DEPARTAMENTO DE ENGENHARIA
INFORMÁTICA

gRPC for service-to-service communication

Instituto Superior de Engenharia do Porto

2022 / 2023

1170640 Ricardo da Costa Aguiar Mourisco

ISEP INSTITUTO SUPERIOR
DE ENGENHARIA DO PORTO

gRPC for service-to-service communication

Instituto Superior de Engenharia do Porto

2022 / 2023

1170640 Ricardo da Costa Aguiar Mourisco

DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Degree in Informatics Engineering

July 2023

ISEP Advisor: **Isabel de Fátima Silva Azevedo**

To my parents and sister, for supporting me through highs and lows.

Acknowledgements

- To Dr Isabel de Fátima Silva Azevedo for coordinating my internship, guiding me through the complications that arise, and in general, putting up with me.
- To Dr Berta Baptista, for all the support you've provided me over the last three years.
- To my close family for being an important anchor over all these years and being the best support structure a person can ask for.
- To my university friends, namely António Dias, António Guerra, Afonso Machado and Hugo Nogueira, for making this university experience much more amusing.
- To my friends Afonso Sousa, Ernesto Romano, João Santos and Hugo Cunha, for helping me in many ways, even if for, sometimes, just distracting me.
- To Jorge, for being one of the most important people in the last decade and one of the reasons I've reached this career point.

Abstract

With the continuous rise of the digital world, more and more devices are connected and communicating without an end user even knowing. This ever-increasing number of communications requires us to look for more efficient protocols to increase throughput and efficiency. As of now, the de facto standard is REST using the HTTP protocol, but there have been alternatives that promise to offer a better experience and performance, one of them being gRPC, a communication protocol developed by Google and based on their internal communication protocol, known as Stubby.

This report documents the development journey of developing a prototype backend API for a used book store based on microservice architecture and implementing HTTP REST and gRPC for internal communication. With this project, we looked to assert if gRPC is a worthwhile replacement for HTTP REST when it comes to service-to-service communication, not only in terms of performance but also the maintainability and learning curve of the protocol.

With this project, we've established that the gRPC programming experience can be efficient, and the framework can bring significant performance benefits compared to a traditional REST approach.

Keywords (Theme): Comunication, Microservices, Books, Backend

Keywords (Technologies): gRPC, REST, HTTP, Protobuf

Index

1 Introduction 1

1.1 Contextualization 1

1.2 Problem Description 1

1.2.1 Objectives 1

1.2.2 Contributions 2

1.2.3 Approach and Work Plan 2

2 State of the Art 3

2.1 Structured Data Serialization 3

2.1.1 XML 3

2.1.2 JSON 4

2.1.3 Protobuf 5

2.1.4 FlatBuffers 6

2.2 HTTP 6

2.2.1 Version 1.1 7

2.2.2 Version 2.0 7

2.3 REST 8

2.4 gRPC 9

2.4.1 What is gRPC? 9

2.4.2 Why gRPC? 11

2.4.3 Industry Implementation Examples 12

3 Analysis and Design 13

3.1 Analysis 13

3.1.1 Domain of the Problem 13

3.1.2 Functional Requirements 14

3.2 Design 14

3.2.1 Architectural Drivers 15

3.2.2 Design Concepts 16

3.2.3 Design Alternatives 18

3.2.4 Detailed Solution Design 21

4 Implementation 28

4.1 Implementation Description 28

4.1.1 Microservices 30

4.1.2 Composer 34

4.2 Tests 40

4.3 Solution Evaluation 43

4.3.1 Maintainability 43

4.3.2 Performance 44

5 Conclusion........ 46

5.1 Achievements 46

5.2 Threats to Validity 46

5.3 Future Work 47

5.4 Personal Appreciation 47

References........ 48

Figure Index

Figure 1: Representation of Unary RPCs (Simon Aronsson, 2020).. 10

Figure 2: Representation of Server Streaming RPCs (Simon Aronsson, 2020)........................... 10

Figure 3: Representation of Client Streaming RPCs (Simon Aronsson, 2020)............................ 11

Figure 4: Representation of Bidirectional Streaming RPCs (Simon Aronsson, 2020) 11

Figure 5: Class structure of APIAuthors (Ricardo Mourisco, 2023b).. 30

Figure 6: Class structure of Composer (Ricardo Mourisco, 2023a).. 34

Figure 7: Unit test success for the domain of APIAuthors (Ricardo Mourisco, 2023b) 40

Figure 8: Inter-layer integration tests between controller, service and repository of APIAuthors (Ricardo Mourisco, 2023b)... 40

Figure 9: Postman test steps and success (Ricardo Mourisco, 2023b) 41

Figure 10: jMeter testing setup.. 44

Figure 11: Response Times: gRPC vs HTTP REST 45

Table Index

Table 1: Software stacks used on each service 28

Table 2: Maintainability metrics for the developed software 43

Table 3: Response Times: gRPC vs HTTP REST 44

Diagram Index

Diagram 1: Domain model 13

Diagram 2: Domain model with aggregates 16

Diagram 3: Domain Model with aggregates and microservices defined 16

Diagram 4: System alternative design 1 18

Diagram 5: System alternative design 2 19

Diagram 6: System alternative design 3 20

Diagram 7: Level 1 logic diagram 21

Diagram 8: Level 2 logic diagram 22

Diagram 9: Level 3 logic diagram for the APIAuthors component 23

Diagram 10: Level 3 Logic diagram for the Composer component 24

Diagram 11: Level 2 Process Diagram of "List exemplars sold by client" 25

Diagram 12: Level 3 Process Diagram of "List exemplars sold by client" on the APIAuthors component 26

Diagram 13: Level 3 Process Diagram of "List exemplars sold by client" on the Composer component 27

Code Excerpt Index

Code Excerpt 1: Excerpt of an XML-formatted configuration file 4

Code Excerpt 2: Excerpt of a JSON-formatted response 5

Code Excerpt 3: Excerpt of a protocol buffer serialization scheme configuration 6

Code Excerpt 4: Except of the gRPC Controller implementation in APIAuthors (Ricardo Mourisco, 2023b) 31

Code Excerpt 5: Excerpt of the APIAuthors HTTP REST Controller implementation in C# (Ricardo Mourisco, 2023b) 31

Code Excerpt 6: Excerpt of the APIBooks HTTP REST Controller implementation in Java (Ricardo Mourisco, 2023c)... 32

Code Excerpt 7: Authors Service Interface in APIAuthors (Ricardo Mourisco, 2023b)............... 32

Code Excerpt 8: Authors Service Implementation in APIAuthors (Ricardo Mourisco, 2023b) ... 33

Code Excerpt 9: Author Entity ORM configuration class (Ricardo Mourisco, 2023b)................. 33

Code Excerpt 10: Book object that aggregates the information (Ricardo Mourisco, 2023a)..... 35

Code Excerpt 11: Authors Fetching Service Interface in the Composer (Ricardo Mourisco, 2023a) 35

Code Excerpt 12: Excerpt of Author Fetching Services gRPC implementation in the composer (Ricardo Mourisco, 2023a) 36

Code Excerpt 13: Excerpt of Author Fetching Services HTTP REST implementation in the composer (Ricardo Mourisco, 2023a) 37

Code Excerpt 14: Excerpt of the composer's appsettings.json configuration file (Ricardo Mourisco, 2023a) 37

Code Excerpt 15: Excerpt of bootstrap code demonstrating configuration functionality (Ricardo Mourisco, 2023a) 38

Code Excerpt 16: Excerpt of Exemplar Composer Service in the composer (Ricardo Mourisco, 2023a) 39

Code Excerpt 17: k6 test script for APICategories gRPC endpoint (Ricardo Mourisco, 2023b) .. 42

Abbreviations List

ADD	Attribute-Driven Design
API	Application Programming Interface
gRPC	g[1] Remote Procedure Call
HTTP	Hypertext Markup Language
IDL	Interface Definition Language
JSON	JavaScript Object Notation
ORM	Object-Relational Mapping
Protobuf	Protocol Buffers
PUB/SUB	Publish–Subscribe Pattern
REST	REpresentational State Transfer
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SSE	Server-Sent Events
XML	Extensible Markup Language

[1] The letter ‘g’ in ‘gRPC’ has a different meaning every version (for example, in version 1.56, 'g' stands for 'galvanized').

1 Introduction

1.1 Contextualization

With an increasingly interconnected world, where, each day, hundreds of new devices and systems come online, there's a significant increase in communication between these. Adding to this increase, the philosophy shift on the software design front of moving from monolithic service development to a microservices architecture development increased these communications, particularly the internal service-to-service kind.

In past years, the standard used to handle these types of communication has been the REST (REpresentational State Transfer) architecture, utilizing the HTTP (Hypertext Markup Language) protocol. However, particularly for internal communication, there has been a search for a faster alternative, with more capacity and less overhead.

With this interest in mind, frameworks, such as gRPC (g Remote Procedure Call), were created, which use HTTP/2 as its communication protocol and Protobuf (Protocol Buffers) as its main IDL (Interface Definition Language).

1.2 Problem Description

In this project, we're looking to assert if, in an internal system with microservices communicating with each other, gRPC could deliver a significant performance advantage over the current standard of REST over HTTP, particularly when it comes to a high volume of traffic and particularly, continuous bidirectional traffic while maintaining reliability.

1.2.1 Objectives

This project has these two main objectives:

- Explore the use of gRPC for service-to-service communication in an application to be developed. An API (Application Programming Interface) gateway will translate REST HTTP requests into internal gRPC calls.
- Compare the performance of gRPC and REST.

The report will comprehensively reflect the findings, and the code will be open-sourced.

1.2.2 Contributions

This project's objective is to demonstrate the benefits of the gRPC technology, even on small-scale projects, in terms of performance and efficiency, as well as spread an approach to converting RESTful services into gRPC services seamlessly.

Ultimately, with the increase of microservices in the industry at large, spreading awareness of a more efficient method of communication will help reduce costs and resource consumption, which will help our environment.

1.2.3 Approach and Work Plan

The first step of this project was to research the technologies at play, ascertain their strengths and weaknesses and assess which to use.

Following that research, I formulated an application with multiple services related to book management to meet the needs of a used book store with business-related features, such as creating and listing books and their descriptions, publishers, authors, and exemplars with some multilingual support. An architecture based on microservices will be adopted. The application should support gRPC and REST for communication between services, and it must be possible to configure one option or another, at least for some services.

All work will be available in a public repository as a contribution to the dissemination of the protocol, its characteristics, and potential difficulties and problems that may condition its adoption. In addition, a report detailing the process will also be formulated to serve as a guide for future projects.

2 State of the Art

To be able to design and build the desired software, I must first look at the state of the art of technology involved in this work, specifically, the various models of structured data serialization and communication protocols, along with research of related works and the prominent solutions used in the industry.

As mentioned in the introduction, the most established method of communication between services is REST, mainly using the HTTP protocol, which sees the highest production usage. REST, unlike most technologies referred to in this report, is not strictly defined, but a set of guidelines.

Before REST, and still enjoying mainstream use in the ecosystem, there was SOAP (Simple Object Access Protocol), a messaging protocol using the XML (Extensible Markup Language) file format to exchange messages, mainly using HTTP as its transport protocol, although, notably, it is not dependent on such.

Finally, we have gRPC, an RPC (Remote Procedure Call) framework that communicates with Protobuf as its IDL, although it can be used with other file formats, using the HTTP/2 communication protocol, and boasts high performance and scalability for all kinds of projects.

2.1 Structured Data Serialization

To communicate data between services, we employ various serialization methods to send the information efficiently and effectively, of which I would like to refer to some that have either historical relevance or were considered for this project. We will start by looking at what could be considered the first widely adopted standard for data serialization: XML, detailed in section 2.1.1. Next, in section 2.1.2, we will look at JSON (JavaScript Object Notation), the current dominant format used in the industry, and why it supplanted its predecessor. After it, in section 2.1.3, we move on to Protobuf, the standard we will use in this project, and elaborate on what distinguishes it from the incumbent formats. Finally, we dive into an alternative to Protobuf, Flatbuffers, and explain its trade-offs. Besides these four options detailed, there are a plethora of other options worth mentioning, such as Thift or Cap'n Proto, but these felt short either due to being outperformed by alternatives, lower community adoption, or being tethered to a specific framework.

2.1.1 XML

XML was developed by the XML Working Group (originally known as the SGML Editorial Review Board), which was formed under the auspices of the World Wide Web Consortium (W3C) in

1996 (Tim Bray et al., 1998), to create a markup language which combined the flexibility of SGML (Standard Generalized Markup Language) with the simplicity of HTML (Ray, 2003). The XML Working Group established a few key points when developing the standard, namely:

- Form should follow function: rather than invent a single, generic language to cover all document types, let there be many languages, each specific to its data (Ray, 2003).
- A document should be unambiguous: information must be structured in a way that there is only one way that it can be interpreted (Ray, 2003).
- Separate markup from presentation: to achieve maximum flexibility, styling should be kept separate from the document information. That way, if you wish to change any part of the presentation, you don't need to change the document data (Ray, 2003).
- Keep it simple: to gain widespread usage, the syntax would need to be simple and straightforward, as users would find no interest in learning the format otherwise (Ray, 2003).
- It should enforce maximum error checking: to maintain data and organization integrity, a file is only valid if it obeys a set of syntax parameters that ensure that the document won't cause inconsistent behaviour across interpreters (Ray, 2003).
- It should be culture-agnostic: unlike many markup languages, which limit their markup to narrow cultural spaces, like the Latin alphabet and English language, XML uses the Unicode character set, allowing any alphabet and language to be used (Ray, 2003).

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <maven.build.timestamp.format>yyyy-MM-dd'T'HH:mm:ss</maven.build.timestamp.format>
    <java.version>17</java.version>
    <protobuf.version>3.19.1</protobuf.version>
    <protobuf-plugin.version>0.6.1</protobuf-plugin.version>
    <grpc.version>1.53.0</grpc.version>

</properties>
```

Code Excerpt 1: Excerpt of an XML-formatted configuration file

2.1.2 JSON

JSON is a data interchange format based on a subset of the JavaScript Programming Language, formalized by Douglas Crockford in 2006, intending to address some of the shortcomings of XML, such as its lack of arrays and excessively verbose syntax. The language is built on two structures: a collection of name/value pairs and an ordered list of values (Douglas Crockford, 2017). Due to its lighter file size and easier readability, JSON took XMLs place as the dominant standard for data communication between systems, where it remains to this day.

```
"id": "978-1491900864",
"authors": [
        {
                "authorId": "RL1",
                "firstName": "Robert",
                "lastName": "Liguori",
                "birthDate": "01/05/1990",
                "country": "PT"
        },
        {
                "authorId": "PL1",
                "firstName": "Patricia",
                "lastName": "Liguori",
                "birthDate": "01/05/1990",
                "country": "ZA"
        }

]
```

Code Excerpt 2: Excerpt of a JSON-formatted response

2.1.3 Protobuf

Protocol buffers, or Protobuf for short, provide a language-neutral, platform-neutral, extensible mechanism for serializing structured data in a forward and backwards-compatible way (Google, 2023a). It was developed by Google initially for their internal service communication only, but following their push to open source several pieces of software, the format was published to the general public.

It is currently in its third version, which introduces some syntax simplifications compared to version two, along with minor interpretation changes.

Unlike the two previous protocols, Protobuf does not store data in text format. Instead, it serializes the data into a binary blob to reduce the size, making it faster than any text-based alternative.

The serialization scheme is defined in a `.proto` file shared between the client and server, removing any ambiguity on type definition.

```
syntax = "proto3";

service APIExemplarExemplarGRPC {

  rpc GetExemplarByID (RequestWithExemplarId) returns (ExemplarGrpcDto);
  rpc GetAllExemplars (google.protobuf.Empty) returns (stream ExemplarGrpcDto);
  rpc AddNewExemplar (CreatingExemplarGrpcDto) returns (ExemplarGrpcDto);
  rpc ModifyExemplar (ExemplarGrpcDto) returns (ExemplarGrpcDto);
  rpc DeleteExemplar (RequestWithExemplarId) returns (ExemplarGrpcDto);
}

message ExemplarGrpcDto {
  string ExemplarId = 1;
  string BookId = 2;
  int32 BookState = 3;
  string SellerId = 4;
  string DateOfAcquisition = 5;
}

message CreatingExemplarGrpcDto {
  string BookId = 1;
  int32 BookState = 2;
  string SellerId = 3;
  string DateOfAcquisition = 4;
}

message RequestWithExemplarId {
  string Id = 1;

}
```

Code Excerpt 3: Excerpt of a protocol buffer serialization scheme configuration

2.1.4 FlatBuffers

FlatBuffers is an efficient cross-platform serialization library for C++, C#, C, Go, Java, Kotlin, JavaScript, Lobster, Lua, TypeScript, PHP, Python, Rust, and Swift, developed at Google for game development and other performance-critical applications (Google, 2021a).

It is very similar to Protocol Buffers in many aspects, such as binary serialization and the usage of a serialization schema. However, FlatBuffers sacrifices compression size for faster deserialization times (Proos & Carlsson, 2020) and lower memory usage, making it geared towards latency-sensitive tasks, such as videogames, or when the applications target device has memory size constraints, such as IoT devices or phones (Google, 2014).

2.2 HTTP

When it comes to transport or communication protocols used in data transfer and exchange, the HTTP protocol is one of the most well-established and reliable options available. Since its debut in 1991 at the hands of Tim Berners-Lee, the protocol has seen many revisions, of which two are of higher interest: 1.1 and 2.0.

2.2.1 Version 1.1

HTTP version 1.1 was released in 1997 and is the oldest revision of the HTTP protocol still in use today. It introduced multiple methods that we rely on heavily, such as DELETE, PATCH and PUT, along with the capability of persistent connections (Fulber-Garcia, 2022). These new capabilities, together with the improvements introduced in version 1.0, formed a standard that would last 18 years before receiving a significant update.

The HTTP protocol provides the following functions:

- Headers: a way to transmit metadata on a request, granting the protocol flexibility and extensibility. Of these, one very notable header was Content-type, which allowed the protocol to transmit any file and not just a plain HTML file (Fulber-Garcia, 2022).
- Versioning: each request communicates which version of the protocol it uses to avoid version collisions (Fulber-Garcia, 2022).
- Status codes: a set of standardized codes that enable the receiver to know the response status at a glance, whether it failed or succeeded, with some degree of detail (Fulber-Garcia, 2022).
- Request methods: indicate the desired action the client wants to perform. The ones that see more use are GET, POST and the previously mentioned methods PUT, PATCH and DELETE.

2.2.2 Version 2.0

At the end of the 2000s, HTTP started to show some limitations due to websites evolving to include much more intricate client interfaces, thus, demanding more resources and bandwidth. Initially, Google was the first to react by releasing the SPDY protocol, intending to reduce latency and increase security. When this protocol started to show clear signs of higher performance versus the current version of HTTP, Mozilla also adopted it on their browser engine. Not long after, the IETF introduced version 2.0 of the HTTP protocol, which took the groundwork laid out by SPDY and build upon it (Fili Wiese, 2022). This new version introduced a few key improvements:

- Request Multiplexing: instead of making a connection for every request, version 2.0 can send and receive multiple requests asynchronously using a single connection (Fulber-Garcia, 2022).
- Request prioritization: in this version, we can specify the order we wish to receive each request, making it much more predictable. For example, it is much more important that

the page stylesheet arrives first than any Javascript files meant for interaction (Fulber-Garcia, 2022).

- Automatic compression: while in version 1.1, where you have to specify that you wish to compress your request, in 2.0, all requests are compressed using the GZip algorithm. (Fulber-Garcia, 2022) Additionally, version 2.0 introduced HPACK, eliminating redundant information in HTTP header packets, further increasing performance and lowering bandwidth (Cloudflare, 2023).
- Connection reset: a function that allows closing a connection between a server and a client and immediately opening a new one (Fulber-Garcia, 2022).
- Server push: to avoid having a server receiving lots of requests constantly, version 2.0 introduced server push functionality, which allows the server to attempt to predict resources that will be requested next, thus, pushing said resources to the client's cache (Fulber-Garcia, 2022).

Additionally, HTTP 2.0 became a binary-based protocol, rather than a text-based one, further increasing efficiency and speed (Fulber-Garcia, 2022).

2.3 REST

REST, an acronym for Representational State Transfer, is an architectural style for distributed network applications designed by Roy Fielding and presented in his PhD dissertation published in 2000. He proposed the following six constraints:

- Client-Server – Concerns should be separated between clients and servers, allowing them to evolve independently and granting more flexible system scalability (Fielding, 2000).
- Stateless – Each request must contain all the necessary information to satisfy it, as the server should only concern itself with the current request rather than tracking previous requests to respond correctly to follow-up requests (Fielding, 2000).
- Cache – To achieve a more efficient architecture, requests must be implicitly or explicitly labelled as cacheable or non-cacheable, allowing clients to reutilize previously requested information and reducing server load (Varanasi & Belida, 2015).
- Uniform Interface – Interactions between clients, servers, and possible intermediary components are based on the uniformity of their interfaces, which simplifies the

architecture and allows independent development and scalability as long as they implement the agreed-on contract (Varanasi & Belida, 2015).

- Layered System – allows an architecture composed of hierarchical layers and restricts knowledge beyond them (Fielding, 2000).
- Code-on-Demand – this optional constraint permits client functionality extension by downloading and executing code via applets or scripts (Varanasi & Belida, 2015).

Currently, the most prolific implementation of this architectural style is using the HTTP protocol, using uniform resource identifiers, or URIs, to map their functions, using the HTTP verbs (such as GET, POST, PUT, DELETE, and PATCH) to mark the intended operation, and using resource formats such as JSON and XML.

As it stands now, this implementation is the standard approach for microservices development, but with the ever-increasing number of services, it starts to show significant deficiencies.

2.4 gRPC

2.4.1 What is gRPC?

Remote Procedure Call, or RPC, is a technique used to build distributed systems, which consists of the ability of one machine to call a function in a remote machine without knowing it was a remote machine itself, using existing communications features in a transparent way (World Wide Web Consortium, 1992).

Before gRPC existed, Google had created a general-purpose RPC framework called Stubby to connect its various microservices across multiple datacentres and services. However, it was not standardized and was purpose-made with the company’s systems in mind (Louis Ryan, 2015). In March 2015, however, Google announced its new iteration of Stubby, intending to make it open-sourced (Indrasiri & Kuruppu, 2020). The result was gRPC, a high-performance remote procedure call framework designed to efficiently connect multiple microservices, regardless of the programming language, with support for load balancing, tracing, health check, and authentication (Google, 2021b).

The language features the following features that make it distinguishable from others:

- It uses the HTTP/2 protocol's new features, which bring several speed benefits compared to its predecessor and, more critically, allow bi-directional streaming, increasing the performance of very high-intensity data exchange between two services (Indrasiri & Kuruppu, 2020).

- It forgoes the usage of text formatting types, such as JSON and XML, in favour of Protobuf, a binary buffer-based serialization format that significantly lowers the size overhead compared to RESTful implementations (Indrasiri & Kuruppu, 2020).
- It has simple and well-defined structures and is strongly typed, which allows for a more coherent exchange of information between services (Indrasiri & Kuruppu, 2020).

The service definition is made with a Protobuf file by default, which defines the data structures, services, and communication contracts (Google, 2023b).

gRPC has four types of service methods, all of which can run synchronously or asynchronously:

- Unary RPCs, where a client sends a single request to the server, and the server responds with a single response (Google, 2022).

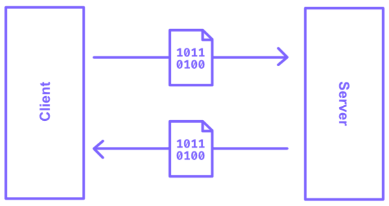


Figure 1: Representation of Unary RPCs (Simon Aronsson, 2020)

- Server Streaming RPCs, where a client sends a single request, and the server returns a stream from which it sends a sequence of messages, with their order maintained (Google, 2022).

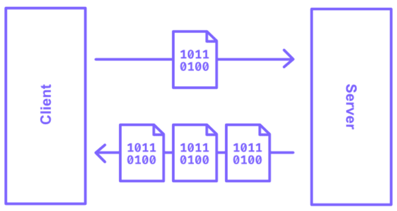


Figure 2: Representation of Server Streaming RPCs (Simon Aronsson, 2020)

- Client Streaming RPCs, where a client sends a sequence of messages, with their order maintained, through a stream provided to the server, and once done, waits for a single response from the server (Google, 2022).

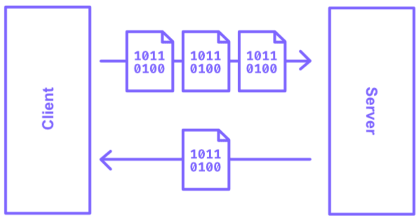


Figure 3: Representation of Client Streaming RPCs (Simon Aronsson, 2020)

- Bidirectional Streaming RPCs, where both sides send sequences of messages using streams. These streams operate independently, which multiple behaviours between the client and server. Like the previous methods, it keeps the messaging order through both streams (Google, 2022).

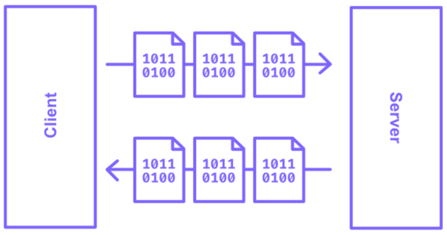


Figure 4: Representation of Bidirectional Streaming RPCs (Simon Aronsson, 2020)

Along with each RPC call, a client or server can send metadata containing information such as authentication data (Google, 2022).

2.4.2 Why gRPC?

Of all the technologies available currently, we're looking for one that would represent an increase in performance and would be suitable for projects of any size. With this in mind:

- SOAP doesn't represent a performance increase in any way, as it uses XML, a very high overhead format, and does not support caching.
- While not mentioned in the previous section, GraphQL is also a popular messaging protocol. However, it is strictly query-based, making it only suitable for data-fetching contexts.

With this in mind, gRPC emerges as the logical candidate for our project: its scalability allows its implementation on projects of any size, and the usage of the more advanced HTTP/2 protocol and Protobuf formatting provide an increase of flexibility and efficiency.

2.4.3 Industry Implementation Examples

With the development of this technology, multiple companies have started to adopt gRPC into their internal communication workflows. Such examples are:

- Uber moved the RAMEN system from SSE (Server-Sent Events) to gRPC to address a lack of reliability, connection management and overhead caused by text-based formatting in SSE. This change resulted in a connection latency reduction of at least 45% (Peng, 2022).
- Netflix, in 2015, switched its internal infrastructure to use gRPC as its communication protocol to reduce what, at the time, was a patchwork of handwritten custom RPC calls which lacked clarity and documentation, replacing it with simple Protobuf configuration lines, reducing their configuration code by 99% (Cloud Native Computing Foundation, 2018).
- Salesforce moved their internal REST-type APIs to gRPC to obtain better speed and interoperability between their services and address some issues with backward compatibility (Cloud Native Computing Foundation, 2019).

2.4.3.1 Netflix

Netflix is one of the biggest subscription-based content streaming platforms and one of the biggest success stories of the practical application of microservice architecture at scale (Indrasiri & Kuruppu, 2020). Initially, the company developed its communication stack for internal service communication using REST over HTTP/1.1, covering nearly 98% of its use cases (Indrasiri & Kuruppu, 2020). However, as they continued to scale up, they started to notice several bottlenecks and limitations related to their original approach: due to being a non-standardized communication stack, consumers required code written from scratch for every service, which proved to be very time-consuming, counter-productive and increase risk of error-prone code making to production (Indrasiri & Kuruppu, 2020). In addition, service implementation and consumption were also an issue, as they had no technologies to create a standardized definition of a service interface (Indrasiri & Kuruppu, 2020). Initially, the company developed their own internal RPC, but it soon decided that it needed a more stable approach, and after evaluation, gRPC proved to be the best fit for its business model (Indrasiri & Kuruppu, 2020). With this adoption, developers saw a massive boost in productivity, with a 99% reduction in configuration code (Cloud Native Computing Foundation, 2018), making what used to be hundreds of lines of code into just two or three lines of Protocol Buffers configuration files (Indrasiri & Kuruppu, 2020). This change also reduced the platform's latency and brought higher stability and maintainability as its code base shrunk significantly (Indrasiri & Kuruppu, 2020)

3 Analysis and Design

After presenting the analysis (section 3.1), following the ADD (Attribute-Driven Design) method (Cervantes & Kazman, 2016) terminology. all the drivers and concepts that influence architectural design are presented in section 3.2.

3.1 Analysis

3.1.1 Domain of the Problem

A used book store owner would like to showcase the book collection to its clients on a website, allowing them to browse and see the available books and their state. For the moment, the owner only wants to conduct transactions with clients in person but allows clients to request books via email after consulting the website. There are two main concepts in this business model: the book and the exemplar.

The exemplar represents the physical book item, identified by a unique identifier. By the nature of this being a used book store, the owner wants to keep track of the client who previously owned it to avoid being sold forgeries or stolen material, as well as the condition of the item and its price.

When it comes to the definition of a book, it is identified by its ISBN, title, language, publisher, its authors, the categories it belongs to, and a description of its content, in multiple languages if necessary. Book authors are characterized by their first and last names and country.

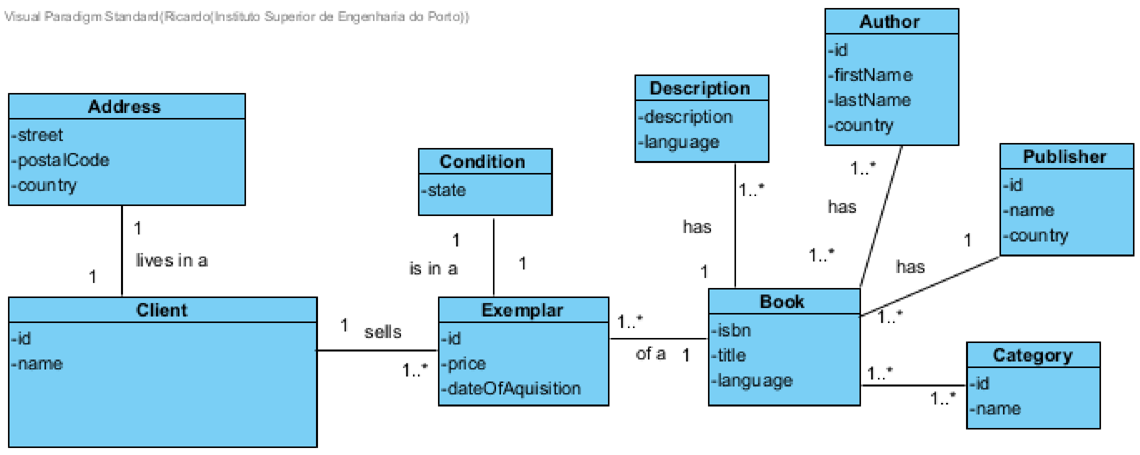


Diagram 1: Domain model

3.1.2 Functional Requirements

Utilizing the third version of Attribute Driven Design (Cervantes & Kazman, 2016), and after meetings with the product owner, I’ve established the following functional requirements:

- As a book store owner/employee:
  - Add/Edit/List/Delete authors.
  - Add/Edit/List/Delete categories.
  - Add/Edit/List/Delete publishers.
  - Add/Edit/List/Delete exemplars.
  - Add/Edit/List/Delete books.
  - Add/Edit/List/Delete clients.
  - List exemplars sold by clients.
- As a client:
  - List all exemplars.
  - Search for books by:
    - ISBN.
    - Author.
    - Language.
    - Publisher.

3.2 Design

Following the analysis process and continuing to follow the third version of Attribute Driven Design (Cervantes & Kazman, 2016), the design process starts with the architectural drivers, where we identify the Primary functional requirements, quality attributes and scenarios, non-functional constraints, and some architectural concerns. Next, we specify the design concepts and patterns applied in this project, and finally, we present diagrams representing the multiple levels of the system, following the C4 model.

3.2.1 Architectural Drivers

Per the analysis of the previous chapter, the Primary functional requirements are:

- Add/List authors.
- Add/List categories.
- Add/List publishers.
- Add/List exemplars.
- Add/List books.
- Add/List clients.
- List all exemplars.

Following ISO 25010 (International Organization for Standardization, 2011), the system design has a focus on maintainability, specifically:

- Modularity: the system must be composed of discrete microservices independent of each other.
- Reusability: the component construction must be ready to reuse parts of its core to interface with different types of communication systems.
- Testability: components must have automatic unit testing covering the domain model, integration testing in-between layers, and acceptance tests.

The maintainability must be evaluated with industry-standard software such as SonarQube or SonarGraph.

In terms of performance, the system must execute at least 500 sequential fetching operations in a period of 1 minute.

The project has a temporal constraint, as we've settled on a date to provide a prototype to present to the business owner that commissioned the system in question.

The software architecture shall be microservices-based, and the communication between the services and composer must be through HTTP REST and gRPC, as the company developing the software wishes to gather experience and performance data when using gRPC versus REST.

The development team must follow onion principles and domain-driven design when developing each microservice, applying the necessary patterns to achieve these principles.

The only system that interacts with the user is a composer, which gathers the information from the various microservices, which remain isolated.

3.2.2 Design Concepts

In this project's design, I apply the concept of Domain Driven Design, which takes us to organize the previous domain model into aggregates containing entities and value objects.

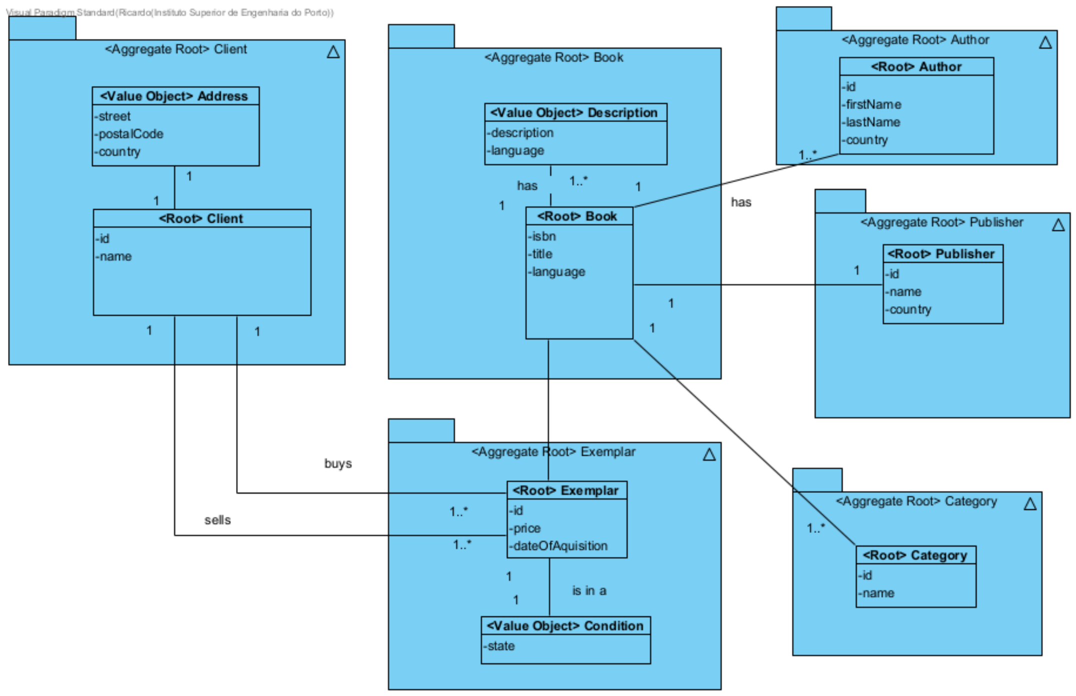


Diagram 2: Domain model with aggregates

Using the concept of microservices, we can divide each aggregate root into separate services.

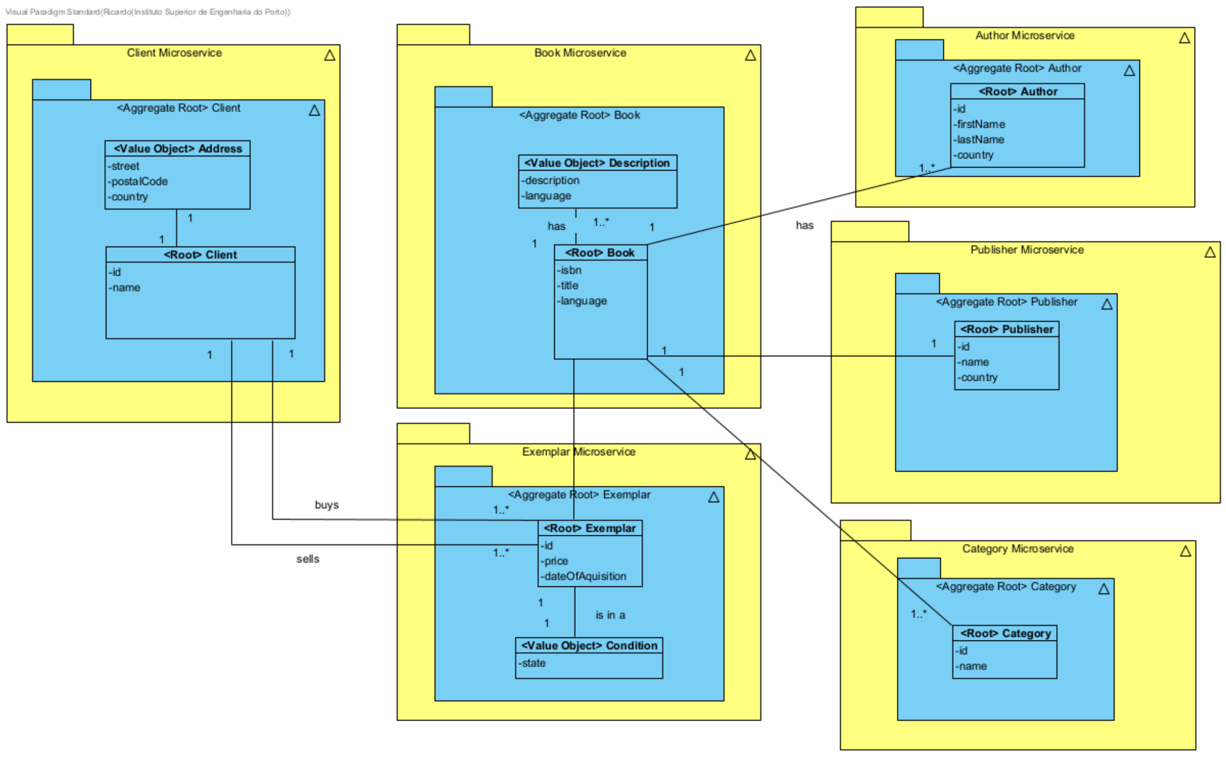


Diagram 3: Domain Model with aggregates and microservices defined

For software development, I adopted an Onion architecture, which means that each software layer only interacts with the layers adjacent to it, applying the Dependency Inversion Principle from SOLID. In addition, I also adopt the other four principles of SOLID: The Single Responsibility Principle, the Open/Close Principle, the Liskov Substitution Principle, and the Interface Segregation Principle.

To aid the adoption of these design concepts, I make use of some design patterns, namely:

- DTO Pattern: to encapsulate domain information that we might not want to transmit in some requests or for unifying requests from multiple microservices in the composer.
- Database per Service Pattern: derived from adopting a microservice architecture, it stands to reason to use this pattern which establishes that a microservice has its database, promoting low coupling (Richardson, 2019).
- Repository Pattern: grants us independence on data storage implementation by adopting an ORM framework rather than tethering to a specific database stack.
- Composer Pattern: to query business data from the system that requires multiple parts of information spread across multiple microservices, I need a piece of software that merges that information and serves it to the end user, which is the job of the composer (Richardson, 2019).

Since there is communication between different programming languages, it is paramount to settle on format standards for some properties due to their different native implementation to maintain full interoperability and avoid data corruption. In this case, when it comes to country identification, I'll be using the ISO 3166 alpha-2 (International Organization for Standardization, 2020) codes, and for language identification, the IETF BCP-47 (Phillips & Davis, 2009) is the chosen standard.

3.2.3 Design Alternatives

Upon analysing the architectural drivers and design concepts, I scaffolded a few possible designs for the system.

The first was communication through all services using gRPC and having one service which could toggle between REST and gRPC to evaluate the speed difference between technologies.

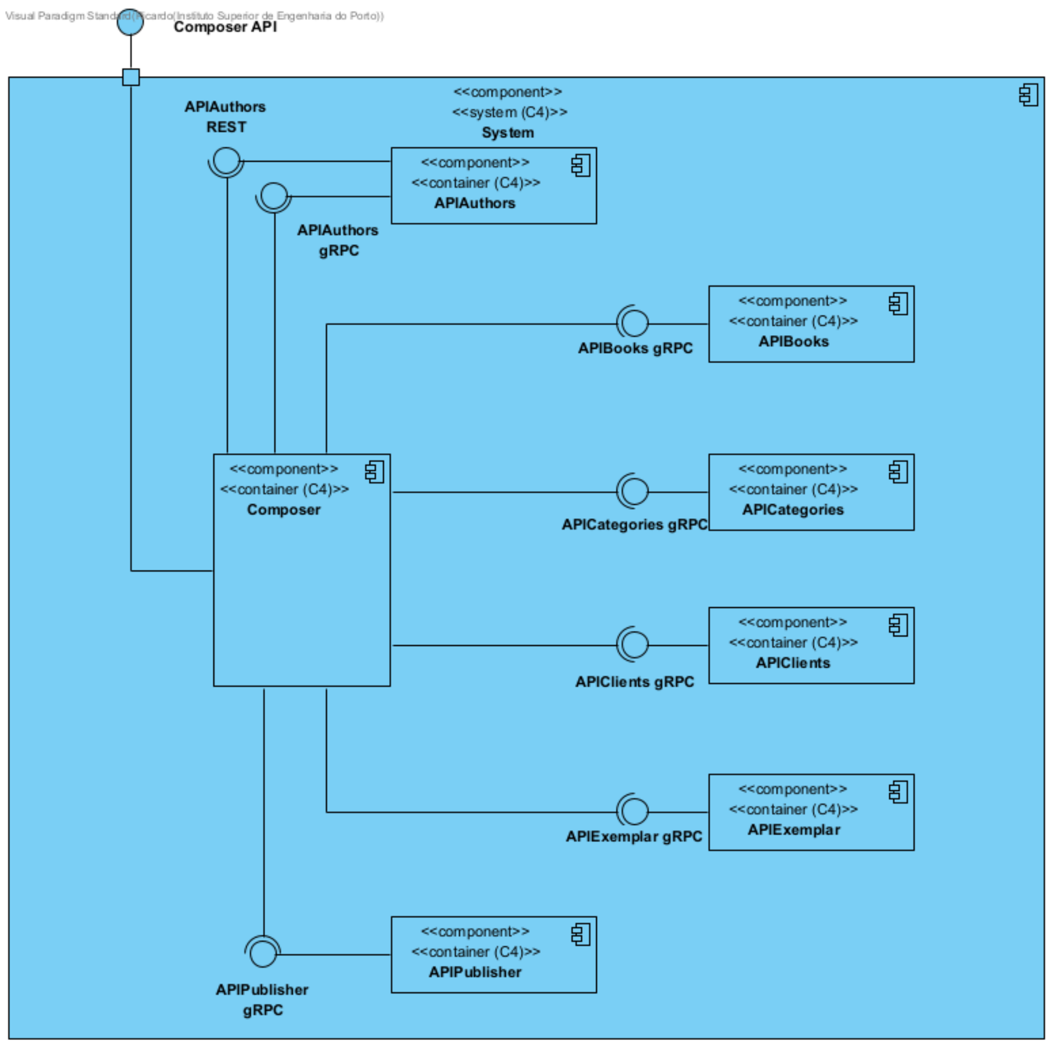


Diagram 4: System alternative design 1

However, I discarded this option since I am inexperienced with gRPC, which could present issues on all services if there were implementation issues.

Secondly was a minor revision of the previous design, with the microservices communicating via REST instead and one microservice being toggleable.

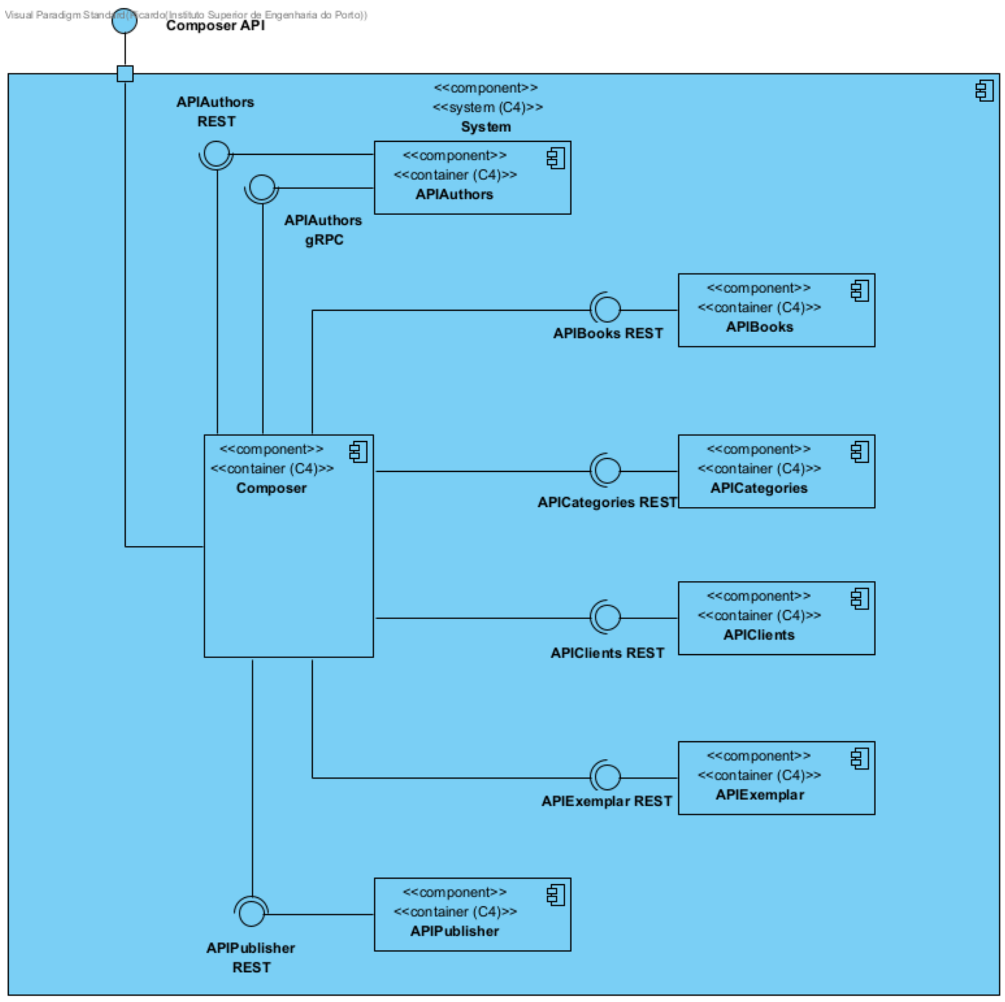

Visual Paradigm Standard(Ricardo(Instituto Superior de Engenharia do Porto))
Composer API
<<component>> <<system (C4)>> System
APIAuthors REST
APIAuthors gRPC
<<component>> <<container (C4)>> APIAuthors
APIBooks REST
<<component>> <<container (C4)>> APIBooks
<<component>> <<container (C4)>> Composer
APICategories REST
<<component>> <<container (C4)>> APICategories
APIClients REST
<<component>> <<container (C4)>> APIClients
APIExemplar REST
<<component>> <<container (C4)>> APIExemplar
APIPublisher REST
<<component>> <<container (C4)>> APIPublisher

Diagram 5: System alternative design 2

This design provides a reliable and well-known connection for most components and allows us still to evaluate the efficiency that gRPC can provide.

Ultimately, I decided not to go with this design, opting instead for the final design, which allows the composer to toggle between protocols for all microservices.

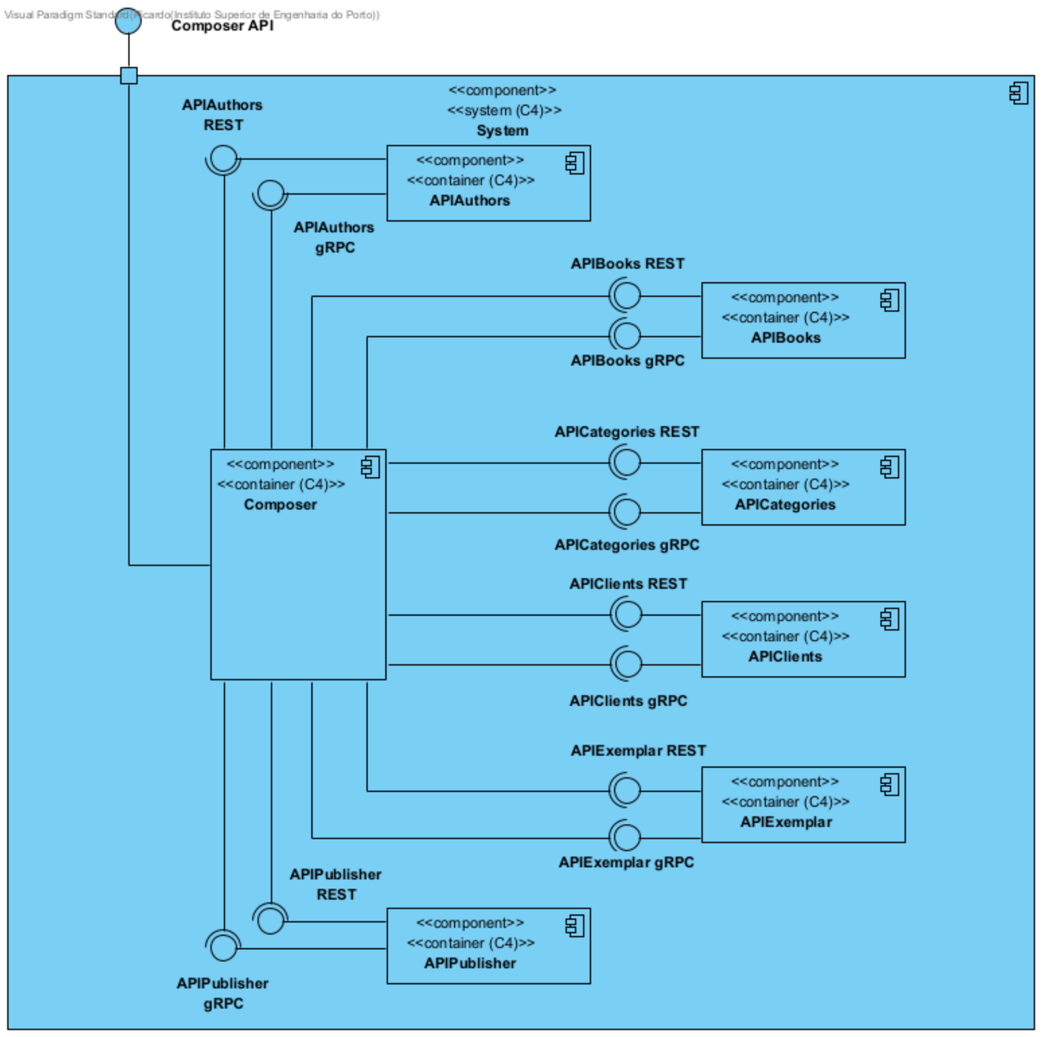


Diagram 6: System alternative design 3

This solution, while being more time-consuming, provides us with a more decisive evaluation of the speed of the two protocols in intercommunication operations.

3.2.4 Detailed Solution Design

With the component design chosen, I start to design its procedures and interactions between each and the outside world. To better convey the information, the solution diagrams follow the C4 Model (Simon Brown, 2018), a set of four hierarchical diagrams going from generic to detailed, combined with the 4+1 View Model (Kruchten, 1995), which provides multiple perspectives, or views of a software solution. This exposition will not present the fourth level, as it will be expanded upon later in the implementation chapter.

We start by looking at level one, known as system context, which shows how the solution appears to the outside world. As we can see, the system has only one external-facing interface, the composer.

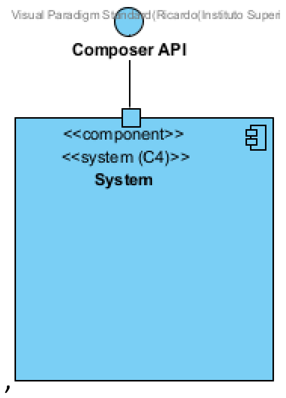


Diagram 7: Level 1 logic diagram

Advancing to level two, we see the various microservice containers and the composer, which fetches all the information from said microservices. As demonstrated in Chapter 3.2.2, we divided the business domain model into multiple microservices. Each microservice has its own ORM (Object-Relational Mapping) system and is responsible for its aggregate only and makes its resources available through two endpoints: one which uses HTTP REST and the other that uses gRPC. The composer, for his part, chooses which one of the endpoints it intends to use for communication, depending on its configuration.

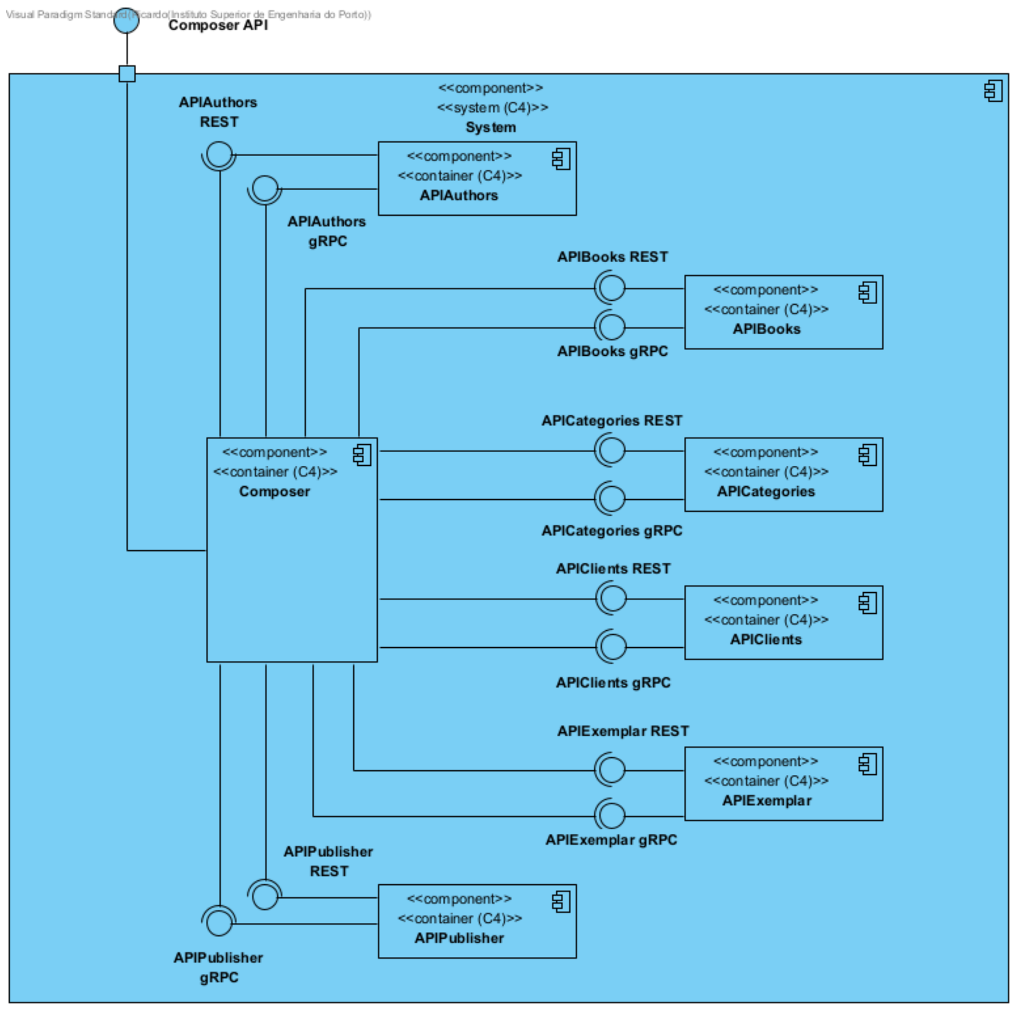


Diagram 8: Level 2 logic diagram

Jumping into level 3, where we detail the components, we have two diagrams: the APIAuthors microservice diagram, which represents the architecture adopted in all microservices, and the composer.

Starting with the APIAuthors diagram, we see the adoption of an onion architecture style, with controllers being the communication broker between the services and the interfaces, the services, which adopt the DTO pattern for data protection and segregation of the domain, the domain model, which is isolated from the outside world, and the persistence, which, using the repository pattern, manages and stores the domain information using an ORM framework. All layer communications use interfaces rather than specific classes to allow ease of modification of the layer implementation. Both protocols have independent controllers but share the same service interface, showing that adding support for gRPC to an HTTP REST solution requires minimal or no refactoring.

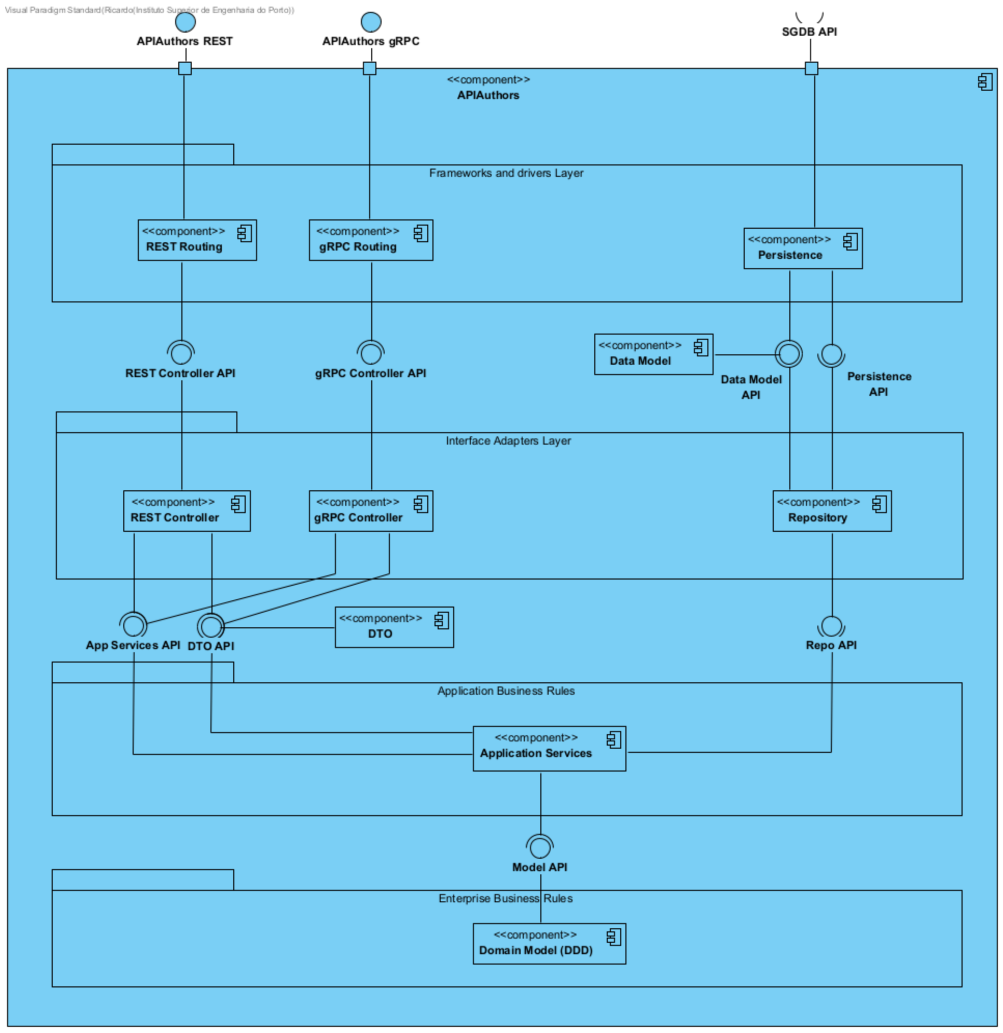

Visual Paradigm Standard(Ricardo(Instituto Superior de Engenharia do Porto))
APIAuthors REST
APIAuthors gRPC
SGDB API
<<component>> APIAuthors
Frameworks and drivers Layer
<<component>> REST Routing
<<component>> gRPC Routing
<<component>> Persistence
<<component>> Data Model
REST Controller API
gRPC Controller API
Data Model API
Persistence API
Interface Adapters Layer
<<component>> REST Controller
<<component>> gRPC Controller
<<component>> Repository
<<component>> DTO
App Services API
DTO API
Repo API
Application Business Rules
<<component>> Application Services
Model API
Enterprise Business Rules
<<component>> Domain Model (DDD)

Diagram 9: Level 3 logic diagram for the APIAuthors component

Moving onto the composer, we have a single HTTP REST controller, which will be the interface available from the outside, and connects to the application services. Here, we have two types of services: the composer and the fetching services. The composer services assemble objects that require multiple parts from multiple microservices by requesting the necessary data from the fetching services, which its sole purpose is to communicate with the systems microservices at the request of controllers or composer services. As with the microservices, all layers communicate via interfaces, with each fetching service having implementations for both protocols.

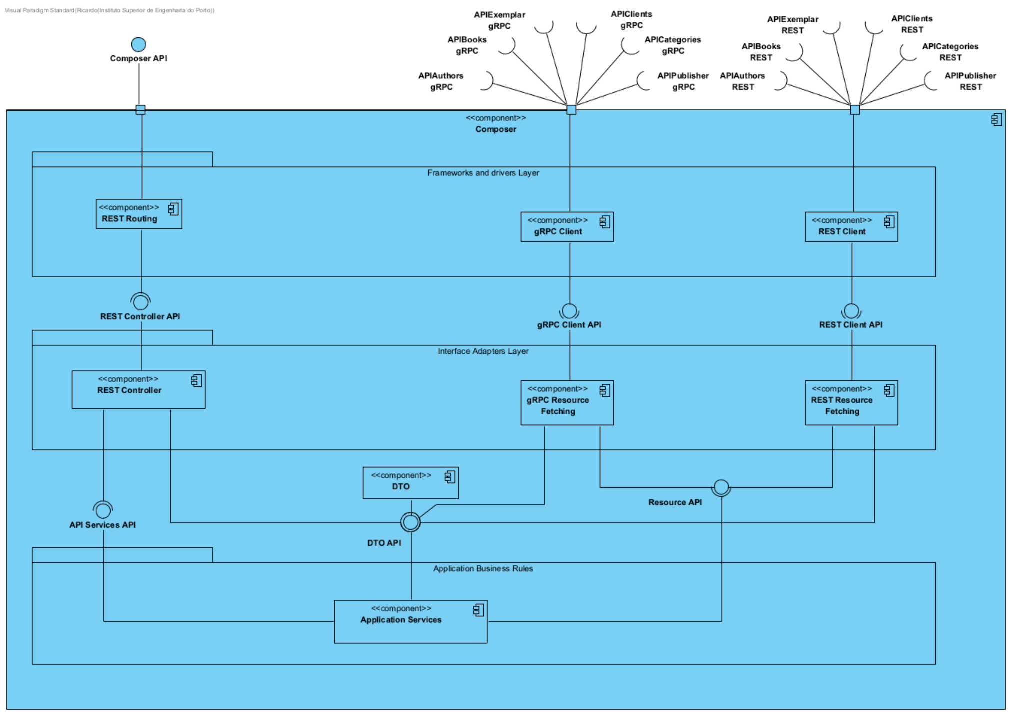
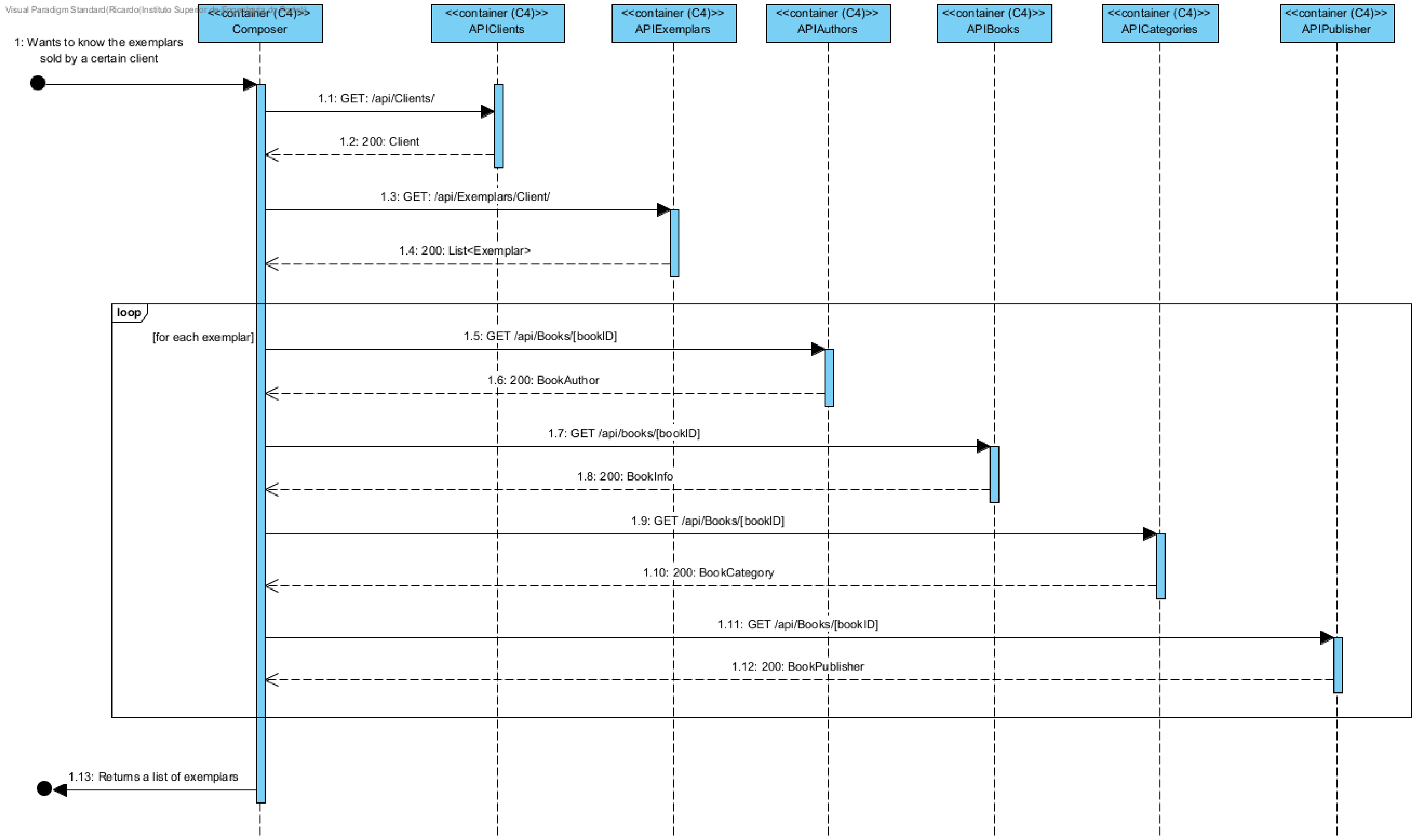


Diagram 10: Level 3 Logic diagram for the Composer component

Moving onto use case processes, I showcase a few diagrams detailing the use case List exemplars sold by a client. I will not present the level one diagram, as it conveys no relevant information.

On the level two diagram of this process, we can observe the mechanics involved in a use case that requires information from all microservices, demonstrating the mechanics of the composer.

The process starts with fetching the client information from APIClients, followed by requesting to APIExemplars to get all the books this client sold. After that, for each book exemplar, we fetch the book information, which requires getting the author information from APIAuthors, the general book information from APIBooks, the categories in which the book is in from APICategories, and finally, the book publisher from APIPublisher. After that, an object is built with all the information and served to the user.



Visual Paradigm Standard(Ricardo(Instituto Superior de Engenharia do Porto))
<<container (C4)>> Composer
<<container (C4)>> APIClients
<<container (C4)>> APIExemplars
<<container (C4)>> APIAuthors
<<container (C4)>> APIBooks
<<container (C4)>> APICategories
<<container (C4)>> APIPublisher
1: Wants to know the exemplars sold by a certain client
1.1: GET: /api/Clients/
1.2: 200: Client
1.3: GET: /api/Exemplars/Client/
1.4: 200: List<Exemplar>
loop
[for each exemplar]
1.5: GET /api/Books/[bookID]
1.6: 200: BookAuthor
1.7: GET /api/books/[bookID]
1.8: 200: BookInfo
1.9: GET /api/Books/[bookID]
1.10: 200: BookCategory
1.11: GET /api/Books/[bookID]
1.12: 200: BookPublisher
1.13: Returns a list of exemplars

Diagram 11: Level 2 Process Diagram of "List exemplars sold by client"

For level three, I first present the view of the APIAuthors microservice, which will represent the flow that all microservices roughly follow. The router receives the call from the composer and redirects it to the controller. The controller calls the required internal service, in this case, GetById, supplying the book ISBN, and then the service requests the object to the repository and converts it to a sanitized DTO to return to the controller and to send as a response to the composer.

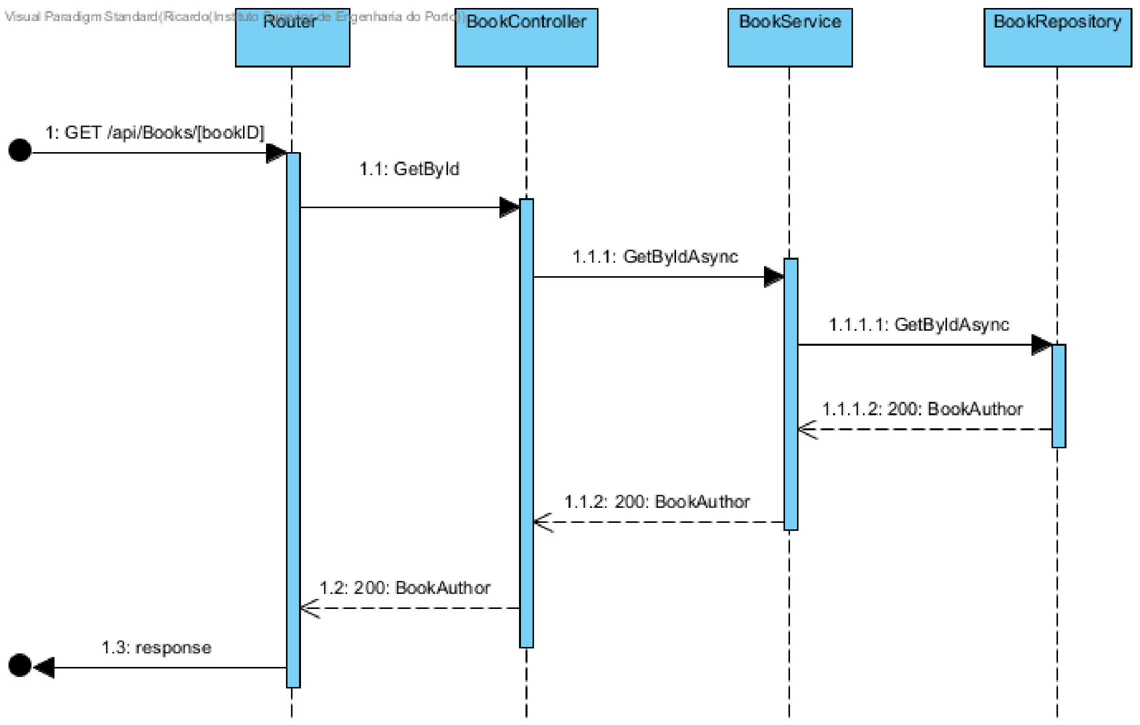


Diagram 12: Level 3 Process Diagram of "List exemplars sold by client" on the APIAuthors component

On the composer side, upon receiving the user request, it passes the client id to the ExemplarComposer service. This service will first obtain the client's information, followed by the list of book exemplars sold by said client. After that, for each book exemplar, it calls the BookComposerService to request the book information for said book exemplar. The BookComposerService aggregates all book information from microservices and returns an object containing all information. Once done, it returns a list of book exemplars to the user.

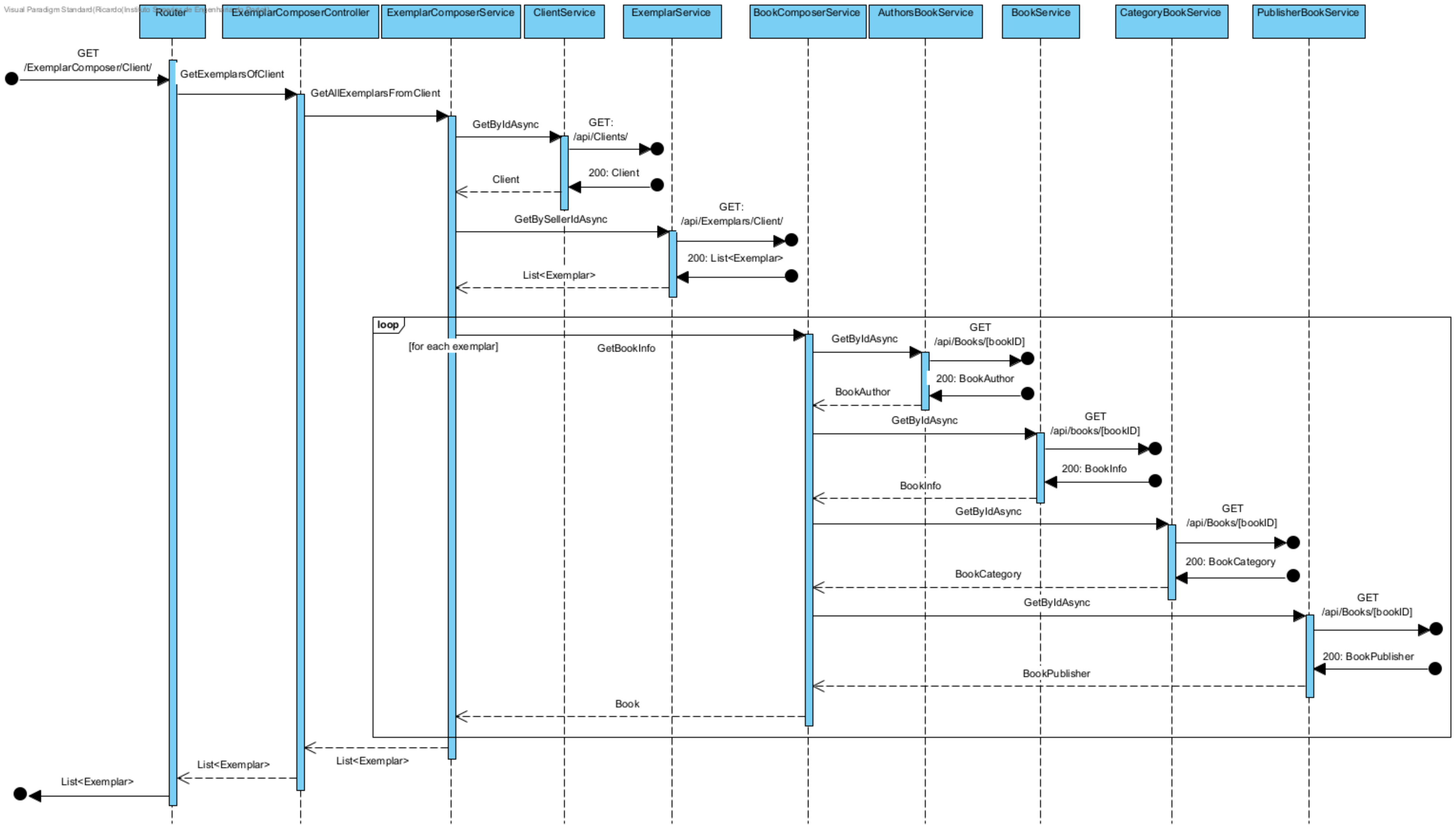


Diagram 13: Level 3 Process Diagram of "List exemplars sold by client" on the Composer component

4 Implementation

In this chapter, I demonstrate the practical implementation of the design detailed in the last chapter by going through the structure of the solution, referring to implementation details, especially in the gRPC implementation. I will also demonstrate the test suite that guarantees the system's functionality and showcase maintainability levels and protocol speed differences using industry-standard software.

4.1 Implementation Description

Following the analysis and design previously enacted in Chapter 3, I developed seven services. Given the absence of any language requirements, I used the following software development stacks:

Table 1: Software stacks used on each service

Service	**Stack**
Composer	C# .NET
Authors	C# .NET + EFCore
Books	Java Spring Boot + JPA
Categories	C# .NET + EFCore
Clients	C# .NET + EFCore
Exemplars	C# .NET + EFCore
Publishers	C# .NET + EFCore

All services that require data storage use In-Memory databases to eliminate possible bottlenecks from the database implementations on the ORM frameworks.

In terms of external dependencies, aside from the ORM frameworks, the C# .NET microservices implemented the native gRPC library (*GRPC for .NET*, 2018/2023) and Newtonsoft’s JSON parser (Newton-King, 2012/2023). The Java Spring Boot microservice implemented the native gRPC library, ModelMapper (*ModelMapper*, 2010/2023), which facilitates object-to-DTO conversions, and “gRPC Spring Boot Starter” (Zhang, 2016/2023), which facilitates the creation of gRPC controllers by automating their initialization.

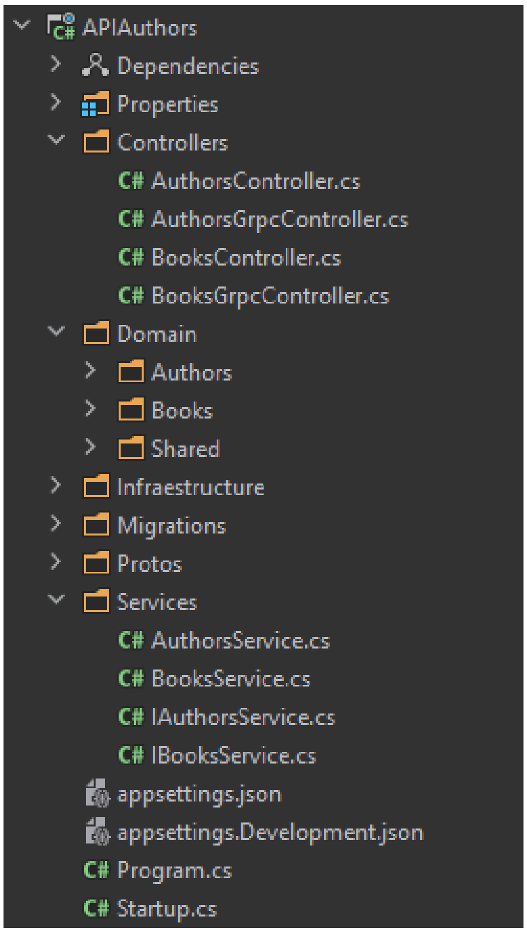
All microservices follow a similar structure, with controllers, both for REST and gRPC communication, a Service layer, which communicates with the repositories and converts domain model objects into DTO objects to be served to the controllers, and repositories, which manage the stored domain information using an ORM framework.

The composer, however, does not contain any repositories or domain models. Its only task is to coordinate actions between the microservices. Because of that, it has many services divided into two categories: data fetching and composition. The data fetching services handle the data exchange between all six microservices, with endpoints for both REST and gRPC, which can be toggled by a configuration setting, while the composition services organize and unify the information from the various microservices, to avoid cluttering the controller layer.

All C# .NET services used a modified version of dddnetcore (Nuno Silva, 2019/2020) as a starting point, while the Java Spring Boot is a service based on java-rest-books (Bouclier, 2017/2023), which was updated to Java 17 and adapted to fit the domain model.

4.1.1 Microservices

Starting with the microservices, all follow the same class structure: Controller, Services, Domain, and Infrastructure.



```
APIAuthors
    Dependencies
    Properties
    Controllers
        AuthorsController.cs
        AuthorsGrpcController.cs
        BooksController.cs
        BooksGrpcController.cs
    Domain
        Authors
        Books
        Shared
    Infraestructure
    Migrations
    Protos
    Services
        AuthorsService.cs
        BooksService.cs
        IAuthorsService.cs
        IBooksService.cs
    appsettings.json
    appsettings.Development.json
    Program.cs
    Startup.cs
```

Figure 5: Class structure of APIAuthors (Ricardo Mourisco, 2023b)

On the controller section, we can see separate controllers for HTTP REST and gRPC. While I would separate them regardless, it helps to have them so due to the gRPC controller extending a class generated by protoc, a Protobuf compiler that simplifies the implementation by reading the `.proto` files in your project and generating structural code for it, which includes all services, methods, and message payloads, simplifying the implementation process and reducing the boilerplate code needed to write.

```
public class AuthorsGrpcController : APIAuthorsAuthorGRPC.APIAuthorsAuthorGRPCBase
{
    private readonly IAuthorsService _service;

    public AuthorsGrpcController(IAuthorsService service)
    {
        _service = service;
    }

    public override async Task<AuthorGrpcDto> GetAuthorByID(RequestWithAuthorId
request, ServerCallContext context)
    {
        var author = await this._service.GetByIdAsync(new AuthorId(request.Id));
        if (author == null)
        {
            var metadata = new Metadata
            {
                { "ID", request.Id }
            };
            throw new RpcException(new Status(StatusCode.NotFound, "No data Found"),
metadata);
        }
        return new AuthorGrpcDto
        {
            AuthorId = author.AuthorId,
            FirstName = author.FirstName,
            LastName = author.LastName,
            BirthDate = author.BirthDate,
            Country = author.Country
        };
    }

}
```

Code Excerpt 4: Except of the gRPC Controller implementation in APIAuthors (Ricardo Mourisco, 2023b)

In the HTTP REST implementation, we use the existing frameworks on C# (ASP.NET Core MVC) and Java (spring-boot-starter-web). Both these frameworks provide a set of annotations that facilitate the definition of resource paths and background service daemons, which, when well used, provide high flexibility and maintainability.

```
namespace APIAuthors.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BooksController : ControllerBase {
        private readonly IBooksService _booksService;

        public BooksController(IBooksService booksService) {
            _booksService = booksService;
        }

        [HttpGet]
        public async Task<ActionResult<IEnumerable<BooksDto>>> GetAll()
        {
            return await _booksService.GetAllAsync();
        }

        [HttpGet("Author/{id}")]
        public async Task<ActionResult<IEnumerable<BooksDto>>>
GetAllFromAuthor(string id)
        {
            return await _booksService.GetByAuthorIdAsync(new AuthorId(id));
        }
        (...)
        }

}
```

Code Excerpt 5: Excerpt of the APIAuthors HTTP REST Controller implementation in C# (Ricardo Mourisco, 2023b)

```
@RestController
@RequestMapping(value = "/api/books")
public class BookController {
    private final IBookService bookService;

    @Autowired
    public BookController(IBookService bookService) {
        this.bookService = bookService;
    }

    @PostMapping
    public ResponseEntity<BookDto> createBook(@Valid @RequestBody CreatingBookDto
book, UriComponentsBuilder ucBuilder) {
        try {
            return new ResponseEntity<>(bookService.Add(book), HttpStatus.OK);
        }
        catch (BookIsbnAlreadyExistsException ex){
            return new ResponseEntity(HttpStatus.BAD_REQUEST);
        }
    }

    @GetMapping("/{isbn}")
    public ResponseEntity<BookDto> getBook(@PathVariable("isbn") String isbn) {
        try{
            return new ResponseEntity<>(bookService.Get(isbn), HttpStatus.OK);
        }
        catch (BookNotFoundException ex){
            return new ResponseEntity(HttpStatus.NO_CONTENT);
        }
    }
        (...)
}
```

Code Excerpt 6: Excerpt of the APIBooks HTTP REST Controller implementation in Java (Ricardo Mourisco, 2023c)

On the services side, the implementation applies the DTO pattern between the information from the repository and the information passed to the controller. Additionally, it implements interface communication for better maintainability and modularity of the solution.

```
public interface IAuthorsService
{
    Task<List<AuthorDto>> GetAllAsync();
    Task<AuthorDto> GetByIdAsync(AuthorId id);
    Task<AuthorDto> AddAsync(CreatingAuthorsDto dto);
    Task<AuthorDto> UpdateAsync(AuthorDto dto);
    Task<AuthorDto> DeleteAsync(AuthorId id);

}
```

Code Excerpt 7: Authors Service Interface in APIAuthors (Ricardo Mourisco, 2023b)

```
public class AuthorsService : IAuthorsService
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IAuthorsRepository _repo;

    public AuthorsService(IUnitOfWork unitOfWork, IAuthorsRepository repo)
    {
        this._unitOfWork = unitOfWork;
        this._repo = repo;
    }

    public async Task<AuthorDto> GetByIdAsync(AuthorId id)
    {
        var author = await this._repo.GetByIdAsync(id);

        if(author == null)
            return null;

        return new AuthorDto(author.Id.AsString(), author.Name.FirstName,
author.Name.LastName, author.BirthDate.ToString(), author.Country.Name);
    }

}
```

Code Excerpt 8: Authors Service Implementation in APIAuthors (Ricardo Mourisco, 2023b)

On Infrastructure, we use the ORM systems we chose for each language, configure the necessary parameters, and apply interface communication for maintainability and modularity.

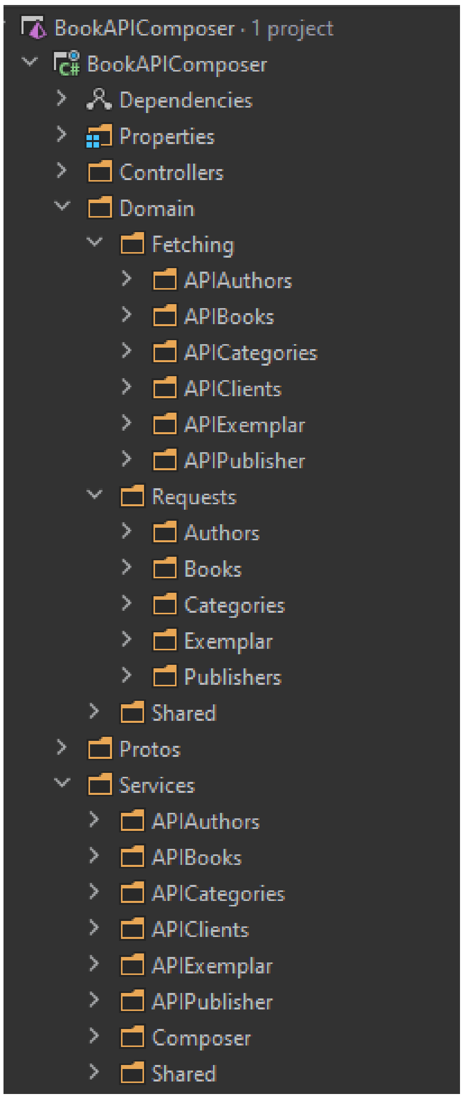
```
internal class AuthorEntityTypeConfiguration : IEntityTypeConfiguration<Author>
{
    public void Configure(EntityTypeBuilder<Author> builder)
    {
        builder.ToTable("Authors", SchemaNames.APIAuthors);
        builder.HasKey(author => author.Id);
        builder.OwnsOne(author => author.Name, Name =>
        {
            Name.Property(property => property.FirstName)
                .HasColumnName("FirstName")
                .IsRequired();
            Name.Property(property => property.LastName)
                .HasColumnName("LastName")
                .IsRequired();
        }).Navigation(author => author.Name).IsRequired();
        builder.Property(author => author.BirthDate)
            .HasColumnName("BirthDate")
            .IsRequired();
        builder.OwnsOne(author => author.Country, Country =>
        {
            Country.Property(property => property.Name)
                .HasColumnName("Country")
                .IsRequired();
        }).Navigation(author => author.Country).IsRequired();
    }

}
```

Code Excerpt 9: Author Entity ORM configuration class (Ricardo Mourisco, 2023b)

Finally, the domain contains the class object definitions and corresponding DTOs, with the needed business rules checks, such as the ISBN format.

4.1.2 Composer

The composer, written in C#, is the service that unifies the information from the various microservices and serves it to the user. This service has no business rules or checks, its only job is to send and assemble data to and from the microservices. The component divides into controllers, domain, and services.



```
BookAPIComposer · 1 project
  BookAPIComposer
    Dependencies
    Properties
    Controllers
    Domain
      Fetching
        APIAuthors
        APIBooks
        APICategories
        APIClients
        APIExemplar
        APIPublisher
      Requests
        Authors
        Books
        Categories
        Exemplar
        Publishers
      Shared
    Protos
    Services
      APIAuthors
      APIBooks
      APICategories
      APIClients
      APIExemplar
      APIPublisher
      Composer
      Shared
```

Figure 6: Class structure of Composer (Ricardo Mourisco, 2023a)

The controllers contain the HTTP REST routes for both the composition outputs and some passthrough controllers to add entities like Authors, who require no composition or decomposition. Like the microservices, this layer calls the needed service to satisfy its request to maintain low coupling and respect the single responsibility principle.

On the domain, we have two types of entities: fetching and requests. The fetching entities are associated with their fetching services and are the objects necessary to receive the data incoming from the microservices, while the request entities are the objects assembled by the composer services using data from various microservices. As referred above, objects do not have any business logic checks, as they act purely as DTOs, as doing so would not meet the composer pattern's rules, violate the single responsibility principle, and create a very maintainability issue, as any change in a microservice business rule would always require changes in the composer.

```
public class Book
{
    public string Id { get; set; }
    public string Title { get; set; }
    public string Language { get; set; }
    public IList<BookDescriptionsDto> Description { get; set; }
    public Publisher Publisher { get;  set; }
    public IList<Author> Authors { get; set; }
    public IList<Category> Categories { get; set; }

    public Book(string id, string title, string language,
IList<BookDescriptionsDto> description, Publisher publisher, IList<Author> authors,
IList<Category> categories)
    {
        Id = id;
        Title = title;
        Language = language;
        Description = description;
        Publisher = publisher;
        Authors = authors;
        Categories = categories;
    }

}
```

Code Excerpt 10: Book object that aggregates the information (Ricardo Mourisco, 2023a)

Finally, we have the services, of which we have two types: fetching and composing services.

Fetching services are the services which retrieve the information from the system's microservices. Each service has two implementations, facilitated by an interface to allow for modularity: the gRPC and the HTTP REST implementation.

The gRPC client, similarly to the case on microservices, uses code generated by the Protobuf compiler, which provides useful methods to create gRPC clients, call the various RPC methods and the DTOs that match the message protocols established on the `.proto` file, making it easy to implement and eliminates the need to write boilerplate code.

```
public interface IAuthorsService
{
    Task<List<AuthorDto>> GetAllAsync();
    Task<AuthorDto> GetByIdAsync(AuthorId id);
    Task<AuthorDto> AddAsync(CreatingAuthorsDto dto);
    Task<AuthorDto> UpdateAsync(AuthorDto dto);
    Task<AuthorDto> DeleteAsync(AuthorId id);

}
```

Code Excerpt 11: Authors Fetching Service Interface in the Composer (Ricardo Mourisco, 2023a)

```
public class AuthorsGrpcService : IAuthorsService
{
    private readonly APIAuthorsAuthorGRPC.APIAuthorsAuthorGRPCClient _client;

    public AuthorsGrpcService(string url)
    {
        var handler = new SocketsHttpHandler
        {
            (...)
        };
        var channel = GrpcChannel.ForAddress(url, new GrpcChannelOptions
        {
            HttpHandler = handler,
        });
        _client = new APIAuthorsAuthorGRPC.APIAuthorsAuthorGRPCClient(channel);
    }

    public async Task<List<AuthorDto>> GetAllAsync()
    {
        try
        {
            using var call = _client.GetAllAuthors(new Empty());
            List<AuthorDto> lstAuth = new List<AuthorDto>();
            await foreach (var reply in call.ResponseStream.ReadAllAsync())
            {
                lstAuth.Add(new AuthorDto(reply.AuthorId, reply.FirstName,
reply.LastName, reply.BirthDate, reply.Country));
            }
            return lstAuth;
        }
        catch (RpcException e)
        {
            Console.WriteLine("Error: {0}: {1}", e.Status.StatusCode,
e.Status.Detail);
            return null;
        }
    }

}
```

Code Excerpt 12: Excerpt of Author Fetching Services gRPC implementation in the composer (Ricardo Mourisco, 2023a)

On the HTTP REST side, I used the native HTTP Client implementation and JSON parser, implementing the same interface.

```
public class AuthorsRestService : IAuthorsService
{
    private readonly HttpClient _client;

    public AuthorsRestService(Uri uri)
    {
        _client = new HttpClient();
        _client.BaseAddress = uri;
        _client.DefaultRequestHeaders.Accept.Clear();
        _client.DefaultRequestHeaders.Accept.Add(
            new MediaTypeWithQualityHeaderValue("application/json"));
    }

    public async Task<List<AuthorDto>> GetAllAsync()
    {
        List<AuthorDto> auth = null;
        HttpResponseMessage response = await _client.GetAsync("api/Authors/");
        if (response.IsSuccessStatusCode)
        {
            auth = await response.Content.ReadFromJsonAsync<List<AuthorDto>>();
        }
        return auth;
    }
}
```

Code Excerpt 13: Excerpt of Author Fetching Services HTTP REST implementation in the composer (Ricardo Mourisco, 2023a)

To allow for easy switching between which protocol to use and the URLs for the microservices, I implemented a few configuration options: seven fields to define the services URLs and a bool, to toggle between gRPC or HTTP REST. The reason why there are seven fields for six services is that while the C# microservices can share the same port for HTTP REST and gRPC, the Java Spring microservice requires two distinct ports and thus requires two separate URLs for that specific service.

```
{
  (…)
  "APIBooksURL": "http://localhost:8080/",
  "APIBooksGrpcURL": "http://localhost:9090/",
  "APIAuthorsURL": "https://localhost:5001/",
  "APICategoriesURL": "https://localhost:5501/",
  "APIClientsURL": "https://localhost:6001/",
  "APIExemplarURL": "https://localhost:6501/",
  "APIPublisherURL": "https://localhost:7001/",
  "UseGrpc": false

}
```

Code Excerpt 14: Excerpt of the composer's appsettings.json configuration file (Ricardo Mourisco, 2023a)

The configuration parameters are then read on the program initialization to decide which services to inject and pass the correct URLs.

```
public class Startup
{
    (...)
    public void ConfigureMyServices(IServiceCollection services)
    {
        if (Convert.ToBoolean(Configuration["UseGrpc"]))
        {
            //APIAuthors
            services.AddSingleton<IAuthorsService>(dep => new
AuthorsGrpcService(Configuration["APIAuthorsURL"]));
            services.AddSingleton<IAuthorsBookService>(dep => new
AuthorsBookGrpcService(Configuration["APIAuthorsURL"]));

            //APIBooks
            services.AddSingleton<IBookService>(dep => new
BookGrpcService(Configuration["APIBooksGrpcURL"]));

            //APICategories
            services.AddSingleton<ICategoryService>(dep => new
CategoryGrpcService(Configuration["APICategoriesURL"]));
            services.AddSingleton<ICategoryBookService>(dep => new
CategoryBookGrpcService(Configuration["APICategoriesURL"]));

            //APIClients
            services.AddSingleton<IClientsService>(dep => new
ClientsGrpcService(Configuration["APIClientsURL"]));

            //APIExemplar
            services.AddSingleton<IExemplarService>(dep => new
ExemplarGrpcService(Configuration["APIExemplarURL"]));

            //APIPublisher
            services.AddSingleton<IPublisherService>(dep => new
PublisherGrpcService(Configuration["APIPublisherURL"]));
            services.AddSingleton<IPublisherBookService>(dep => new
PublisherBookGrpcService(Configuration["APIPublisherURL"]));
        }
        else
        {
            //APIAuthors
            services.AddSingleton<IAuthorsService>(dep => new AuthorsRestService(new
Uri(Configuration["APIAuthorsURL"])));
            services.AddSingleton<IAuthorsBookService>(dep => new AuthorsBookRestService(new
Uri(Configuration["APIAuthorsURL"])));

            //APIBooks
            services.AddSingleton<IBookService>(dep => new BookRestService(new
Uri(Configuration["APIBooksURL"])));

            //APICategories
            services.AddSingleton<ICategoryService>(dep => new CategoryRestService(new
Uri(Configuration["APICategoriesURL"])));
            services.AddSingleton<ICategoryBookService>(dep => new CategoryBookRestService(new
Uri(Configuration["APICategoriesURL"])));

            //APIClients
            services.AddSingleton<IClientsService>(dep => new ClientsRestService(new
Uri(Configuration["APIClientsURL"])));

            //APIExemplar
            services.AddSingleton<IExemplarService>(dep => new ExemplarRestService(new
Uri(Configuration["APIExemplarURL"])));

            //APIPublisher
            services.AddSingleton<IPublisherService>(dep => new PublisherRestService(new
Uri(Configuration["APIPublisherURL"])));
            services.AddSingleton<IPublisherBookService>(dep => new
PublisherBookRestService(new Uri(Configuration["APIPublisherURL"])));
        }
        (...)
    }

}
```

Code Excerpt 15: Excerpt of bootstrap code demonstrating configuration functionality (Ricardo Mourisco, 2023a)

Finally, we have the composition services, likely the most complex code in this system, requiring good task organization and optimization. These services take information from fetching services and aggregate their information to return an object with all information for, for example, the exemplars of a specific client. In this example, we first fetch the client's information and the exemplars that said client sold.

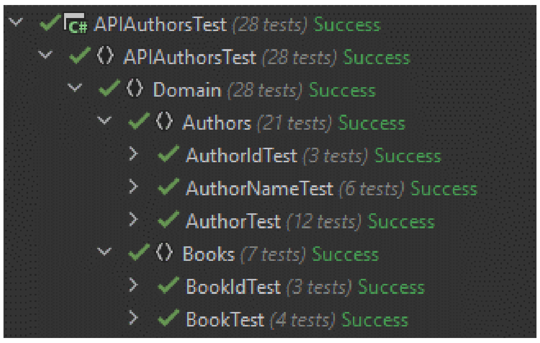
After that, we loop through all exemplars and obtain the book information. However, instead of blindingly fetching every book, we check if there is one other exemplar of the same book and copies its information, saving us time and bandwidth.

```
public class ExemplarComposerService : IExemplarComposerService
{
    private readonly IExemplarService _exemplarService;
    private readonly IClientsService _clientsService;
    private readonly IBookComposerService _bookComposerService;
    (...)
    public async Task<List<ExemplarPackage>> GetAllExemplarsFromClient(ClientId id)
    {
        var client = await _clientsService.GetByIdAsync(id);
        if (client == null)
        {
            return null;
        }

        var exemplars = await _exemplarService.GetBySellerIdAsync(id);
        if (exemplars == null)
        {
            return null;
        }

        List<ExemplarPackage> packages = new List<ExemplarPackage>();

        for (int i = 0; i < exemplars.Count; i++)
        {
            var existsAlready = packages.FirstOrDefault(pack => pack.Book.Id ==
exemplars[i].BookId);
            if (existsAlready == null)
            {
                var book = await _bookComposerService.GetBookInfo(new
BookId(exemplars[i].BookId));
                var exemplarList = new List<Exemplar>
                {
                    new Exemplar(exemplars[i].ExemplarId, exemplars[i].BookState,
                        exemplars[i].DateOfAcquisition, client)
                };
                packages.Add(new ExemplarPackage(book, exemplarList));
            }
            else
            {
                existsAlready.Exemplars.Add(new Exemplar(exemplars[i].ExemplarId,
exemplars[i].BookState,
                    exemplars[i].DateOfAcquisition, client));
            }
        }

        return packages;
    }
    (...)

}
```

Code Excerpt 16: Excerpt of Exemplar Composer Service in the composer (Ricardo Mourisco, 2023a)

4.2 Tests

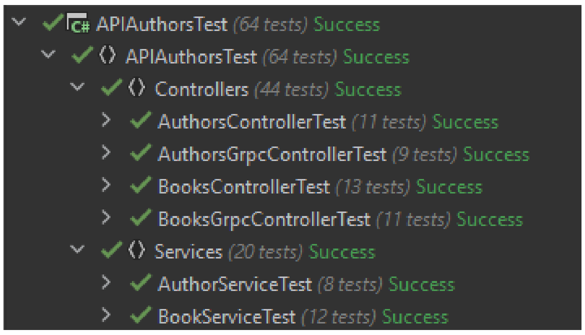
As defined in Section 3.2.1, I've developed automatic unit tests for the microservice's domain model, which cover all business rules defined by the product owner. These tests use the xUnit (*XUnit.Net*, 2013/2023) framework on C# and jUnit (*JUnit 5*, 2015/2023) on Java.



```
APIAuthorsTest (28 tests) Success
  APIAuthorsTest (28 tests) Success
    Domain (28 tests) Success
      Authors (21 tests) Success
        AuthorIdTest (3 tests) Success
        AuthorNameTest (6 tests) Success
        AuthorTest (12 tests) Success
      Books (7 tests) Success
        BookIdTest (3 tests) Success
        BookTest (4 tests) Success
```

Figure 7: Unit test success for the domain of APIAuthors (Ricardo Mourisco, 2023b)

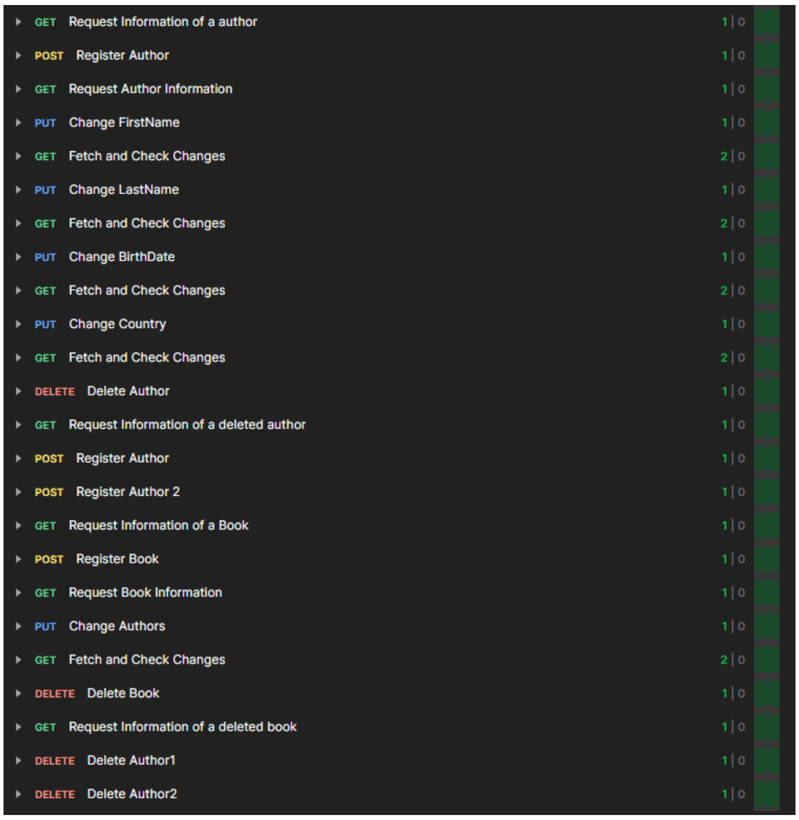
Next, I wrote inter-layer integration tests for the microservices to assure the correct function of layers. These tests used the previously mentioned testing frameworks along with mocking frameworks.



```
APIAuthorsTest (64 tests) Success
  APIAuthorsTest (64 tests) Success
    Controllers (44 tests) Success
      AuthorsControllerTest (11 tests) Success
      AuthorsGrpcControllerTest (9 tests) Success
      BooksControllerTest (13 tests) Success
      BooksGrpcControllerTest (11 tests) Success
    Services (20 tests) Success
      AuthorServiceTest (8 tests) Success
      BookServiceTest (12 tests) Success
```

Figure 8: Inter-layer integration tests between controller, service and repository of APIAuthors (Ricardo Mourisco, 2023b)

Finally, utilizing Postman for HTTP, and k6 for gRPC, I've created acceptance tests that test the use cases defined by the product owner, along with some direct microservice testing.



```
GET     Request Information of a author            1 | 0
POST    Register Author                            1 | 0
GET     Request Author Information                 1 | 0
PUT     Change FirstName                           1 | 0
GET     Fetch and Check Changes                    2 | 0
PUT     Change LastName                            1 | 0
GET     Fetch and Check Changes                    2 | 0
PUT     Change BirthDate                           1 | 0
GET     Fetch and Check Changes                    2 | 0
PUT     Change Country                             1 | 0
GET     Fetch and Check Changes                    2 | 0
DELETE  Delete Author                              1 | 0
GET     Request Information of a deleted author    1 | 0
POST    Register Author                            1 | 0
POST    Register Author 2                          1 | 0
GET     Request Information of a Book              1 | 0
POST    Register Book                              1 | 0
GET     Request Book Information                   1 | 0
PUT     Change Authors                             1 | 0
GET     Fetch and Check Changes                    2 | 0
DELETE  Delete Book                                1 | 0
GET     Request Information of a deleted book      1 | 0
DELETE  Delete Author1                             1 | 0
DELETE  Delete Author2                             1 | 0
```

Figure 9: Postman test steps and success (Ricardo Mourisco, 2023b)

```
import grpc from 'k6/net/grpc';
import { check } from "k6";

let URL = 'localhost:5501';
let client = new grpc.Client();
client.load(['Protos'], 'APICategories.proto');

export default () => {
    client.connect(URL, {})

    let dataTagOnly = { Id: 'TEG' }

    let response =
client.invoke('APICategoriesRPC.APICategoriesCategoryGRPC/GetCategoryByID',
dataTagOnly);
    check(response, {'status is not found': (r) => r && r.status ===
grpc.StatusNotFound});

    let data = {
        CategoryId: 'TEG',
        Name: 'Tag1'
    };

    response =
client.invoke('APICategoriesRPC.APICategoriesCategoryGRPC/AddNewCategory', data);
    check(response, {'status is ok Add': (r) => r.status === grpc.StatusOK});

    response =
client.invoke('APICategoriesRPC.APICategoriesCategoryGRPC/GetCategoryByID',
dataTagOnly);
    check(response, {'status is ok Get': (r) => r.status === grpc.StatusOK});

    data = {
        CategoryId: 'TEG',
        Name: 'Tag2'
    };

    response =
client.invoke('APICategoriesRPC.APICategoriesCategoryGRPC/ModifyCategory', data);
    check(response, {'status is ok after changing Name': (r) => r.status ===
grpc.StatusOK,
        'Name change success': (r) => r.message.Name === 'Tag2'});

    response =
client.invoke('APICategoriesRPC.APICategoriesCategoryGRPC/DeleteCategory',
dataTagOnly);
    check(response, { 'status is ok delete': (r) => r.status === grpc.StatusOK});

    client.close()

}
```

Code Excerpt 17: k6 test script for APICategories gRPC endpoint (Ricardo Mourisco, 2023b)

All these tests are available in their respective repositories, with the Postman collections under the Postman folder and the k6 scripts under the k6 folder (Ricardo Mourisco, 2023d).

4.3 Solution Evaluation

As detailed in Chapter 3, we've determined two measurable metrics required by the product owner: maintainability and response time.

4.3.1 Maintainability

Starting with maintainability, to evaluate all components, I used SonarGraph, a static code analyser that allows you to monitor a software system for technical quality and enforce rules regarding software architecture (hello2morrow, 2023b) and provides metrics from which we can infer the quality of the software. In the following table, I have presented some prominent quality metrics provided by SonarGraph. Note that all five C# microservices are grouped into one solution, and thus, evaluated together.

Table 2: Maintainability metrics for the developed software

	C# Microservices	**Java Microservice**	**C# Composer**
ACD	3.16	3.93	4.96
Maintainability (%)	94.54%	91.07%	86.13%
Relative Entanglement (%)	0.00%	0.00%	0.00%
Average Complexity	2,28	1.37	2.61

- ACD: Average component dependency, as defined by John Lakos (Lakos, 1996), which takes the average number of components a component depends on directly and indirectly, allowing us to evaluate global coupling (hello2morrow, 2023a).
- Maintainability: This metric estimates maintainability as a percentage by looking at the dependency structure between components based on several factors, both negative, like cyclic dependencies and low-level classes with a high volume of dependencies, and positive, such as keeping good vertical boundaries, avoiding having too many layers and having components being independent, as in, not having any dependencies (hello2morrow, 2023a).
- Relative Entanglement: consists of the sum of relative cyclicities on component and namespace/directory levels. The higher the value, the more likely the presence of very large cycle groups in the solution (hello2morrow, 2023a).
- Average Complexity: a "weighted average modified extended cyclomatic complexity for fully analysed code" (hello2morrow, 2023a), as formulated by Thomas J. McCabe (McCabe, 1976).

The full reports, which contain more quality metrics, can be found in their respective repositories (Ricardo Mourisco, 2023d).

4.3.2 Performance

For performance, we had an objective of the system being capable of 500 sequential fetching operations over one minute. However, we also were looking to infer the performance difference between protocols. We've used jMeter (*Apache/Jmeter*, 2010/2023), a performance and stress test program to test the response time and throughput. The set-up consists of one thread sequentially executing a fetching operation during a period of one minute, with no caching functionalities enabled and with a 10-second warm-up period.

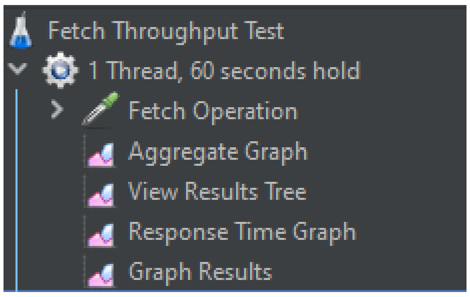

Fetch Throughput Test
1 Thread, 60 seconds hold
Fetch Operation
Aggregate Graph
View Results Tree
Response Time Graph
Graph Results

Figure 10: jMeter testing setup

Table 3: Response Times: gRPC vs HTTP REST

	gRPC	HTTP REST
Nº of Operations	826	509
Average Response Time (ms)	72	117
Median Response Time (ms)	70	115
95% Line (ms)	80	124

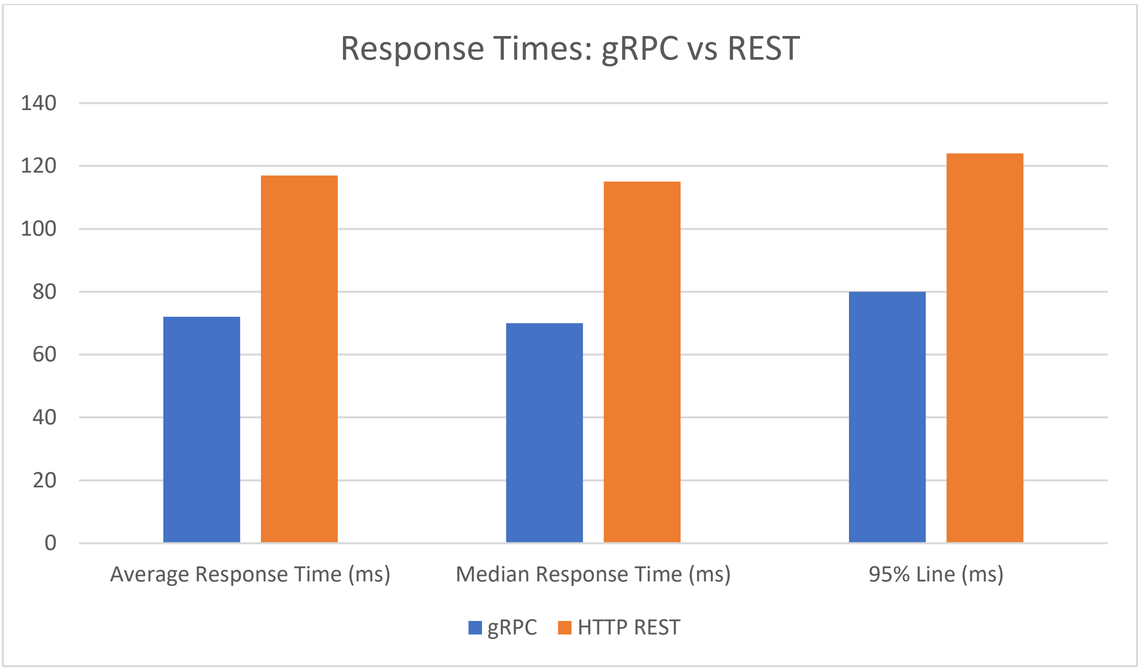


Figure 11: Response Times: gRPC vs HTTP REST

With the results obtained, we can assert that, in this test, both meet the defined throughput objective and that the gRPC protocol performs 61% faster than the HTTP REST protocol. The median response and 95% line follow the same trend, with the absolute time difference varying at most by 1 millisecond.

The test script used is available on the composer repository (Ricardo Mourisco, 2023a) so that it can be peer-reviewed by other developers.

5 Conclusion

5.1 Achievements

Of the objectives set forth at the start of this project, all of them were achieved:

- I've explored the use of gRPC in service-to-service communication, its development tools and experience, and the difficulty of implementing it alongside a classic HTTP REST endpoint.
- I compared the performance between gRPC and HTTP REST for communication, showing that gRPC is, in general, faster than HTTP REST.

In addition to that, derived from the requirements set forth by the product owner, I've developed an efficient and easily maintainable solution for its use case.

All developed software is publicly available on a project page (Ricardo Mourisco, 2023d), or in its respective repositories, allowing others to easily continue or expand my work if they choose to do so.

5.2 Threats to Validity

In this project, there were some pitfalls in the development and testing that have varying degrees of severity. These limitations are:

- Suboptimal testing environment: the fact that this project was designed by myself only and with limited resources, all testing occurred on my machine, which is not ideal. Ordinarily, you'd like to test in various machines and possibly even in virtualized server environments. Adding to that, my system is not isolated, and therefore, could introduce discrepancies in performance testing due to scheduling of background tasks, for example.
- Insufficient performance testing: while the testing demonstrated in this report is sufficient to conclude the proposals set at the beginning of the project, there is still quite a lot of room for more testing, not only of more scenarios but potentially different tools, like Grafana's k6. Ultimately, the time constraint was the bigger roadblock to performance testing expansion.
- DTO Layer involved with gRPC communication: as mentioned in the report, gRPC, via the protoc compiler, generates its own DTOs based on the message fields in the `.proto` file. This implicates an extra layer of DTO translation if one does not want to also

reimplement the service layer specifically to serve the gRPC endpoint. Furthermore, on the composer side, you run into the same problem, with the necessity of unification of objects into one set.

5.3 Future Work

Along with that, this project can still be improved and expanded upon, with things such as:

- Improved logging and error communication: the error and message communication, especially on the gRPC side is currently very crude and would benefit from improvements on, for example, error specificity.
- Expand features: while the system does pack a sizeable number of features, it can still be expanded and enhanced in many different areas. One of these features is the ability to fuzzy search for some fields, like Category names or ISBNs.
- Add authentication: originally planned but discarded due to the limited timetable, authentication would be necessary before the system ever goes into production. This includes authentication of higher positions, such as the book store owner, to specific certificates for communication between microservices to avoid possible man-in-the-middle attacks.
- Docker support: microservices benefit from having docker container builds, as it eases the deployment process and expansion.

5.4 Personal Appreciation

Overall, this project allowed me to expand my knowledge in various areas, such as communication protocols, with more focus on gRPC, the usage of the microservice design and its patterns, which I hadn’t previously used, and performance measuring tools and software like jMeter and SonarGraph. I’m happy with the end product developed, even though, I would’ve liked to have had more time to thoroughly flesh out some aspects, and I’ve enjoyed the experience involved with the research, development and problem-solving that surrounded this project.

References

Apache/jmeter. (2023). [Java]. The Apache Software Foundation. https://github.com/apache/jmeter (Original work published 2010)

Bouclier, S. (2023). *Java-rest-books* [Java]. https://github.com/sbouclier/java-rest-books (Original work published 2017)

Cervantes, H., & Kazman, R. (2016). *Designing software architectures: A practical approach*. Addison-Wesley.

Cloud Native Computing Foundation. (2018, December 4). *Netflix*. Cloud Native Computing Foundation. https://www.cncf.io/case-studies/netflix/

Cloud Native Computing Foundation. (2019, August 28). *Salesforce*. Cloud Native Computing Foundation. https://www.cncf.io/case-studies/salesforce/

Cloudflare. (2023). *HTTP/2 vs. HTTP/1.1*. Cloudflare. https://www.cloudflare.com/learning/performance/http2-vs-http1.1/

Douglas Crockford. (2017, December). *JSON*. https://www.json.org/json-en.html

Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures* [Doctoral dissertation]. University of California, Irvine.

Fili Wiese. (2022, June 2). *HTTP/2*. Http.Dev. https://http.dev/2

Fulber-Garcia, V. (2022, March 29). *HTTP: 1.0 vs. 1.1 vs 2.0 vs. 3.0 | Baeldung on Computer Science*. https://www.baeldung.com/cs/http-versions

Google. (2014). *FlatBuffers: FlatBuffers white paper*. https://flatbuffers.dev/flatbuffers_white_paper.html

Google. (2021a). *FlatBuffers: FlatBuffers*. https://flatbuffers.dev/index.html#flatbuffers_overview

Google. (2021b, November 9). *About gRPC | gRPC*. https://grpc.io/about/

Google. (2022, December 21). *Core concepts, architecture and lifecycle*. GRPC. https://grpc.io/docs/what-is-grpc/core-concepts/

Google. (2023a, January 20). *Protocol Buffers—Overview*. Protocol Buffers Documentation. https://protobuf.dev/overview/

Google. (2023b, February 16). *Introduction to gRPC*. GRPC. https://grpc.io/docs/what-is-grpc/introduction/

GRPC for .NET. (2023). [C#]. grpc. https://github.com/grpc/grpc-dotnet (Original work published 2018)

hello2morrow. (2023a). *21.1. Language Independent Metrics*. https://eclipse.hello2morrow.com/doc/standalone/content/core_metrics.html

hello2morrow. (2023b). *hello2morrow—Sonargraph*. https://www.hello2morrow.com/products/sonargraph

Indrasiri, K., & Kuruppu, D. (2020). *gRPC: Up and running: building cloud native applications with Go and Java for Docker and Kubernetes* (First edition). O'Reilly.

International Organization for Standardization. (2011, March). *Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models (ISO Standard No. 25010:2011)*. ISO. https://www.iso.org/standard/35733.html

International Organization for Standardization. (2020, August). *Codes for the representation of names of countries and their subdivisions—Part 1: Country code (ISO Standard No. 3166-1:2020)*. ISO. https://www.iso.org/standard/72482.html

JUnit 5. (2023). [Java]. JUnit. https://github.com/junit-team/junit5 (Original work published 2015)

Kruchten, P. B. (1995). The 4+1 View Model of architecture. *IEEE Software*, *12*(6), 42–50. https://doi.org/10.1109/52.469759

Lakos, J. (1996). *Large-scale C++ software design*. Addison-Wesley Pub. Co.

Louis Ryan. (2015, September 8). *GRPC Motivation and Design Principles*. GRPC. https://grpc.io/blog/principles/

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

ModelMapper. (2023). [Java]. ModelMapper. https://github.com/modelmapper/modelmapper (Original work published 2010)

Newton-King, J. (2023). *Json.NET* [C#]. https://github.com/JamesNK/Newtonsoft.Json (Original work published 2012)

Nuno Silva. (2020). *Dddnetcore* [C#]. https://bitbucket.org/nunopsilva/dddnetcore/src/master/ (Original work published 2019)

Peng, A. R., Shahbaz Kaladiya, Shivani Bhatia, Xinlin. (2022, August 16). *Uber's Next Gen Push Platform on gRPC*. Uber Blog. https://www.uber.com/en-GB/blog/ubers-next-gen-push-platform-on-grpc/

Phillips, A., & Davis, M. (2009). *Tags for Identifying Languages* (Request for Comments RFC 5646). Internet Engineering Task Force. https://doi.org/10.17487/RFC5646

Proos, D. P., & Carlsson, N. (2020). *Performance Comparison of Messaging Protocols and Serialization Formats for Digital Twins in IoV*.

Ray, E. T. (2003). *Learning XML* (Second edition). O'Reilly.

Ricardo Mourisco. (2023a). *Csharp-composer* [C#]. https://bitbucket.org/1170640/csharp-composer/src/master/

Ricardo Mourisco. (2023b). *Csharp-services* [C#]. https://bitbucket.org/1170640/csharp-services/src/master/

Ricardo Mourisco. (2023c). *Java-books* [Java]. https://bitbucket.org/1170640/java-books/src/master/

Ricardo Mourisco. (2023d). *GRPC Communication*. 1170640 / GRPCCOM - Bitbucket. https://bitbucket.org/1170640/workspace/projects/GRPCCOM

Richardson, C. (2019). *Microservices patterns: With examples in Java*. Manning Publications.

Simon Aronsson. (2020, November 12). *Performance testing gRPC services*. https://k6.io/blog/performance-testing-grpc-services

Simon Brown. (2018). *The C4 model for visualising software architecture*. https://c4model.com/

Tim Bray, Jean Paoli, & C. M. Sperberg-McQueen. (1998, February 10). *Extensible Markup Language (XML) 1.0*. https://www.w3.org/TR/1998/REC-xml-19980210.html#sec-intro

Varanasi, B., & Belida, S. (2015). *Spring REST*. Apress Distributed to the book trade worldwide by Springer.

World Wide Web Consortium. (1992). *What Is RPC? (Introduction to RPC)*. https://www.w3.org/History/1992/nfs_dxcern_mirror/rpc/doc/Introduction/WhatIs.html

XUnit.net. (2023). [C#]. xUnit.net. https://github.com/xunit/xunit (Original work published 2013)

Zhang, M. (2023). *GRPC Spring Boot Starter* [Java]. https://github.com/yidongnan/grpc-spring-boot-starter (Original work published 2016)