valid auto-generated responses within the bounds of defined configured Open API specification of the respective producer services. The specification should be valid up to date and consistent, it should not contain any errors. The specification is the main document that is used by the integrators, this is one of the key reasons to adopt Open API, which helps actively maintain and evolve the API with the services. Usually, these are mitigated with developer review cycles. The reported failures are corrected after updating the configuration with the missing schema definitions. Then the new schema is added to the CTFM by replacing the existing schema and the test is executed, results are as follows and interpreted.

5.4.1.3 CTFM results with corrected producer service specification

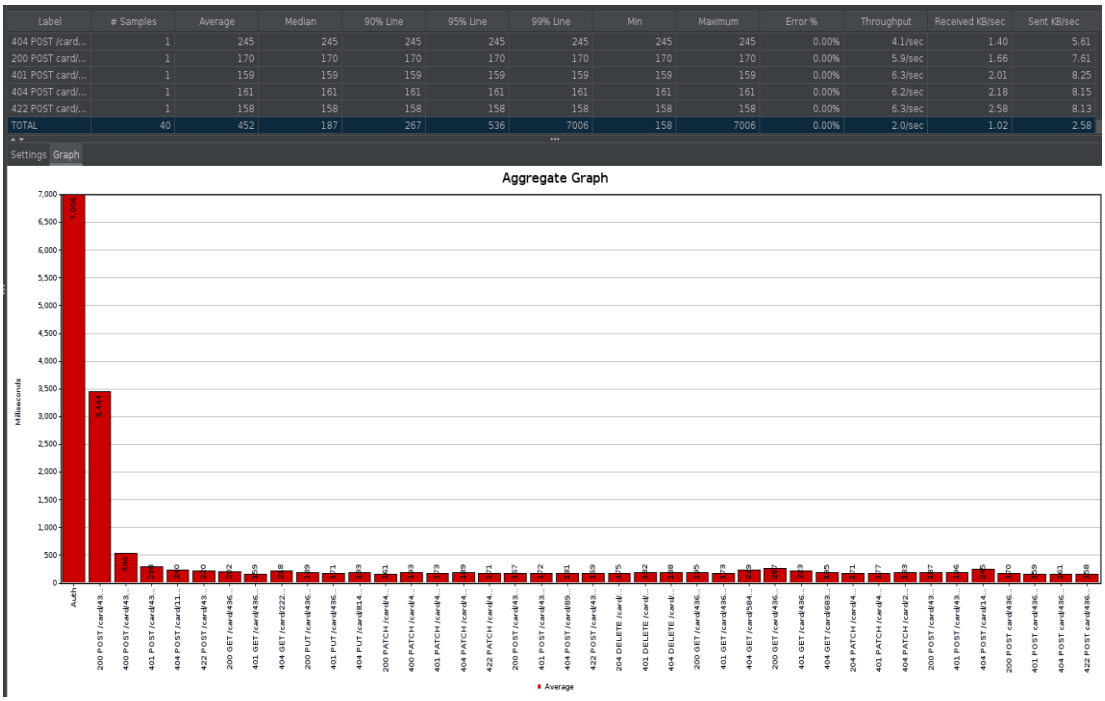


Figure 5.50: CTFM results with corrected producer specification

The above figure shows the results after correcting the producer specification, which was adding the missing specifications and correcting the reported incorrect specification.

The above scenario is useful for the integrators when they will find out that the system doesn't support existing behaviour or schema definition change. It'll help them to verify with the producer service maintainers whether to come to a common agreement, which party should adopt the change.

5.5 **Performance test**

The section will test the CTFM performance how it behaves based on the number of users and requests. Initially for a single user then gradually increase for 10 and 100 users. Gathered results are as follows.

Hardware	Software
16GB RAM	Operating system – PopOS(Debian)
256GBM2, 1TB HDD	Testing tool – Apache Jmeter
Intel Core i5 1.6GHz	

Table 5.5: Testing resources and environment

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
401 POST /card/436247292/activate	1	118	118	118	0.00	0.00%	8.5/sec	2.66	11.03	322.0
404 POST /card/111111111/activate	1	126	126	126	0.00	0.00%	7.9/sec	2.67	10.33	344.0
422 POST /card/436247292/activate	1	120	120	120	0.00	0.00%	8.3/sec	3.37	10.81	414.0
200 GET /card/436247292/pin	1	133	133	133	0.00	0.00%	7.5/sec	2.43	9.26	331.0
401 GET /card/436247292/pin	1	106	106	106	0.00	0.00%	9.4/sec	2.97	11.76	322.0
404 GET /card/222222222/pin	1	126	126	126	0.00	0.00%	7.9/sec	2.66	9.89	343.0
200 PUT /card/436247292/pin	1	119	119	119	0.00	0.00%	8.4/sec	2.37	10.93	289.0
401 PUT /card/436247292/pin	1	111	111	111	0.00	0.00%	9.0/sec	2.88	11.89	327.0
404 PUT /card/951940865/pin	1	117	117	117	0.00	0.00%	8.5/sec	2.90	11.28	347.0
200 PATCH /card/436247292/status	1	104	104	104	0.00	0.00%	9.6/sec	2.67	12.27	284.0
400 PATCH /card/436247292/status	1	128	128	128	0.00	0.00%	7.8/sec	2.85	10.12	373.0
401 PATCH /card/436247292/status	1	119	119	119	0.00	0.00%	8.4/sec	2.64	10.89	322.0
404 PATCH /card/307069152/status	1	126	126	126	0.00	0.00%	7.9/sec	2.68	10.28	346.0
422 PATCH /card/436247292/status	1	120	120	120	0.00	0.00%	8.3/sec	3.22	10.59	396.0
200 POST /card/436247292/3ds	1	115	115	115	0.00	0.00%	8.7/sec	2.41	11.12	284.0
401 POST /card/436247292/3ds	1	119	119	119	0.00	0.00%	8.4/sec	2.64	10.91	322.0
404 POST /card/127368869/3ds	1	127	127	127	0.00	0.00%	7.9/sec	2.63	10.22	342.0
422 POST /card/436247292/3ds	1	115	115	115	0.00	0.00%	8.7/sec	3.36	11.03	396.0
204 DELETE /card/436247292/3ds	1	115	115	115	0.00	100.00%	8.7/sec	2.41	10.85	284.0
401 DELETE /card/436247292/3ds	1	120	120	120	0.00	0.00%	8.3/sec	2.62	10.56	322.0
404 DELETE /card/380518696/3ds	1	131	131	131	0.00	0.00%	7.6/sec	2.56	9.68	344.0
200 GET /card/436247292	1	131	131	131	0.00	0.00%	7.6/sec	3.11	9.33	417.0
401 GET /card/436247292	1	117	117	117	0.00	0.00%	8.5/sec	2.69	10.62	322.0
404 GET /card/120446111	1	127	127	127	0.00	100.00%	7.9/sec	3.24	9.78	421.0
200 GET /card/436247292/transaction	1	170	170	170	0.00	0.00%	5.9/sec	16.84	7.26	2931.0
401 GET /card/436247292/transaction	1	169	169	169	0.00	100.00%	5.9/sec	17.35	7.42	3002.0
404 GET /card/315334164/transaction	1	195	195	195	0.00	0.00%	5.1/sec	27.54	6.43	5499.0
204 PATCH /card/436247292/distributor	1	119	119	119	0.00	100.00%	8.4/sec	2.33	10.78	284.0
401 PATCH /card/436247292/distributor	1	117	117	117	0.00	0.00%	8.5/sec	2.69	11.13	322.0
404 PATCH /card/317386767/distributor	1	124	124	124	0.00	0.00%	8.1/sec	2.74	10.51	348.0
200 POST /card/436247292/transfer-balance	1	116	116	116	0.00	0.00%	8.6/sec	2.39	11.69	284.0
401 POST /card/436247292/transfer-balance	1	118	118	118	0.00	0.00%	8.5/sec	2.66	11.65	322.0
404 POST /card/934845165/transfer-balance	1	126	126	126	0.00	0.00%	7.9/sec	2.75	10.91	355.0
200 POST card/436247292/pin/send	1	118	118	118	0.00	0.00%	8.5/sec	2.39	10.96	289.0
401 POST card/436247292/pin/send	1	107	107	107	0.00	0.00%	9.3/sec	2.98	12.27	327.0
404 POST card/436247292/pin/send	1	115	115	115	0.00	0.00%	8.7/sec	2.96	11.41	349.0
422 POST card/436247292/pin/send	1	104	104	104	0.00	0.00%	9.6/sec	3.92	12.35	417.0
TOTAL	40	138	104	686	89.42	10.00%	7.2/sec	4.60	9.07	656.9

Figure 5.51: Average response time for a single user

Label ↓	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
404 GET /card/221929001	1	382	382	382	0.00	100.00%	2.6/sec	1.08	3.25	424.0
404 GET /card/187971031	1	401	401	401	0.00	100.00%	2.5/sec	1.03	3.10	421.0
404 GET /card/148091543/transaction	1	419	419	419	0.00	0.00%	2.4/sec	12.60	2.99	5405.0
404 DELETE /card/934418687/3ds	1	441	441	441	0.00	0.00%	2.3/sec	0.76	2.87	343.0
404 DELETE /card/877321494/3ds	1	423	423	423	0.00	0.00%	2.4/sec	0.84	3.00	362.0
404 DELETE /card/832211321/3ds	1	475	475	475	0.00	0.00%	2.1/sec	0.70	2.67	340.0
404 DELETE /card/759794301/3ds	1	408	408	408	0.00	0.00%	2.5/sec	0.82	3.11	343.0
404 DELETE /card/713266207/3ds	1	479	479	479	0.00	0.00%	2.1/sec	0.69	2.65	340.0
404 DELETE /card/694994523/3ds	1	384	384	384	0.00	0.00%	2.6/sec	0.88	3.30	347.0
404 DELETE /card/666868683/3ds	1	455	455	455	0.00	0.00%	2.2/sec	0.77	2.79	357.0
404 DELETE /card/565748830/3ds	1	444	444	444	0.00	0.00%	2.3/sec	0.78	2.85	355.0
404 DELETE /card/510123907/3ds	1	466	466	466	0.00	0.00%	2.1/sec	0.71	2.72	341.0
404 DELETE /card/462678737/3ds	1	343	343	343	0.00	0.00%	2.9/sec	0.97	3.70	342.0
401 PUT /card/436247292/pin	10	370	323	404	20.95	0.00%	2.7/sec	0.86	3.57	327.0
401 POST card/436247292/pin/send	10	269	146	341	71.88	0.00%	3.8/sec	1.21	4.97	327.0
401 POST /card/436247292/transfer-balance	10	341	227	419	69.47	0.00%	3.2/sec	1.00	4.37	322.0
401 POST /card/436247292/activate	10	357	225	502	81.62	0.00%	2.8/sec	0.88	3.65	322.0
401 POST /card/436247292/3ds	10	409	350	471	37.71	0.00%	2.4/sec	0.76	3.15	322.0
401 PATCH /card/436247292/status	10	436	333	591	69.61	0.00%	2.5/sec	0.79	3.27	322.0
401 PATCH /card/436247292/distributor	10	395	312	500	48.71	0.00%	2.9/sec	0.92	3.81	322.0
401 GET /card/436247292/transaction	10	494	376	633	67.03	100.00%	2.6/sec	9.26	3.28	3622.5
401 GET /card/436247292/pin	10	411	353	544	58.58	0.00%	2.5/sec	0.79	3.14	322.0
401 GET /card/436247292	10	389	351	522	51.75	0.00%	2.6/sec	0.81	3.21	322.0
401 DELETE /card/436247292/3ds	10	415	326	491	49.56	0.00%	2.4/sec	0.75	3.04	322.0
400 POST /card/436247292/activate	10	424	257	591	85.04	0.00%	2.9/sec	1.09	3.73	389.3
400 PATCH /card/436247292/status	10	427	347	547	61.07	0.00%	2.5/sec	0.91	3.28	369.5
204 PATCH /card/436247292/distributor	10	374	307	426	34.26	100.00%	2.9/sec	0.81	3.75	284.0
204 DELETE /card/436247292/3ds	10	398	331	451	34.07	100.00%	2.4/sec	0.67	3.03	284.0
200 PUT /card/436247292/pin	10	389	332	467	39.15	0.00%	2.7/sec	0.76	3.48	289.0
200 POST card/436247292/pin/send	10	322	239	445	70.65	0.00%	3.5/sec	0.99	4.51	289.0
200 POST /card/436247292/transfer-balance	10	345	261	415	52.44	0.00%	3.1/sec	0.86	4.18	284.0
200 POST /card/436247292/activate	10	359	213	471	87.26	0.00%	3.1/sec	0.85	3.92	284.0
200 POST /card/436247292/3ds	10	396	334	456	36.21	0.00%	2.4/sec	0.67	3.08	284.0
200 PATCH /card/436247292/status	10	348	300	395	35.10	0.00%	2.6/sec	0.73	3.35	284.0
200 GET /card/436247292/transaction	10	501	424	567	40.17	0.00%	2.5/sec	10.13	3.05	4200.3
200 GET /card/436247292/pin	10	445	342	599	73.58	0.00%	2.5/sec	0.79	3.12	320.8
200 GET /card/436247292	10	413	338	497	55.43	0.00%	2.5/sec	1.01	3.05	415.1
TOTAL	400	429	120	2982	275.25	10.00%	21.8/sec	14.45	27.56	679.8

Figure 5.52: Average response time for a ten users

Label ↓	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
404 DELETE /card/202934118/3ds	1	4131	4131	4131	0.00	0.00%	14.5/min	0.08	0.31	345.0
404 DELETE /card/194652401/3ds	1	4219	4219	4219	0.00	0.00%	14.2/min	0.08	0.30	346.0
404 DELETE /card/172347772/3ds	1	4177	4177	4177	0.00	0.00%	14.4/min	0.08	0.06	356.0
404 DELETE /card/165676657/3ds	1	4182	4182	4182	0.00	0.00%	14.3/min	0.08	0.06	343.0
404 DELETE /card/162987359/3ds	1	4245	4245	4245	0.00	0.00%	14.1/min	0.08	0.06	341.0
404 DELETE /card/132092273/3ds	1	4489	4489	4489	0.00	0.00%	13.4/min	0.07	0.06	341.0
404 DELETE /card/120092888/3ds	1	4216	4216	4216	0.00	0.00%	14.2/min	0.08	0.30	346.0
404 DELETE /card/114038600/3ds	1	4252	4252	4252	0.00	0.00%	14.1/min	0.08	0.30	346.0
404 DELETE /card/112405147/3ds	1	4236	4236	4236	0.00	0.00%	14.2/min	0.08	0.06	346.0
404 DELETE /card/111845175/3ds	1	4574	4574	4574	0.00	0.00%	13.1/min	0.07	0.28	343.0
404 DELETE /card/107816950/3ds	1	4507	4507	4507	0.00	0.00%	13.3/min	0.08	0.05	357.0
404 DELETE /card/106420403/3ds	1	4442	4442	4442	0.00	0.00%	13.5/min	0.08	0.06	342.0
404 DELETE /card/104848348/3ds	1	4114	4114	4114	0.00	0.00%	14.6/min	0.08	0.31	357.0
401 PUT /card/436247292/pin	100	3736	448	4671	892.45	0.00%	1.5/sec	0.46	1.33	327.0
401 POST card/436247292/pin/send	100	2976	1916	4625	826.88	0.00%	1.1/sec	0.34	0.97	327.0
401 POST /card/436247292/transfer-balance	100	3567	2693	4647	564.71	0.00%	1.0/sec	0.32	0.98	322.0
401 POST /card/436247292/activate	100	3070	234	4793	1284.78	0.00%	2.4/sec	0.74	2.11	322.0
401 POST /card/436247292/3ds	100	4062	1842	4812	424.72	0.00%	1.1/sec	0.33	0.94	322.0
401 PATCH /card/436247292/status	100	3933	873	4268	670.95	0.00%	1.2/sec	0.38	1.08	322.0
401 PATCH /card/436247292/distributor	100	3912	3281	4670	429.79	0.00%	58.6/min	0.31	0.87	322.0
401 GET /card/436247292/transaction	100	4413	4025	4867	186.49	100.00%	57.7/min	3.58	0.81	3805.7
401 GET /card/436247292/pin	100	3564	370	4898	1084.28	0.00%	1.7/sec	0.54	1.45	322.0
401 GET /card/436247292	100	4175	3576	4538	124.41	0.00%	58.6/min	0.31	0.81	322.0
401 DELETE /card/436247292/3ds	100	4145	3454	4355	123.76	0.00%	58.9/min	0.31	0.84	322.0
400 POST /card/436247292/activate	100	2923	282	4933	1298.52	0.00%	2.6/sec	0.99	2.33	389.8
400 PATCH /card/436247292/status	100	3945	800	4985	758.42	0.00%	1.3/sec	0.45	1.12	365.0
204 PATCH /card/436247292/distributor	100	4054	3264	4625	312.66	100.00%	58.2/min	0.27	0.85	284.0
204 DELETE /card/436247292/3ds	100	4117	2624	4333	221.07	100.00%	59.8/min	0.28	0.84	284.0
200 PUT /card/436247292/pin	100	3727	427	4924	989.10	0.00%	1.5/sec	0.43	1.37	289.0
200 POST card/436247292/pin/send	100	3198	2056	4641	737.72	0.00%	1.0/sec	0.30	0.93	289.0
200 POST /card/436247292/transfer-balance	100	3669	2959	4629	542.00	0.00%	59.8/min	0.28	0.94	284.0
200 POST /card/436247292/activate	100	2890	257	4920	1465.90	0.00%	2.9/sec	0.80	2.51	284.0
200 POST /card/436247292/3ds	100	4065	1606	4571	481.05	0.00%	1.1/sec	0.30	0.95	284.0
200 PATCH /card/436247292/status	100	3824	787	4951	767.41	0.00%	1.3/sec	0.37	1.14	284.0
200 GET /card/436247292/transaction	100	4389	4084	4810	194.22	0.00%	57.8/min	3.73	0.80	3971.6
200 GET /card/436247292/pin	100	3493	281	4893	1147.29	0.00%	1.9/sec	0.58	1.53	319.2
200 GET /card/436247292	100	4215	3759	4522	130.93	0.00%	58.4/min	0.39	0.79	414.9
TOTAL	4000	4110	225	31192	2889.11	11.00%	22.4/sec	15.58	19.42	712.9

Figure 5.53: Average response time for a hundred users

No of users	Average response time (ms)	Total no. of requests
1	138	40
10	429	400
100	4110	4000

Table 5.6: Performance results summary

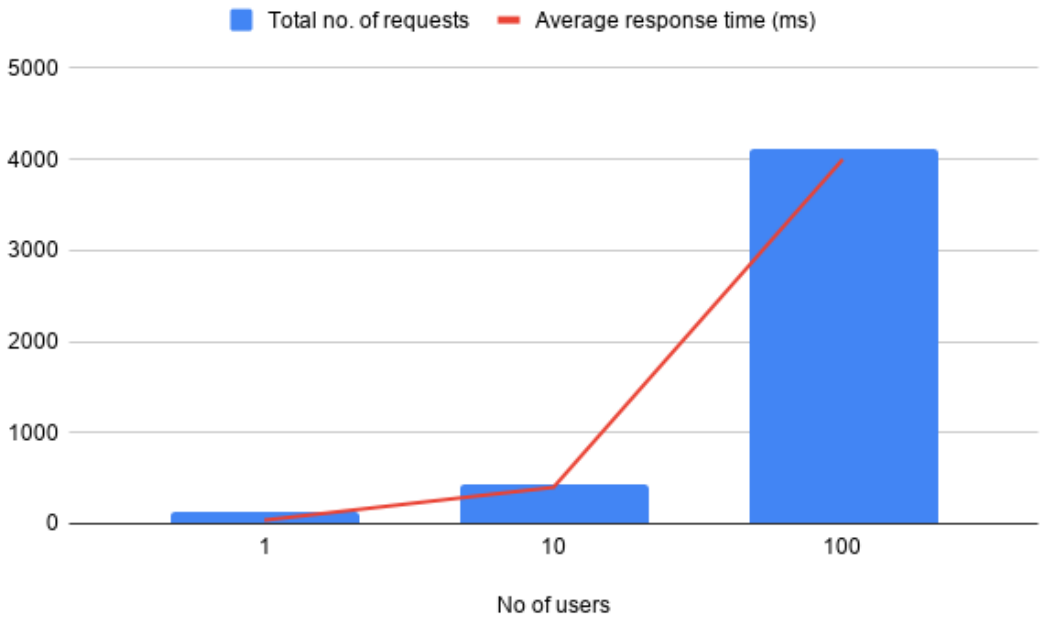


Figure 5.54: Performance results summary graph

In a summary, the CTFM was able to serve a single user with 40 unique requests with an average response time of 138ms, for 10 users and 400 unique requests with an average response time of 429ms, for 100 and 4000 unique requests with an average response time of 4.1s.

CHAPTER 6 - CONCLUSION

Microservices architecture has gained a significant amount of recognition during recent years, which has been evolving and has been widely adopted by the software industry. One of the main reasons for the adaption is to fulfil the rapidly changing and demanding business requirements. The industry is gradually moving away from Monolithic architecture [1]. Unlike the monoliths, because of the architecture complexity and autonomy of the microservices [34], it has introduced challenges in integration testing [50]. Maintaining a remote resource-intensive development environment with microservices would be labour-intensive to maintain and not cost-effective due to the higher frequency of deployments with shorter development lifecycles and service changes. Also running the necessary services in a local development environment would be nearly impossible due to the lack of resources, unavailability of services and complexities of initializing and maintaining the services.

Currently, available tools such as WireMock[20], Hooverfly[21], or PACT[7] has many limitations and doesn't provide a smooth experience to the developers, all of them having learning curves and dependencies in the code base such as packages or libraries and except for PACT other tools has limited support for the programming languages or scripting languages and also licensing is required to get the complete access.

6.1 Research contribution

The main research objective is to provide a solution for the integration tests in REST-based microservices, which should overcome the limitations of the currently available tools. The inexpensive solution should be easy to maintain and should evolve with the services and should not require any extra effort to initialize or maintain. The solution also should be able to run in a local or remote development environment with limited resource requirements.
The research has been conducted to find a reliable solution to overcome the identified problems and challenges, to improve the development productivity, and smooth integration testing solution. The authors have identified the potential service virtualization solution based on the widely adopted Open API specifications. Almost all the currently available programming or scripting languages encourages the use of OAS when it comes to providing API interfaces, which has become a standard. The authors have followed the proposed methodology and implemented the POC that would mitigate the identified problems and limitations with the existing tools.

The POC framework is evaluated under chapter five for the implemented scope. The evaluations have proved the solution is a success and were able to successfully emulate producer microservices by virtualizing the services with its Open API specification. The solution improves the productivity of the developers which requires hardly any configurations or dependencies as packages or libraries on the consumer or producer code base. Easy to re-produce edge cases and failures which is difficult to reproduce in real services. The solution significantly reduces the integration time and effort which directly reduces the development time.

6.2 Research limitations

Authors have successfully managed to implement and evaluate the CTFM and it has proven to satisfy the expectations and requirements of the proposed solution. Under these stages, the following limitations are identified.

The CTFM only supports the REST API based microservices, which adopts Open API specifications and only support Content-Type “application/json”. The framework allows extending its functionality which is based on Laravel 8. Users willing to extend the functionality must have knowledge of PHP and the Laravel framework and the framework design patterns.

The service virtualization and behaviour solely based on the respective producers’ Open API specification, The OAS definition for each service should be aligned and up to date with the actual behaviour and the potential responses of the respective producer service. Usually, the OAS specification is defined by the annotations in the codebase which should be kept up to date and should evolve with the service changes and additions, typically the OAS specification is verified by the review cycles which involves human interactions. If the reviewer fails to identify the anomaly in the OAS, there could be scenarios that OAS definition might not represent the complete actual service behaviours or document with incorrect or missing schema definitions. Such scenarios only can be identified after the deployment stage and be rectified with another development iteration of the respective service, since the integrators depend on the OAS and refer to it as the API documentation. The CTFM unable to detect such scenarios ahead but will help to identify these anomalies in the service integrations so the integrators can report to the maintainers of the producer service for the verifications.

Another limitation would be unable to automatically produce the behaviours which involve database interactions, such as a resource that exists in the database validations, or a resource not found responses or authentication or authorization error responses. This is mainly due to the CTFM doesn’t contain any business logic to han-

dle such situations. By default, the framework selects and automatically generate the response for the first response in the service schema. Most of the time which falls under the HTTP status 2xx range. But such edge cases necessary for the consumers to test their integrations to handle the failures. To facilitate that, the framework has provided a way to produce such scenarios which are described under Chapter 4, the CTFM will verify the expected response schema availability in the provided OAS if so it'll generate a response accordingly or else it would complain that such expectation will not be produced by the producer service.

6.3 **Future work**

The implemented solution was a POC for the research objective, which only supports a single version of the OAS for a specific producer, as a future enhancement, it'll support multiple version of a producers' OAS if any, which helps to verify the integration changes of the older versions of the producers' API where necessary for the consumer services.

In the current implementation, the OAS has to be manually configured in the registration of a producer service. As a future enhancement, to automatically download and kept up to date, the available OAS on each release cycle of the respective configured producer. This is possible on the CI/CD of the producer release cycles, where the specification is published and available online for reference. Finally, extend the solution for other media types such as XML and GraphQL

REFERENCES

1 G. Mazlami, J. Cito and P. Leitner, "Extraction of Microservices from Monolithic Software Architectures - IEEE Conference Publication", Ieeexplore.ieee.org, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/8029803/.

2 P. Di Francesco, I. Malavolta and P. Lago, "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption - IEEE Conference Publication", Ieeexplore.ieee.org, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7930195/.

3 Z. Bloom, "How We Deploy 300 Times a Day", product.hubspot.com, 2017. [Online]. Available: http://product.hubspot.com/blog/how-we-deploy-300-times-a-day.

4 "Microservices Pattern: Monolithic Architecture pattern", microservices.io, 2017. [Online]. Available: http://microservices.io/patterns/monolithic.html.

5 V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. Reiter and V. Sekar, "Gremlin: Systematic Resilience Testing of Microservices - IEEE Conference Publication", Ieeexplore.ieee.org, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7536505/.

6 M. Fowler and T. Clemson, "Testing Strategies in a Microservice Architecture", martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/articles/microservice-testing.

7 "Pact Foundation", GitHub, 2017. [Online]. Available: https://github.com/pact-foundation.

8 S. Neuman, Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, February 2015.

9 J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, 1st ed. Addison-Wesley Professional, July 2010.

10 P. M. Duvall, S. Matyas, and A. Glover, Continuous Integration:Improving Software Quality and Reducing Risk, 1st ed. AddisonWesley Professional, June 2007.

11 J. Fields, Working effectively with unit tests, 1st ed. CreateSpace Independent Publishing Platform, 2014.

12 M. Noback, Microservices for everyone, 1st ed. Leanpub, 2017, pp. 48-71, 75-87, 119-141.

13 I. Nadareishvili, R. Mitra, M. McLarty and M. Amundsen, *Microservice Architecture Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly, 2016, pp. 5-8, 13-20, 103-110. [Design architecture]

14 J. Bonér, Reactive Microservices Architecture Design Principles for Distributed Systems, 1st ed. O'Reilly, 2016, pp. 8-22, 32-38.

15 D. Taibi, V. Lenarduzzi and C. Pahl, "Architectural Patterns for Microservices: A Systematic Mapping Study.", in 8th International Conference on Cloud Computing and Services Science, Funchal, Madeira, Portugal, 2018, pp. 221-227.

16 M. Fowler, "bliki: TestDouble", martinfowler.com, 2017. [Online]. Available: https://martinfowler.com/bliki/TestDouble.html.

17 G. Meszaros, XUnit test patterns, 1st ed. Upper Saddle River, NJ: Addison-Wesley, 2012, pp. 256-341. For test doubles

18 Y. Park and S. Sharma, "Providing Service Using A Virtualization Infrastructure", Journal of Service Science (JSS), vol. 2, no. 2, pp. 17-22, 2009. Available: 10.19030/jss.v2i2.4283.

19 S. Upadhyay and A. Acharya, "Testing Service-Oriented Applications using Service Virtualization", pp. 1-3, 2015.

20 "WireMock",*WireMock*, 2018. [Online]. Available: http://wiremock.org/.

21 "What is Hoverfly? — Hoverfly v1.3.1 documentation",*Docs.hoverfly.io*, 2018. [Online]. Available: https://docs.hoverfly.io.

22 Day, P., 2014. n-Tiered Test Automation Architecture for Agile Software Systems. *Procedia Computer Science*, 28, pp.332-339.

23 F. Vera-Rivera, "A development process of enterprise applications with microservices", Journal of Physics: Conference Series, vol. 1126, p. 012017, 2018. Available: 10.1088/1742-6596/1126/1/012017.

24 C. Fetzer, "Building Critical Applications Using Microservices", *IEEE Security & Privacy*, vol. 14, no. 6, pp. 86-89, 2016. Available: 10.1109/msp.2016.129.

25 Taibi, Davide & Lenarduzzi, Valentina & Pahl, Claus & Janes, Andrea. (2017). Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. 1-5. 10.1145/3120459.3120483.

26 A. Sundar, "AN INSIGHT INTO MICROSERVICES TESTING STRATEGIES", *Infosys.com*, 2018. [Online]. Available: https://www.infosys.com/services/it-services/validation-solution/white-papers/documents/microservices-testing-strategies.pdf.

27 Soldani, Jacopo & Tamburri, Damian & Heuvel, Willem-Jan. (2018). The Pains and Gains of Microservices: A Systematic Grey Literature Review. Journal of Systems and Software. 146. 10.1016/j.jss.2018.09.082.

28 A. Brogi, D. Neri and J. Soldani, "A microservice-based architecture for (customisable) analyses of Docker images", *Software: Practice and Experience*, vol. 48, no. 8, pp. 1461-1474, 2018. Available: 10.1002/spe.2583.

29 LI, Y., LI, Y. and WANG, D., 2018. Design of Reading Platform Based on Microservice Architecture. *DEStech Transactions on Computer Science and Engineering*, (iece).

30 Kholy, M. and Fatatry, A., 2019. Framework for Interaction Between Databases and Microservice Architecture. *IT Professional*, 21(5), pp.57-63.

31 P. Vincent, "Integration tests don't cut it for microservices. Here's what does. | TechBeacon", *TechBeacon*, 2019. [Online]. Availa-

ble:https://techbeacon.com/app-dev-testing/integration-tests-dont-cut-it-microservices-heres-what-does.

32 Hu, Y., de Laat, C. and Zhao, Z., 2019. Optimizing Service Placement for Microservice Architecture in Clouds. *Applied Sciences*, 9(21), p.4663.

33 D. Rajput and Rajesh R V, *Building microservices with Spring*, 1st ed. Packt, 2016, pp. 371-380.

34 GOU, L., CHEN, Q., LIANG, J. and LIAO, X., 2019. Technical Research and Application Analysis of Microservice Architecture. *DEStech Transactions on Computer Science and Engineering*, (iccis).

35 Barabanov, A. and Makrushin, D., 2020. Authentication and Authorization in Microservice-Based Systems: Survey of Architecture Patterns. *Voprosy kiberbezopasnosti*, (4(38), pp.32-43.

36 N. Raychev, "Test automation in microservice architecture", *Ieeesem.com*, 2020. [Online]. Available: http://www.ieeesem.com/researchpaper/Test_automation_in_microservice_architecture.pdf.

37 R. Stein, "Message Pact - Contract testing in event-driven applications", *codecentric AG Blog*, 2020. [Online]. Available: https://blog.codecentric.de/en/2019/11/message-pact-contract-testing-in-event-driven-applications.

38 F. Rosner, "Implementing a Pact workflow with Pact Broker and Gitlab CI", *codecentric AG Blog*, 2020. [Online]. Available: https://blog.codecentric.de/en/2020/02/implementing-a-consumer-driven-contract-testing-workflow-with-pact-broker-and-gitlab-ci

39 "Home - OpenAPI Initiative", *OpenAPI Initiative*, 2017. [Online]. Available: https://www.openapis.org/.

40 T. Otwell, "Validation - Laravel - The PHP Framework For Web Artisans", *Laravel.com*, 2021. [Online]. Available: https://laravel.com/docs/8.x/validation.

41 N. El Ioini, "S-Test: A Framework for Services Testing", *https://www.researchgate.net/*, 2015. [Online]. Available: https://www.researchgate.net/publication/300337281_S-Test_A_Framework_for_Services_Testing.

42 X. Bai, S. Lee, W. Tsai and Y. Chen, "Ontology-Based Test Modeling and Partition Testing of Web Services", *https://www.researchgate.net*, 2008. [Online]. Available: https://www.researchgate.net/publication/221587173_Ontology-Based_Test_Modeling_and_Partition_Testing_of_Web_Services.

43 Z. Zheng and M. Lyu, "Optimal Fault Tolerance Strategy Selection for Web Services", *International Journal of Web Services Research*, vol. 7, no. 4, pp. 21-40, 2010.

44 N. Dharmaji, "A Study of Containerization as a Micro service Deployment Model", *International Journal for Research in Applied Science and Engineering Technology*, vol. 8, no. 5, pp. 1365-1367, 2020.

45 C. K., "A Systematic Study of Micro Service Architecture Evolution and their Deployment Patterns", *International Journal of Computer Applications*, vol. 182, no. 29, pp. 18-24, 2018.

46 A. Balalaie, A. Heydarnoori, P. Jamshidi, D. Tamburri and T. Lynn, "Microservices migration patterns", *Software: Practice and Experience*, 2018.

47 M. Bozkurt, M. Harman and Y. Hassoun, "Testing and verification in service-oriented architecture: a survey", *Software Testing, Verification and Reliability*, vol. 23, no. 4, pp. 261-313, 2012.

48 G. Beydoun, D. Xu and V. Sugumaran, "Service Oriented Architectures (SOA) Adoption Challenges", *International Journal of Intelligent Information Technologies*, vol. 9, no. 2, pp. 1-6, 2013.

49 R. Evans, N. Alshuqayran and N. Ali, "Towards Micro Service Architecture Recovery: An Empirical Study", *https://www.researchgate.net*, 2018. [Online]. Available: https://www.researchgate.net/publication/324503215_Towards_Micro_Service_Architecture_Recovery_An_Empirical_Study.

50 R. O'Connor, P. Elger and P. Clarke, "Continuous software engineering-A microservices architecture perspective", *Journal of Software: Evolution and Process*, vol. 29, no. 11, p. e1866, 2017.

APPENDIX

Appendix I: Framework packages and libraries

```
{
  "name": "ctfm/framework",
  "type": "project",
  "description": "Microservice integration testing framework",
  "keywords": [
     "framework",
     "CTFM"
  ],
  "license": "MIT",
  "require": {
     "php": "^7.4",
     "fideloper/proxy": "^4.4",
     "fruitcake/laravel-cors": "^2.0",
     "guzzlehttp/guzzle": "^7.0.1",
     "jenssegers/mongodb": "^3.8",
     "laravel/framework": "^8.12",
     "laravel/tinker": "^2.5",
     "laravel/ui": "^3.2"
  },
  "require-dev": {
     "facade/ignition": "^2.5",
     "fakerphp/faker": "^1.9.1",
     "laravel/sail": "^1.0.1",
     "mockery/mockery": "^1.4.2",
     "nunomaduro/collision": "^5.0",
     "phpunit/phpunit": "^9.3.3"
  },
  "config": {
     "optimize-autoloader": true,
     "preferred-install": "dist",
```

```
"autoload-dev": {
    "psr-4": {
        "Tests\\": "tests/"
    }
},
"minimum-stability": "dev",
"prefer-stable": true,
"scripts": {
    "post-autoload-dump": [
        "Illuminate\\Foundation\\ComposerScripts::postAutoloadDump",
        "@php artisan package:discover --ansi"
    ],
    "post-root-package-install": [
        "@php -r \"file_exists('.env') || copy('.env.example', '.env');\""
```

Figure 6.55 composer.json

```
{
  "private": true,
  "scripts": {
    "dev": "npm run development",
    "development": "mix",
    "watch": "mix watch",
    "watch-poll": "mix watch -- --watch-options-poll=1000",
    "hot": "mix watch --hot",
    "prod": "npm run production",
    "production": "mix --production"
  },
  "devDependencies": {
    "@babel/preset-react": "^7.0.0",
    "@date-io/date-fns": "1.x",
    "@material-ui/core": "^4.9.0",
    "@material-ui/icons": "^4.5.1",
    "@material-ui/lab": "^4.0.0-alpha.47",
    "@material-ui/pickers": "^3.2.10",
    "@pmmmwh/react-refresh-webpack-plugin": "^0.4.3",
    "@reduxjs/toolkit": "^1.4.0",
    "@testing-library/jest-dom": "^4.2.4",
    "@testing-library/react": "^9.3.2",
    "@testing-library/user-event": "^7.1.2",
    "axios": "^0.21",
    "bootstrap": "^4.0.0",
    "browser-sync": "^2.26.14",
    "browser-sync-webpack-plugin": "^2.2.2",
    "cross-env": "^7.0.3",
    "date-fns": "^2.12.0",
    "eslint-friendly-formatter": "^4.0.1",
    "google-maps-react": "^2.0.2",
    "highcharts": "^8.0.0",
    "highcharts-react-official": "^2.2.2",
    "jquery": "^3.2",
    "laravel-mix": "^6.0.6",
    "lodash": "^4.17.19",
    "moment": "^2.24.0",
    "popper.js": "^1.12",
    "postcss": "^8.1.14",
    "react": "^16.2.0",
    "react-dom": "^16.2.0"
```

```
"dependencies": {
    "@material-ui/core": "^4.11.3",
    "@material-ui/icons": "^4.11.2",
    "material-ui-dropzone": "^3.5.0",
    "prettier": "2.1.1"
},
"eslintConfig": {
    "extends": "react-app"
},
"browserslist": {
    "production": [
        ">0.2%",
        "not dead",
        "not op_mini all"
    ],
    "development": [
        "last 1 chrome version",
        "last 1 firefox version",
        "last 1 safari version"
    ]
```

Figure 6.56 Package.json

Appendix II: CTFM Docker configuration

```
FROM php:7.4.16-fpm


RUN apt update -y
RUN apt install -y software-properties-common


# Install system dependencies
RUN apt update \
  && DEBIAN_FRONTEND=noninteractive apt-get install -yqq --fix-missing \
  apt-utils \
  git \
  curl \
  libpng-dev \
  libonig-dev \
  libxml2-dev \
  default-mysql-client \
  zlib1g-dev \
  libzip-dev \
  zip \
  unzip \
  nano \
  && apt-get clean \
  && rm -rf /var/lib/apt/lists/*


# Install PHP extensions
RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd soap \
xml iconv simplexml zip posix


# Install pecl packages
RUN pecl install xdebug \
  && docker-php-ext-enable xdebug


#install and enable mongo
RUN apt update -y
RUN apt install -y libcurl4-openssl-dev pkg-config libssl-dev --fix-missing \
&& pecl install mongodb && docker-php-ext-enable mongodb
```

Figure 6.57 Dockerfile

```
version: "3.5"
services:
 mongodb:
   container_name: ctfm-mongodb
   image: mongo:4.4.4
   restart: unless-stopped
   environment:
     MONGO_INITDB_DATABASE : "${MONGO_DB_DATABASE}"
     MONGO_INITDB_ROOT_USERNAME: "${MONGO_DB_ADMIN_USERNAME}"
     MONGO_INITDB_ROOT_PASSWORD: "${MONGO_DB_ADMIN_PASSWORD}"
   volumes:
     - ctfm-storage:/data/db
   deploy:
     replicas: 1
   ports:
     - "27023:27017"

 ctfm-app:
   container_name: ctfm-app
   restart: unless-stopped
   build:
     context: .
     dockerfile: ./etc/docker/app/Dockerfile
   volumes:
     - ./etc/docker/app/php/php.ini:/usr/local/etc/php/conf.d/php.ini
     - ./etc/docker/app/php/xdebug.ini:/usr/local/etc/php/conf.d/xdebug.ini
     - ./:/var/www/app
   depends_on:
     - mongodb
   environment:
     MONGO_DB_PORT: "${MONGO_DB_PORT}"
     MONGO_DB_HOST: mongodb
     MONGO_DB_DATABASE: "${MONGO_DB_DATABASE}"
     MONGO_DB_USERNAME: "${MONGO_DB_ADMIN_USERNAME}"
     MONGO_DB_PASSWORD: "${MONGO_DB_ADMIN_PASSWORD}"
```

Figure 6.58 Docker compose file

```
USERID=$(shell id -u)
GROUPID=$(shell id -g)

DOCKER_COMPOSE=docker-compose -f docker-compose.yml
IMAGE_NAME:=ctfm
PHP_SERVICE:=ctfm-app
DOMAIN_NAME:=ctfm.test

.PHONY: help
help: ## Shows all available commands with their description

  $(info Available Commands:)
  @cat $(MAKEFILE_LIST) | grep -e "^[a-zA-Z_\-]*: *.*## *" | awk 'BEGIN {FS = ":.*?## "}; {printf "\033[36m%-30s\033[0m %s\n", $$1, $$2}'

.PHONY: ps
ps: ## List containers

  $(DOCKER_COMPOSE) ps

.PHONY: build
build: ## Builds the docker images and executes the vendors
  $(DOCKER_COMPOSE) build
  $(DOCKER_COMPOSE) run --rm -u $(USERID):$(GROUPID) $(PHP_SERVICE) composer install

.PHONY: up
up: ## Starts, and attaches to containers for a service

  $(DOCKER_COMPOSE) up -d
  @$(MAKE) -s ps

.PHONY: start
start: ## Start stoped containers

  $(DOCKER_COMPOSE) start
  @$(MAKE) -s ps
```

Figure 6.59 Make file