Politécnico do Porto

Instituto Superior de Engenharia do Porto

Using Protocol Buffers in REST applications

Tiago Adriano Mendes Ribeiro

LETI

Degree in Telecommunications and Informatics Engineering

Departamento de Engenharia Eletrotécnica

Instituto Superior de Engenharia do Porto

June, 2024

This report partially satisfies the requirements defined for the Project/Internship course, in the 3^{rd} year, of the Degree in Telecommunications and Informatics Engineering.

Candidate: Tiago Adriano Mendes Ribeiro, No. 1210924,
`1210924@isep.ipp.pt`

Scientific Guidance: Isabel Azevedo, `ifp@isep.ipp.pt`

Company: ISEP
Advisor: Isabel Azevedo, `ifp@isep.ipp.pt`

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto
Rua Dr. António Bernardino de Almeida, 431, 4200–072 Porto

June, 2024

Aos meus pais, irmão, família e amigos…

Abstract

Protobuf is a data serialization format developed by Google to be faster, smaller, and simpler than other formats like XML and JSON. It's normally used with gRPC applications, but it can also be used with REST applications.

With the evolution of technology, more and more users need the ability to send and retrieve data faster and reliably, creating a need to organize, synchronize, and manage data from one machine to another. This process becomes critical when machines need to serve more than one user. Technologies like JSON and XML are used to help fulfill this purpose. Both are formats used to store information translated from data structures or object states so it can be saved or transmitted and later reconstructed. This process is called data deserialization/serialization.

This project explores Protobuf and the possible advantages of using it as a data serialization format for REST applications. An analysis is also made to the performance and reliability evaluation when an application uses JSON and when it uses Protobuf. Furthermore, contains the process of migrating the application from REST to Protobuf.

In the end, the results, like performance and payload size, are compared to conclude that Protocol Buffers (Protobuf), as the only data format to a REpresentational State Transfer (REST) application, is better than JavaScript Object Notation (JSON).

Keywords: REST, Protobuf, JSON, Data Format, Microservice, JMeter

Resumo

Protobuf é um formato de serialização de dados desenvolvido pela Google para ser mais rápido, pequeno e simples do que outros formatos, como por exemplo XML e JSON. É normalmente usado em aplicações gRPC, mas também pode ser usado em aplicações REST.

Com a evolução da tecnologia, mais e mais utilizadores necessitam de ter a capacidade de mandar e receber dados rapidamente e de forma confiável. Este processo torna-se crítico quando as máquinas precisam de servir mais que um utilizador. Tecnologias como JSON e XML são usadas para auxiliar esse propósito. Ambos são formatos usados para guardar informação traduzida de uma estrutura de dados, para que possa ser salva ou transmitida e, mais tarde, reconstruída. Este processo é chamado desserialização/serialização de dados.

Este projeto explora o Protobuf e as possíveis vantagens de usá-lo como o formato de serialização de dados em aplicações REST. Também é feita uma análise da performance e confiança de uma aplicação quando esta usa JSON e quando a mesma usa Protobuf. Para além disso, contém o processo de migração de uma aplicação de JSON para Protobuf.

No final, os resultados, como a performance e o tamanho do *payload*, são comparados concluindo que Protobuf, como o único formato de uma aplicação REST, é melhor do que JSON.

Palavras-Chave: REST, Protobuf, JSON, Formato de Dados, Microsserviço, JMeter

Contents

List of Figures

List of Tables

Listings

List of Acronyms

DEE	*Departamento de Engenharia Electrotécnica*
DTO	Data transfer object
Enum	Enumeration
gRPC	Google Remote Procedure Call
IDL	Interface Definition Language
ISEP	*Instituto Superior de Engenharia do Porto*
JSON	JavaScript Object Notation
LETI	*Licenciatura em Engenharia de Telecomunicações e Informática*
PESTA	*Projeto/Estágio*
POJO	Plain Old Java Objects
Protobuf	Protocol Buffers
REST	REpresentational State Transfer
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language
XML	eXtensible Markup Language

Chapter 1

Introduction

This chapter introduces the project done during the semester for the *Projeto/Estágio* (PESTA) curricular unit of the 3rd year of the *Licenciatura em Engenharia de Telecomunicações e Informática* (LETI), from the *Departamento de Engenharia Electrotécnica* (DEE), of the *Instituto Superior de Engenharia do Porto* (ISEP). The project focuses on the exploration and study of the use of Protobuf in REpresentational State Transfer (REST) applications.

1.1 Context

Protobuf is a data serialization format developed by Google to be faster, smaller, and simpler than other formats like eXtensible Markup Language (XML) and JSON [1]. gRPC applications usually use Protobuf, but not REST ones.

With the evolution of technology, more and more users need the ability to send and retrieve data faster and reliably, creating a need to organize, synchronize, and manage data from one machine to another. This process becomes critical when machines need to serve more than one user. Technologies like JSON and XML help fulfil this purpose. These formats store information translated from data structures or object states. Thus, it can be saved or transmitted and later reconstructed. This process is called data deserialization/serialization. Like other technologies, both have upsides and downsides [2].

Some big companies, like Netflix, Cisco, Uber, and some others [3], are already adopting Google Remote Procedure Call (gRPC) because of its benefits. Even though Protobuf is commonly utilized in gRPC applications, opting for Protobuf over JSON in REST applications is unconventional.

1.2 Project Description

Protobuf is traditionally used with gRPC, but it can also be used with REST applications. Several projects [4], including academic ones [5], have already compared the performance of gRPC and REST, but both used different serialization formats. LinkedIn recently, after using JSON for a while, adopted Protobuf to exchange data between microservices more efficiently across its platform and their open-source REST framework, Rest.Li [6]. This leaves the potential benefits of using Protobuf in REST applications largely unexplored.

1.2.1 Objectives and Research Approach

The objective of this project is to find out the possible benefits in performance, efficient serialization and reduced payload size, of using Protobuf instead of JSON as the leading choice for data interchange format in REST applications that use a microservice architecture. After choosing a REST application built using Spring-Boot that uses JSON and fills the requisites, the same will endure some tests, with the help of load balance software, to analyse the performance, effectiveness and reliability. The next step involves migrating the application from JSON to Protobuf, preserving the same functionalities. Then, maintaining the same conditions, the new application will face the same tests as the initial one. After an extended analysis of both results, it will be possible to conclude if there are any benefits to using Protobuf with REST.

1.3 Schedule

Figure 1.1 presents the tasks and how they were organized.

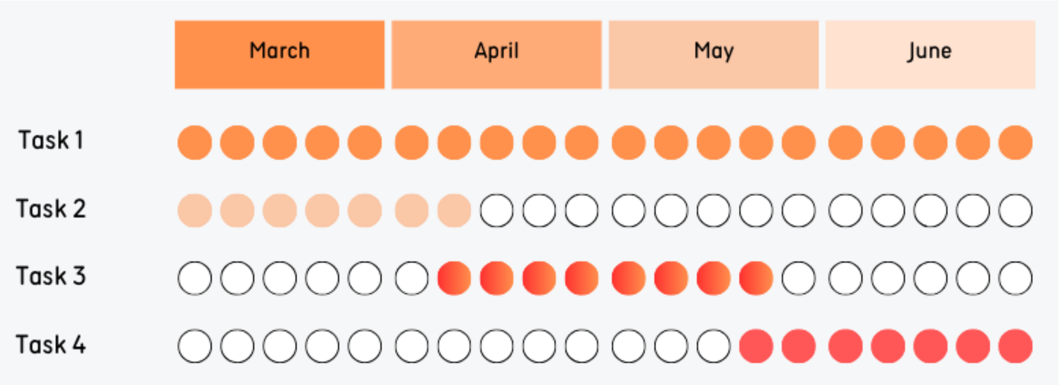


Figure 1.1: Grantt Chart

- **Task 1 -** Report Writing
- **Task 2 -** Looking for a REST Application and Reading documentation about Protobuf
- **Task 3 -** Migrating the project from JSON to Protobuf
- **Task 4 -** Testing both solutions using JMeter, comparing and discussing the results

1.4 Document Organization

This document consists of six chapters. The first chapter gives an introduction, contextualises the project and its objectives, and disposes a calendar with work plans. The second one contains a deep dive into the primary technology and other important concepts. The third chapter is about the application chosen for this project, explaining the design and how it works. The fourth one explains the steps taken to migrate the application to Protobuf. The fifth chapter presents all the results obtained by testing both projects and a discussion about them. Finally, the sixth chapter concludes the project, delivering the conclusions made by analysing the work that was done.

Chapter 2

State of the Art

This chapter presents and explores the primary technology used and other important concepts. Also introduces the principal tools used.

2.1 Data Serialization and Deserialization

Serialization is the process of converting a data object into a series of bytes that saves the state of the object into an easily transmittable form. After being serialized, the data can be delivered to another data store, like a database, file, application memory. The inverse process also exists. Deserialization process constructs an object or a data structure by taking a series of bytes, thus making the data easier to read and modify as a native structure in a programming language [7].

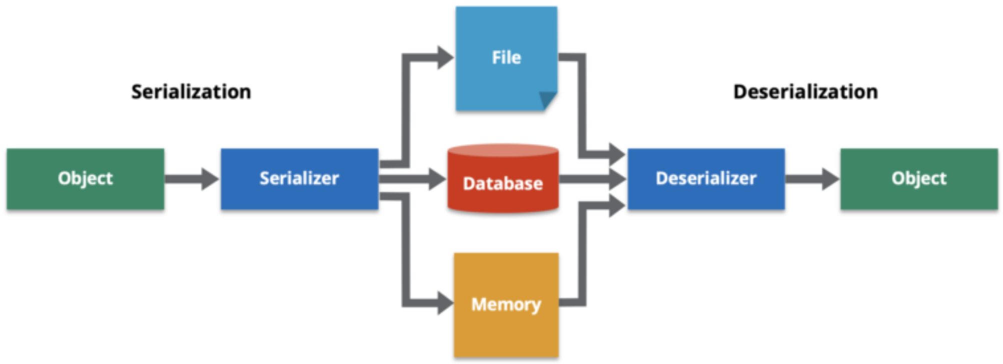


Figure 2.1: Data Serialization and Deserialization [8]

Data serialization can be done in a legion of different ways and with different formats. Which method and format to choose depends on the requirements set up by the project. This choice may affect the size of the data and the performance in which the data is serialize/deserialize and its memory use [7].

Some examples of data serialization formats are XML, JSON, Protobuf, FlatBuffers, PSON, Thrift, YAML.

In this section XML, JSON and Protobuf, will be explored with a comparison between Protobuf and JSON.

2.1.1 eXtensible Markup Language

eXtensible Markup Language (XML) is a markup language derive from Standard Generalized Markup Language (SGML) and developed by W3C. An XML document consists of some well-defined markups called tags, and the data defined within them. The tags are containers that describe the enclosed data and also organize it [9].

Some fundamental design considerations of XML include simplicity and human readability, as W3C specifies with *"XML shall be straightforwardly usable over the Internet"* and *"XML documents should be human-legible and reasonably clear"* [9]. One of the primary uses for XML is object serialization for the transfer of data between applications [10].

The code available at the listing 2.1 provides a simple example of a XML code.

```xml
<Product>
    <id>1</id>
    <name>Your Product Name</name>
    <price>12.3</price>
</Product>
```

Listing 2.1: A simple description of a product in XML

2.1.2 JavaScript Object Notation

JavaScript Object Notation (JSON) is a lightweight data-interchange format. It is easy for humans to read and write and for machines to parse and generate. It is known as the most widely used format for interchanging data on the web after XML [11]. It originates from a subset of the JavaScript Programming Language Standard ECMA-262. It is represented by using two primary data structures: ordered lists, recognized as arrays, and name/value pairs, recognized as objects [10].

The code available at the listing 2.2 provides a simple example of a JSON code.

```
{
    "id": 1,
    "name": "Your Product name",
    "price": 12.3
}
```

Listing 2.2: A simple description of a product in JSON

2.1.3 Protocol Buffers

During this document, Protocol Buffers (Protobuf) was mentioned many times, being this the top technology talked in this document. Protocol Buffers was developed by Google, and are a language-neutral, platform-neutral, extensible mechanism for serializing structured data, like XML, but smaller, faster, and simpler. Protobuf provides a format for taking compiled data and serializing it by turning it into bytes represented in decimal values. The schema description it is done in a **.proto**, then, using a proto compiler you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages [12]. Some of the languages supported by Protobuf are C++, C#, Dart, Go, Java, Kotlin, Objective-C, Python, and Ruby. With proto3, you can also work with PHP [12]. As mentioned before, Protobuf is mainly used with gRPC, but it can also be used with REST applications.

```
syntax = "proto3";
package example;

message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;
}
```

Listing 2.3: Protobuf .proto definition example

For the first line of code in the listing 2.3, we define the version, in this case, proto3, but if that field was left blank the compiler would automatically assume the version as proto2. Has mentioned before, Protobuf has different versions, proto1, proto2 and proto3. Proto1 is deprecated, proto2 and proto3 are still active. Proto3 can be viewed as a simplification of proto2. Both are wire compatible, meaning that the same construct in proto2 and proto3 will have the same binary representation. This means they can reference symbols across versions and generate code that works well together [13].

Using Protocol Buffers

This section has the purpose to explain the syntax of .proto file and how to structure Protobuf data.

Messages

The *message* is used to define the information to be exchanged between client-server, but it can be viewed as akin to classes. It is defined in the *.proto* file. When the file is set up it an be used alongside another programming language such as Java. The data is added by following this scheme: Field assignments - Type, Name, Tag.

```
message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;
}
```

Listing 2.4: Protobuf .proto message example

Services

Protobuf allows for Remote Procedure Call (RPC) and the *service* is what defines what methods a client can request remotely. The *service* accepts a request *message* and a response *message*.

```
service ProductService {
    rpc GetProduct(ProductRequest) returns (ProductResponse);
}
```

Listing 2.5: Protobuf .proto service example

Scalar Values

A message is normally composed of a number of different scalar values. As can be seen in the table bellow, Protobuf is easily used with other programming languages: C++, Java/Kotlin, Python, Go, Ruby, C#, PHP and Dart. We can easily map the data type of a programming language to Protobuf type and vice versa.

.proto Type	Notes	C++ Type	Java/Kotlin Type	Python Type	Go Type	Ruby Type	C# Type	PHP Type	Dart Type
double		double	double	float	float64	Float	double	float	double
float		float	float	float	float32	Float	float	float	double
int32	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint32 instead.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer	int
int64	Uses variable-length encoding. Inefficient for encoding negative numbers – if your field is likely to have negative values, use sint64 instead.	int64	long	int/long	int64	Bignum	long	integer/string	Int64
uint32	Uses variable-length encoding.	uint32	int	int/long	uint32	Fixnum or Bignum (as required)	uint	integer	int
uint64	Uses variable-length encoding.	uint64	long	int/long	uint64	Bignum	ulong	integer/string	Int64
sint32	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int32s.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer	int
sint64	Uses variable-length encoding. Signed int value. These more efficiently encode negative numbers than regular int64s.	int64	long	int/long	int64	Bignum	long	integer/string	Int64
fixed32	Always four bytes. More efficient than uint32 if values are often greater than 2^{28}.	uint32	int	int/long	uint32	Fixnum or Bignum (as required)	uint	integer	int
fixed64	Always eight bytes. More efficient than uint64 if values are often greater than 2^{56}.	uint64	long	int/long	uint64	Bignum	ulong	integer/string	Int64
sfixed32	Always four bytes.	int32	int	int	int32	Fixnum or Bignum (as required)	int	integer	int
sfixed64	Always eight bytes.	int64	long	int/long	int64	Bignum	long	integer/string	Int64
bool		bool	boolean	bool	bool	TrueClass/FalseClass	bool	boolean	bool
string	A string must always contain UTF-8 encoded or 7-bit ASCII text, and cannot be longer than 2^{32}.	string	String	str/unicode	string	String (UTF-8)	string	string	String
bytes	May contain any arbitrary sequence of bytes no longer than 2^{32}.	string	ByteString	str (Python 2)					
bytes (Python 3)				bytes (Python 3)	[]byte	String (ASCII-8BIT)	ByteString	string	List<int>

Table 2.1: Mapping Table for possible Scalar Types [12]

Message Field

A message field is divided by field type, check table 2.1 for available types, field name that represents a particular element within a message, when multiple words are used to identify a field, each one is separated by underscore "_" and the field tag, like shown in the figure 4.1.

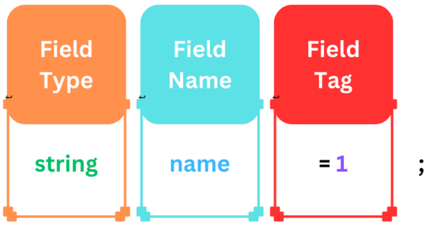


Figure 2.2: Protobuf .proto message field

Field Tags are simply a number that ranges from 1 to 536870911. However, there are some rules to this tags [12].

- The given number must be unique among all fields for that message.
- Field numbers 19,000 to 19,999 are reserved for the Protocol Buffers implementation. The Protobuf compiler will complain if you use one of these reserved field numbers in your message.
- You cannot use any previously reserved field numbers or any field numbers that have been allocated to extensions.

Field Tags also allow a way to speed up data:

- Numbers 1 to 15 only use 1 byte.
- Numbers 16 to 2047 uses 2 bytes.

A field can be *singular* or *repeated*. A *singular* field, which is the default so there is no need to write it, can only have none or exactly one value. A *repetead* field contains a array or list of values, it can be repeated any number of times, even zero.

```
message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;
  repeated int32 discounts = 4;
}
```

Listing 2.6: Protobuf .proto fields example

Enum

An Enum is used when the values are known or fixed. An Enum must start with tag 0 (default value) and all values are capitalized. Check the example below, where the Enum defines the status of the product:

```
message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;


  enum ProductStatus {
       AVAILABLE = 0;
       OUT_OF_STOCK = 1;
       DISCONTINUED = 2;
  }

  ProductStatus status = 4;

}
```

Listing 2.7: Protobuf .proto enum example

Nesting

Messages can be added inline into another message allowing message(s) types within a message type. In programming language, this functionality is well known and it is called aggregation. In the example code below the message `ProductDimensions` is nested inside the message `Product`.

```
message Product {
  int32 id = 1;
  string name = 2;
  double price = 3;


  message ProductDimensions {
       int32 lenght = 1;
       int32 height = 2;
       int32 mass = 3;
  }

  ProductDimensions dimensions = 4;
}
```

Listing 2.8: Protobuf .proto message nesting example

2.1.4 JSON vs Protobuf

Table 2.2, that was adapted from the table present in the document cited in [14], compares some data formats. Looking at Protobuf and JSON it is evident that Protobuf it is way more efficient. It compensates the lack of language support, compared to JSON, by supporting HTTP/2. Although with REST, Protobuf can only use HTTP.

Feature	**Protobuf**	**JSON**	**XML**	**PSON**	**FlatBuffers**
Format Type	Binary	Text-Based	Text-Based	Binary	Binary
Efficiency (size)	High	Medium	Low	High	High
Efficiency (speed)	High	Medium	Low	High	High
Human Readable	No	Yes	Yes	No	No
Language Support	10	13+	13+	Limited	13
Versioning Support	Yes	No	No	No	Yes
Launched Publicly	2008	2002	1996	-	2014

Table 2.2: Comparison Between Data Formats

The faster the data is transmitted throughout a network the better. Smaller data is faster to transmit. Also, there is no need for the data to be human-readable during transmission. This is where Protobuf shines. Figures 2.3 and 2.4 demonstrate a comparison between the size and time of JSON and Protobuf based on tests performed to consider encoding and decoding benchmarks and common browsers [15].

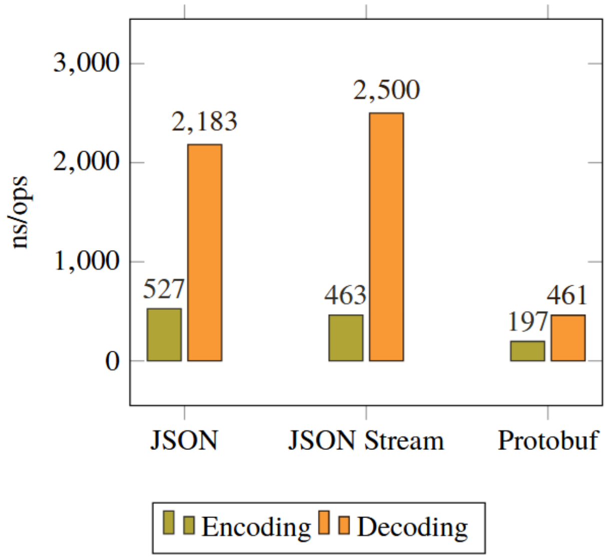


Figure 2.3: Encoding and Decoding Performance of Protobuf vs JSON [15]

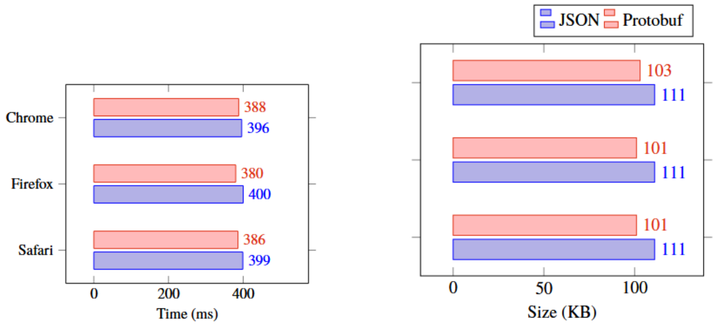


Figure 2.4: Compression Environment of Protobuf vs JSON [15]

The authors conclude that Protobuf has a significantly better performance than JSON [15]. The messages are significantly smaller and are transmitted much faster at the same time.

2.2 Microservices

Microservices is an architectural style that structures an application as a collection of services that are independently deployable and loosely coupled, where services are typically organized around business capabilities [16] [17]. Figure 2.2 represents a microservice cloud architecture.

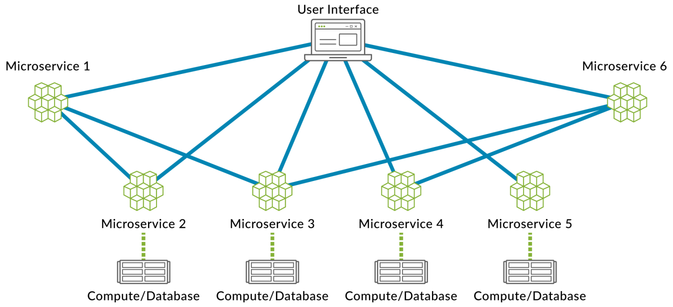


Figure 2.5: A microservice cloud architecture [18]

Over the last few years, microservices popularity has been rising due to their benefits and characteristics [16]:

- Scalability - because each service is independent, if one needs more resources it can scale and adapt with no need to do the same to the other services;

- Availability - if one service fails, the application can still run allowing it to tolerate some fails so the user experience is not disturbed;

- Elasticity - each service can grow or shrink with the progress of the work;

- Organization - because each service is independent, erases the need for a developer to know the whole application and makes it so each one has a small code base which allows for easier maintainability;
- Fault Tolerance - because everything is distributed with various systems, a unique point of failure ceases to exist.

Despite all of these benefits, microservices have some difficulties and challenges [16]:

- Complexity - each service is simple, but the whole application can be a very complex system;
- Connectivity - Because the software is divided throughout various services it creates a need to maintain a connection between them to exchange information to answer a client request. This communication can interfere with the response time, making it more prolonged;
- Compatibility - if a service is changed/updated can create an incompatibility with other services.

2.3 Google Remote Procedure Call

Google Remote Procedure Call (gRPC) is a modern open-source high-performance RPC framework that can run in any environment. It can efficiently connect services in and across data centres with pluggable support for load balancing, tracing, health checking and authentication. It is also applicable in the last mile of distributed computing to connect devices, mobile applications and browsers to backend services [**?**].

gRPC works by defining a service interface containing how the service can be consumed by the customers, what methods the customers are allowed to call remotely, what parameters and message formats to use when invoking methods, and so on. gRPC uses Protobuf as the Interface Definition Language (IDL) to define the service interface.

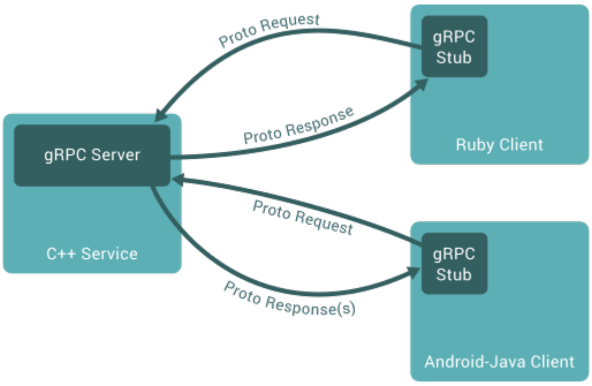


Figure 2.6: A microservice and two consumers based on gRPC [19]

2.4 REpresentational State Transfer

REpresentational State Transfer (REST) is a set of constraints that, when applied to the design of a system, creates a software architectural style. Some of the constraints that define a REST system are [20]:

- Client-Server system - this design pattern enforces the separation of concerns, which helps the client and the server components evolve independently, by giving the client the user interface concerns and the server the data storage concerns, increasing the portability of the user interface across multiple platforms and improve scalability b simplifying the server components. The server and the client can evolve but the contract between them cannot break;
- Stateless - statelessness mandates that each request from the client to the server must contain all of the information necessary to understand and complete the request, the server cannot take advantage of any previously stored context information on the server;
- Caching System - a response should implicitly or explicitly label itself as cacheable or non-cacheable, if it is cacheable, the client application gets the right to reuse the response for equivalent request;
- Uniformly Accessible - each resource must have a unique address and a valid point of access;
- Layered - it must support scalability, the system is divided by layers, where a layer must use the layer below and communicate with the layer on top;
- Code on Demand - although this constraint is optional, applications can be extendable at runtime by allowing the downloading of code on demand.

2.5 Tools and Technologies

2.5.1 JMeter

JMeter, from Apache, is a open source software designed to load test functional behaviour and measure performance. [21]

JMeter can be use to simulate a heavy load on a server, group of servers, network or object it order to test its strength or to analyze overall performance under different load types.

JMeter was used to perform load tests on both problems, allowing to obtain and save data to easily compared the results. This results can be made of samples, elapsed time, bytes, error percentage, throughput and so one.

2.5.2 Postman

Postman is an API platform for building and using APIs. Postman simplifies each step of the API lifecycle and streamlines collaboration so it can be possible to create better APIs faster [22].

As an API platform, postman has integrated tools and processes that allow teams to effectively build, manage, publish, and consume APIs. It helps producers manage the entire API life cycle.

It was mainly used to do small tests during the implementation, and to retrieve some information, like the authorization token, when performing the tests.

Chapter 3

Analysis & Design

This chapter covers the design of the chosen application and other topics.

3.1 Problem

To effectively execute alterations and verify the differences between REST JSON and REST Protobuf, a pivotal application is leveraged to achieve the desired outcome: **Subscription Management**. This application was developed in the context of the curricular unit *Sistemas Distribuídos* that belongs to the first semester of the third year of the curse LETI in ISEP and can be accessed in the public BitBucket repository [23]. All members and professors involved in the development of this project permitted to use it and make alterations. It has a free license, and meets the requirements, this means that uses REST JSON and follows the microservice architecture, check 2.2 for more information about, and it is familiar with what makes it easier to make changes and work with it.

3.1.1 Context & Analysis

ACME, Inc, is a fictional SaaS-based music streaming service that needs a service-oriented solution to manage the subscriptions of the service. So **Subscription Management** application was created. The application provides plans, some free, others paid. The plans can be managed by a marketing director, who can create new ones, edit and deactivate them. The difference between a deactivated plan and an active one is that a deactivated plan does not show as an option to subscribe neither can be subscribed to. A plan can be deactivated even if someone has an activated subscription. A user can check the available plans and register or login to an account. When creating an account the user can choose a plan, otherwise it will be assigned the free plan. After creating an account the user passes to a subscriber that as a subscription associated. A subscriber can change the plan, view the details of the plan that is associated with the subscription, renew the subscription and cancel the subscription.

Domain Model

Figure 3.1 represents the main concepts of the problem (User, Plan, Subscription) and how they are associated.

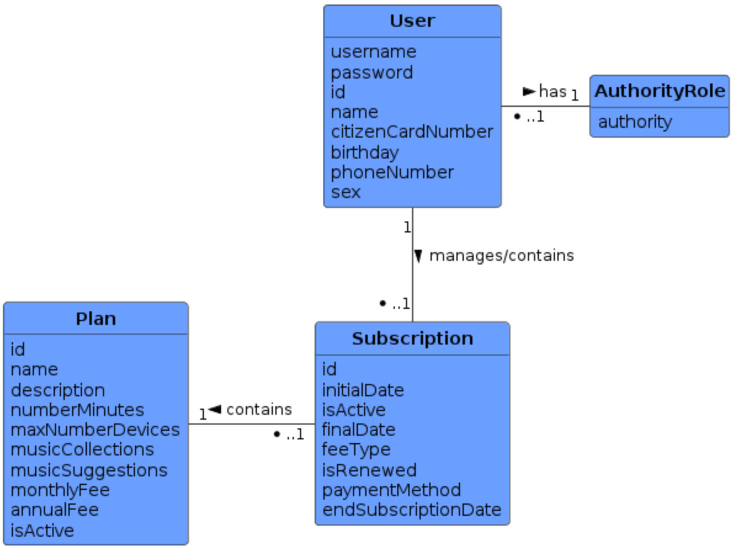


Figure 3.1: Domain Model

Use Case Diagram

Figure 3.2 presents the use case diagram, that allows to visually illustrate the available functionalities to the different users.

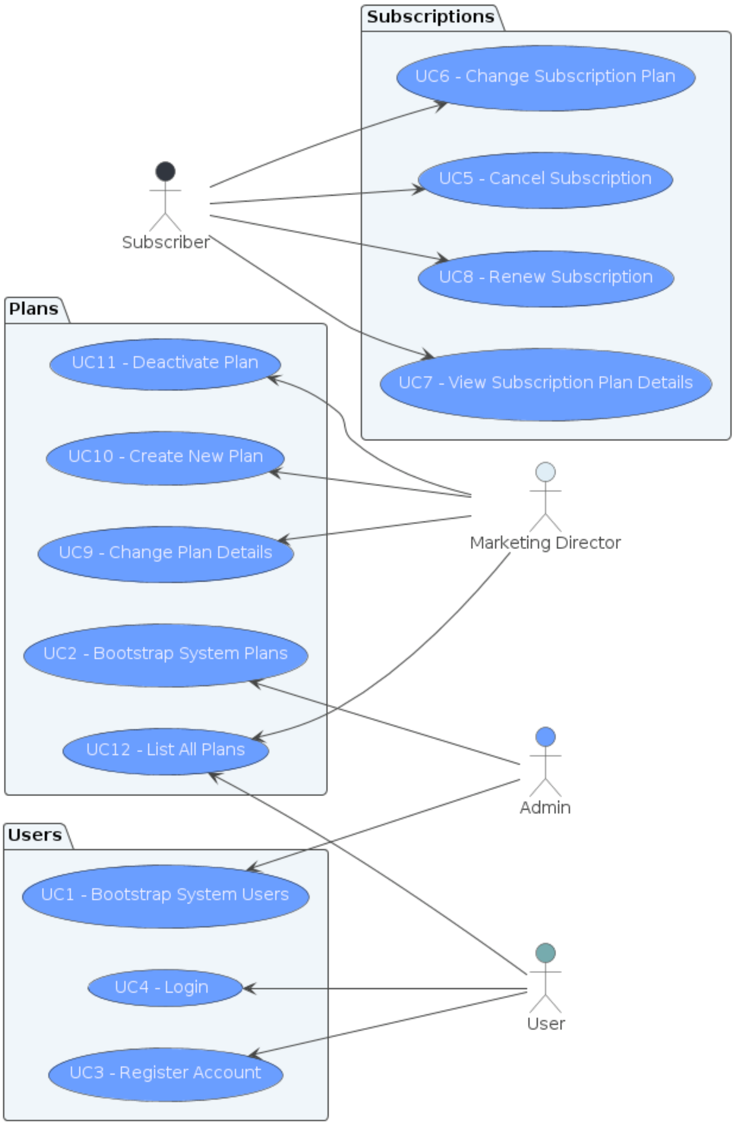


Figure 3.2: Use Case Diagram

Use Case Diagram - Extra Feature

To assist the testing stage, a new use case emerged. The use case, "UC13 - Get subscriptions by plan", allows the retrieval of all the subscriptions that have a specific plan associated to it. Figure 3.3 represents the new use case diagram.

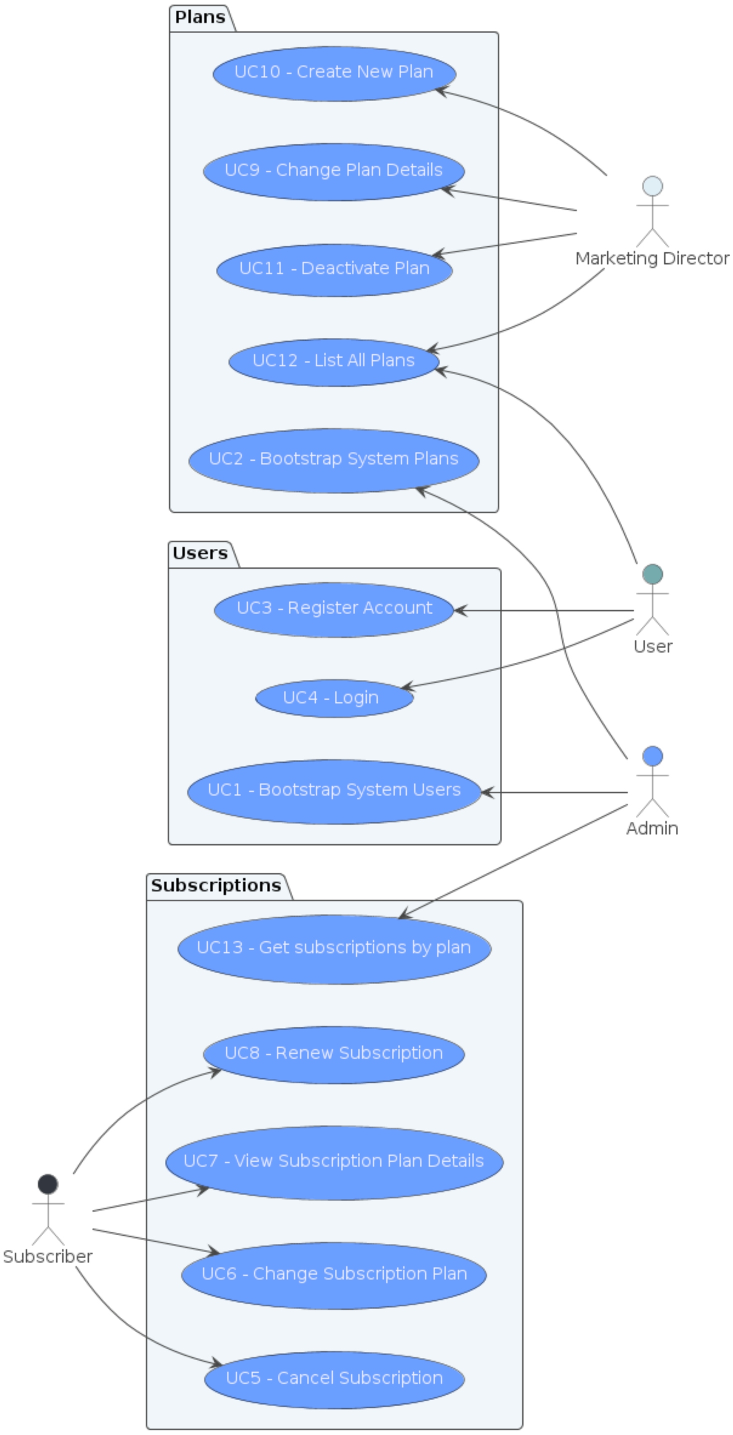

Plans
UC10 - Create New Plan
UC9 - Change Plan Details
UC11 - Deactivate Plan
UC12 - List All Plans
UC2 - Bootstrap System Plans
Marketing Director
Users
UC3 - Register Account
UC4 - Login
UC1 - Bootstrap System Users
User
Admin
Subscriptions
UC13 - Get subscriptions by plan
UC8 - Renew Subscription
UC7 - View Subscription Plan Details
UC6 - Change Subscription Plan
UC5 - Cancel Subscription
Subscriber

Figure 3.3: New Use Case Diagram

3.2 Design - Project without Gateway

3.2.1 Logical View

Figure 3.4 and figure 3.5 correspond to the component diagram level 1 (L1) and level 2 (L2), respectively. The application is divided into microservices and has two instances of each service. Each instance exposes an HTTP port for the client to send in requests. Each service has a unique database, following the database-per-service pattern. There is no duplicated data. For example, if an entity Plan is created in "Plans:P1" only that database knows of its existence. When "P2:Plan" receives a request for that entity, it needs to ask "P1:Plan" for information about it. More on that in the section 3.2.3.

Each instance communicates with the other one of their kind. Some of them also communicate with different ones. HTTP serves as the protocol of choice to establish that connection. Because they use HTTP, the address of each instance that one needs to talk to has to be known by that instance. For example, the instance "P1:Plan" needs to get information from "P2:Plan" and one of the "S?:Subscription" instances, it does not matter what each one of those kind is, so "Plans:P1" has to know the address of those instances. For that reason, this solution offers little scalability. Also, to prevent an infinite loop, when an instance requests another one, it uses different endpoints. These endpoints are exclusive to instances. The section 3.2.3 provides additional details.

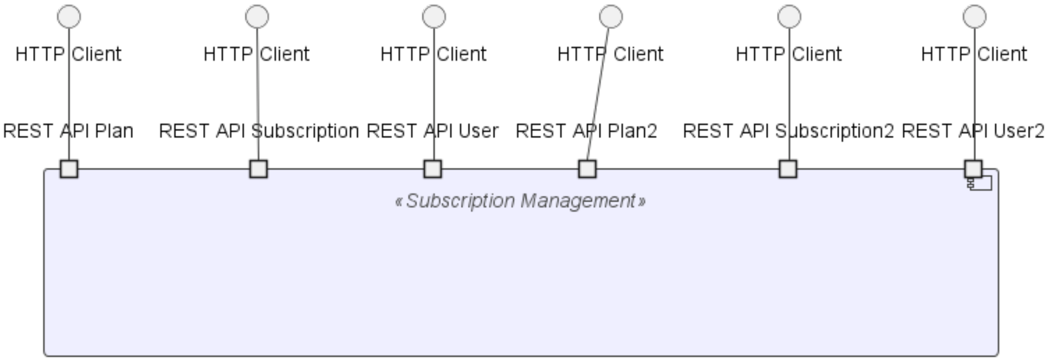


Figure 3.4: Component Diagram Level 1

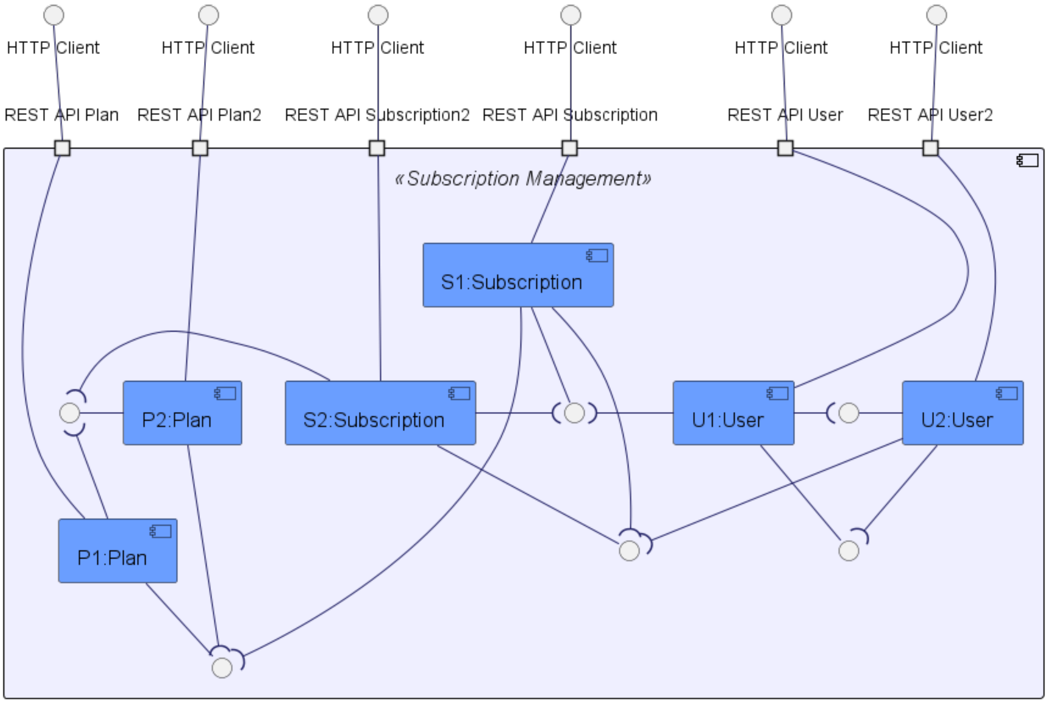


Figure 3.5: Component Diagram Level 2

3.2.2 Physical View

Figure 3.6 represents the Deployment Diagram and shows how the problem was deployed. As can be seen, every instance is working under the same machine. The databases were managed with the H2 software.

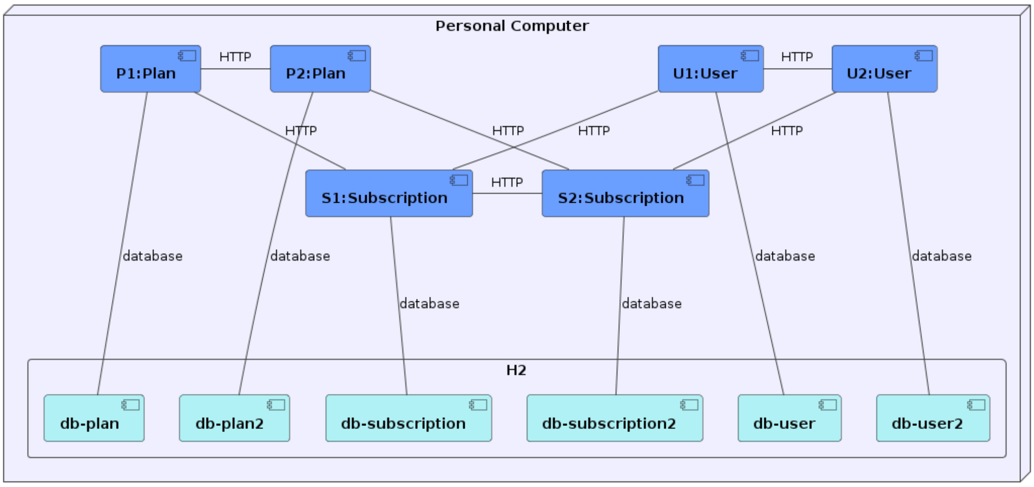


Figure 3.6: Deployment Diagram

3.2.3 Process View

Figure 3.7 and figure 3.8 are sequence diagrams level 2 from the plan service. Figure 3.9 is a sequence diagram level 2 from the subscription service. The purpose of these diagrams is to help understand the flow of the requests.

GET Plans - Endpoint to get all plans

Because there is no duplicated data, when one instance receives a request from a client, it needs to ask the other one about their data. After receiving it the instance has every data necessary and can return to the client all information, as is visible in figure 3.7.

As previously mentioned, when an instance requests another, it utilizes a distinct endpoint from the one accessible to clients. Employing the same endpoint would trigger an infinite loop, as the second instance would seek data it lacks. For instance, if "P1:Plan" received the request, it would solicit all available data from the other existent instance, "P2:Plan". Subsequently, the latter would reciprocate by requesting comprehensive data from the former, perpetuating the cycle. This scenario is averted by prefixing all endpoints with "internal", signalling to the instance to solely retrieve information from its database without redundant cross-verification.

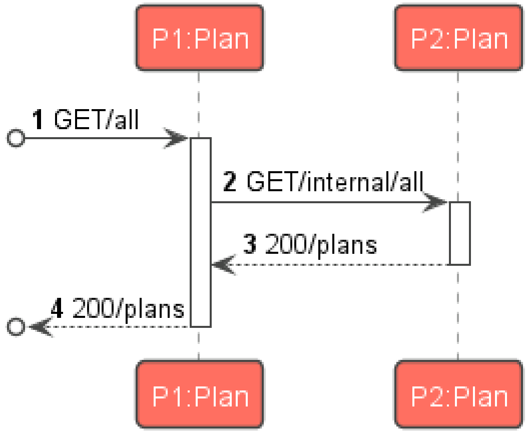


Figure 3.7: Sequence Diagram Level 2 GET All Plans

POST Plan - Endpoint to create a plan

The same thing will happen in figure 3.8, because one instance does not know if a plan exists in the other or not, so, after verifying on its database, sends a request to the other one. After both searches retrieve a null value or the HTTP code 404 - Not Found, in the case of the HTTP request, the instance can save the plan and inform the client of the success.

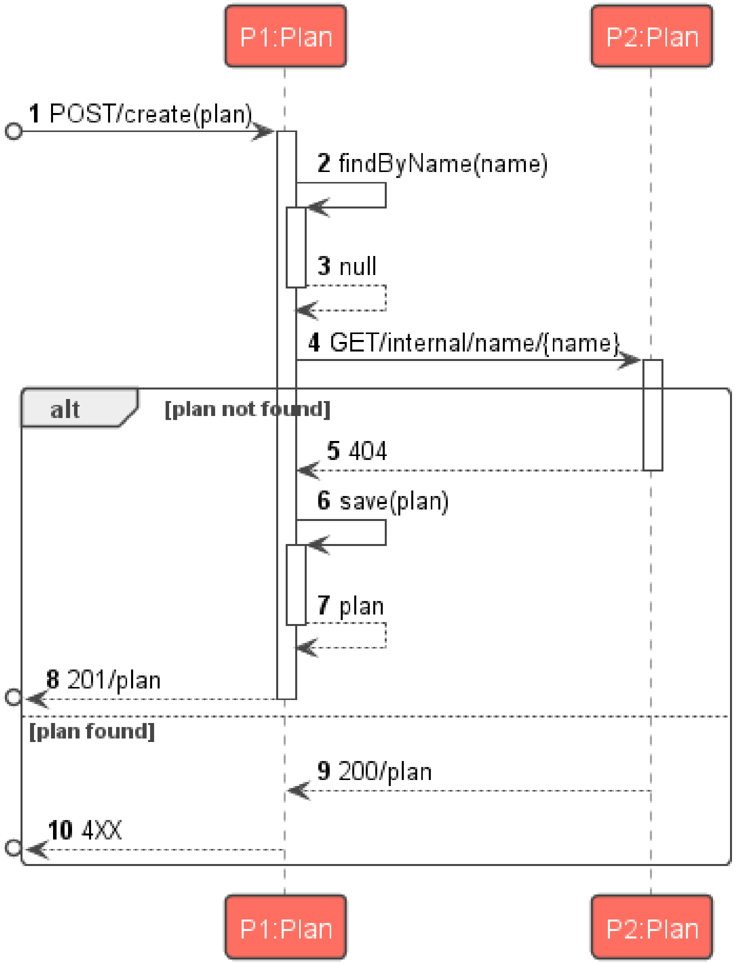


Figure 3.8: Deployment Diagram Level 2 POST Create Plan

GET UserPlanDetails - Endpoint to get the information of the plan the user is subscribed to

Figure 3.9 helps us understand the communication with other instances of different services.

As stipulated, the application extracts the username from the JWT token obtained upon user login. In the event of an invalid JWT token, the application promptly issues HTTP code 401 - Unauthorized Access. Furthermore, during subscription creation, the application verifies the validity of the plan. The response to this inquiry will consistently be either 401 or 200 - OK, providing the plan upon successful validation.

The subscription services abstain from retaining any data concerning the plans. When queried about plans, they promptly solicit information from the plan service. Given that "S1:Subscription" or "S2:Subscription" represents an instance from a distinct service compared to "P?:Plan", it employs the standard HTTP client endpoint.

Notably, the instance from the Plan service may differ as long as the requesting instance has the knowledge to talk to one of them.

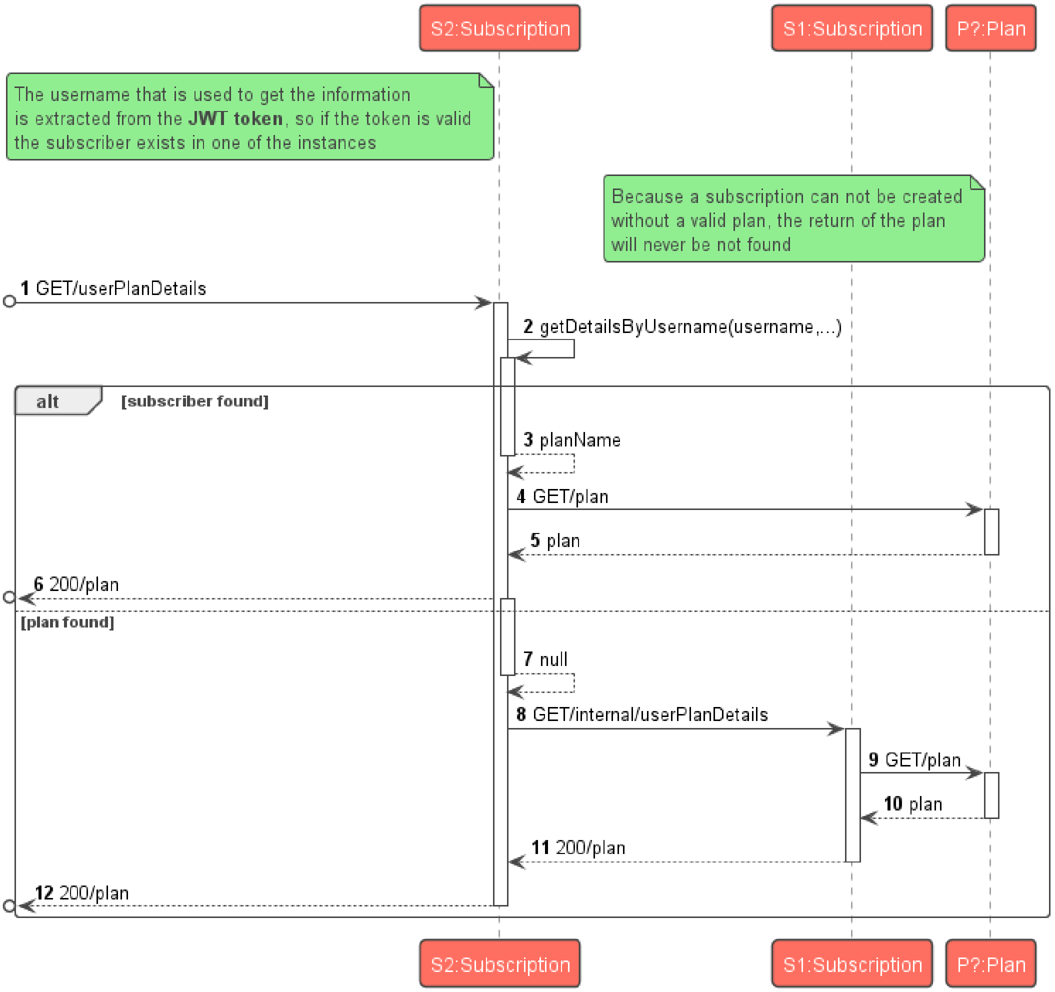

S2:Subscription
S1:Subscription
P?:Plan
The username that is used to get the information is extracted from the **JWT token**, so if the token is valid the subscriber exists in one of the instances
Because a subscription can not be created without a valid plan, the return of the plan will never be not found
1 GET/userPlanDetails
2 getDetailsByUsername(username,...)
alt
[subscriber found]
3 planName
4 GET/plan
5 plan
6 200/plan
[plan found]
7 null
8 GET/internal/userPlanDetails
9 GET/plan
10 plan
11 200/plan
12 200/plan
S2:Subscription
S1:Subscription
P?:Plan

Figure 3.9: Sequence Diagram Level 2 GET Plan Details

3.3 Design - Project with Gateway

In the modified project a gateway was established to receive the HTTP requests and route them to the microservices.

An API gateway provides a single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services. This avoids the necessity for the client to access the distinct individual services [24].

The main difference in the diagrams between the initial project and the modified one is in the format used for internal and external communication. This communication is done mainly with Protobuf.

3.3.1 Physical View

JSON Project

In comparison to the deployment diagram 3.6, the one in figure 3.10 has an extra service gateway, that is in charge of routing the requests to the proper microservice.

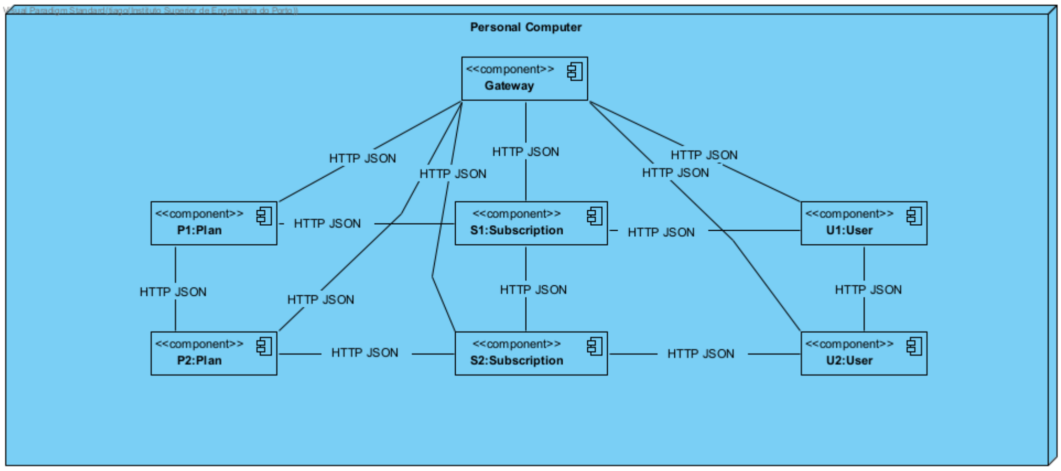

Visual Paradigm Standard(tiago(Instituto Superior de Engenharia do Porto))
Personal Computer
<<component>> Gateway
<<component>> P1:Plan
<<component>> S1:Subscription
<<component>> U1:User
<<component>> P2:Plan
<<component>> S2:Subscription
<<component>> U2:User
HTTP JSON

Figure 3.10: Deployment Diagram for the JSON Project with Gateway

Protobuf Project V1

In the deployment diagram represented by the figure 3.11, it is possible to analyse that every instance still runs in the same computer, the difference is the communication between microservices. This communication is made utilizing Protobuf. The gateway receives and responds in JSON.

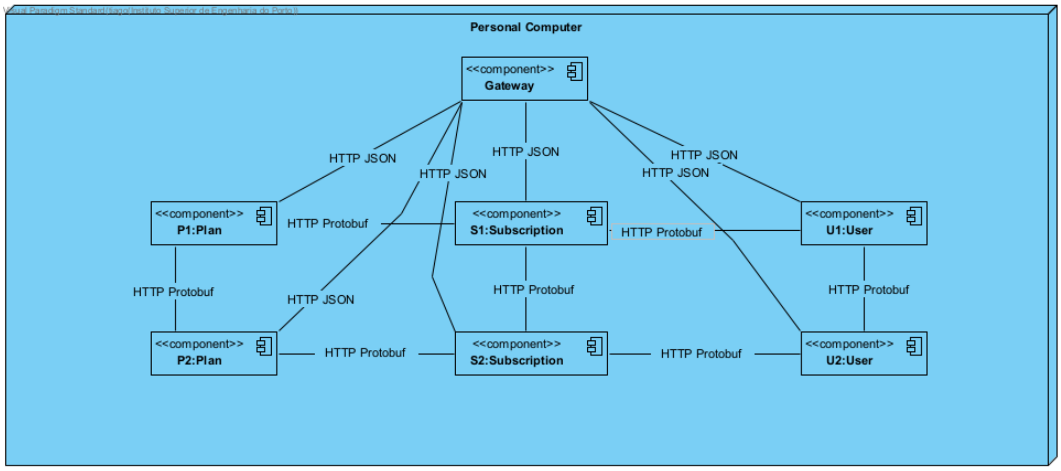


Figure 3.11: Deployment Diagram with Gateway Responding in JSON

Protobuf Project V2

In the figure 3.12, the gateway receives a request in either JSON or Protobuf and responds in Protobuf.

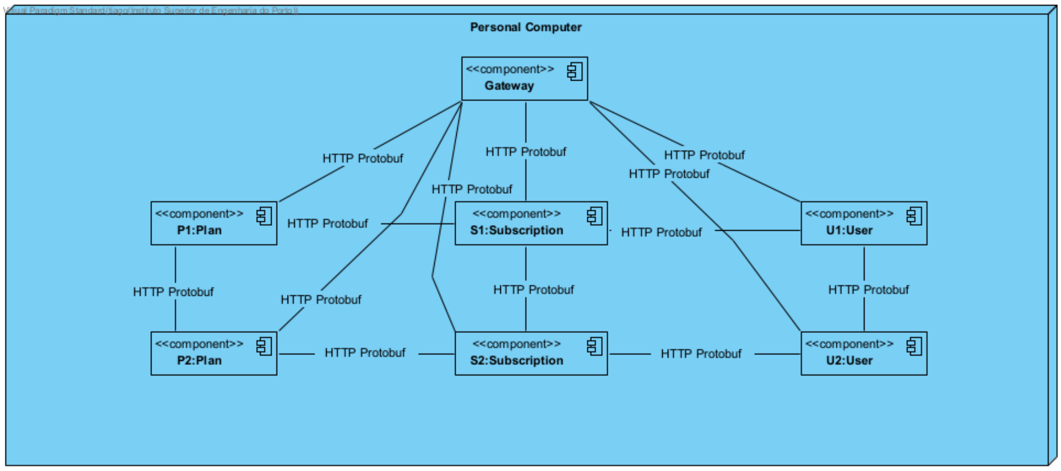


Figure 3.12: Deployment Diagram with Gateway Responding in Protobuf

3.3.2 Logical View

The logical view has changed. The connection between the instances is still the same, but now they return and receive the request through a gateway that exposes a single port for the HTTP clients. The diagram in figure 3.13 represents that change, although it was simplified to only include the microservices and not all the instances to facilitate the understating. For all projects, the diagram is the same.

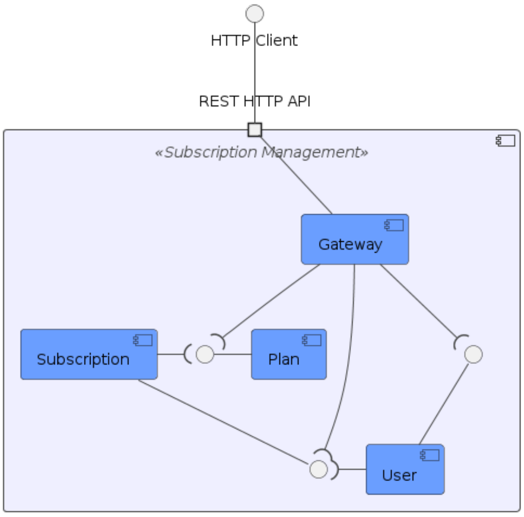


Figure 3.13: Modified Component Diagram Level 2

Chapter 4

Design & Implementation

As was refereed previously, the initial project was developed using JSON and REST following a microservice architecture, with three different microservices with each having two instances.

Throughout this chapter, it is presented the principal objective and the strategy of the migration process with the implementation of the solution. The main objective being to change the way the services interchange data between them and how clients receive it.

All the documentation and source files are publicly available at GitHub [25].

4.1 Assessment of Current System

Because the system was built using Spring Boot, when receiving a request any service can easily parse the JSON entity to a normal java class without using external libraries. However, when a service has the need to make a request, the entity to be sent or received is parsed to JSON using an external library, GSON [26], to a java class, or vice versa. Analyze the listing 4.1.

```
public ArrayList<Plan> getAllPlans() throws Exception {
    String url = this.getBaseUrl() + "/internal/all";

    try (CloseableHttpClient httpClient =
        HttpClients.createDefault()) {
        ArrayList<Plan> array = new ArrayList<>();
```

```
            HttpGet httpRequest = new HttpGet(url);
            try (CloseableHttpResponse response =
                httpClient.execute(httpRequest)) {
                Gson gson = new Gson();
                if (response.getStatusLine().getStatusCode() ==
                    HttpStatus.SC_OK) {
                    Plan[] plans =
                        gson.fromJson(EntityUtils.toString(response.getEntity()),
                        Plan[].class);
                    array.addAll(List.of(plans));
                    return array;
                }
            }
        }
        return null;
    }
```

Listing 4.1: snippet of code showing an entity being convert to an ArrayList<Plan> using GSON

With Protobuf, because it is a binary format, it can be parse to a java class using the bytes received, eliminating the need to use external libraries.

When a client wants to create, for example, a plan, he needs to send a request body. Those bodies are defined with simple java class, that use annotations to automatically create getters, setters and constructors, as visible in the listing 4.2. Those requests can be replaced with a Protobuf class. This way the message can be send in bytes and rapidly parse to a java class, possibly decreasing the elapsed time.

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class CreatePlanRequest {

    @NotNull
    @NotBlank
    @Size(min = 1, max = 32)
    private String name;

    @NotNull
    @NotBlank
    @Size(min = 0, max = 2048)
    private String description;

    @NotNull
    private int numberOfMinutes;

    @NotNull
    @NotBlank
    @Size(min = 1, max = 16)
    private String musicSuggestions;

    @NotNull
    private int musicCollections;

    @NotNull
    private float monthlyFee;

    @NotNull
    private float annualFee;
}
```

Listing 4.2: snippet of code showing a request from the Plan service

Because, when defining a .proto file it is not possible to define the size or if the variable can be null those verification's need to be done "manually".

Another key point is that the Data transfer object (DTO) class can be exchange by the class generated by the .proto file. Also, the requests, that are a simple java class visible in the listing 4.2, are also replaced.

All of the microservices utilizes DTO to transfer data between them. This is just another Plain Old Java Objects (POJO), listing 4.3, that can be replace with, once again, a Protobuf class.

```
@Data
@Schema(description = "A Plan")
public class PlanDTO {

    private String name;

    private String description;

    private int numberOfMinutes;

    private String musicSuggestions;

    private int musicCollections;

    private float monthlyFee;

    private float annualFee;

    private boolean isActive;
}
```

Listing 4.3: snippet of code showing the PlanDTO class

4.2 Migration Plan

The first step in the migration is to create all the .proto schemas, for the DTO and for the body requests, and compile them in order to obtain the Protobuf classes.

After having everything defined and compiled they need to enter in action, replacing all the POJO they are meant to replace.

With everything in place the mapper will need to be adjust in favour of working with the new compiled classes, to transform the Protobuf entity to a jpa entity.

The form how the microservices deal with information sent internally also changes, parsing the bytes array obtained in the answer to the Protobuf entity and therefore to the jpa entity, without utilising external libraries.

For the last part, before the testing to verify equality between both applications, solve all small logic problems that the migration can create.

4.3 Initial Steps

The initial approach to the project was to use a simple demo project that uses JSON and REST to explore Protobuf, understanding how to migrate from JSON to Protobuf maintaining the same functionalities and what would be the principal differences.

4.3.1 Dependencies

In the listing 4.4 it is possible to verify the dependencies added to the *pom.xml* in order to have Protobuf working correctly. Protoc is the compiler that generates the Java code from the .proto file.

```
<!-- Protocol Buffers Dependencies -->
      <dependency>
         <groupId>com.google.protobuf</groupId>
         <artifactId>protobuf-java</artifactId>
         <version>3.25.3</version>
      </dependency>

      <dependency>
         <groupId>com.google.protobuf</groupId>
         <artifactId>protoc</artifactId>
         <version>3.25.3</version>
         <type>pom</type>
```

```
     </dependency>
```

Listing 4.4: *pom.xml* Protobuf dependencies

4.4 Protobuf Schema Definition

When starting the migration process, one of the first things to do is to define our classes in the .proto file and generate the Java classes. In the listing 4.5, the plan JPA class is defined using .proto syntax. It is crucial to define a repeated `Plan` message because the application is an endpoint to return a list of plans and it is not possible to create a`List<Plan>`, making this the simpler way to return that list.

```
syntax = "proto3";
package example;
option java_package =
    "com.example.sisdi_plans.planmanagement.model.proto";

message Plan {
  string name = 1;
  string description = 2;
  int32 numberOfMinutes = 3;

  enum MusicSuggestions {
    AUTOMATIC = 0;
    PERSONALIZED = 1;
  }

  MusicSuggestions musicSuggestions = 8;

  int32 musicCollections = 4;
  float monthlyFee = 5;
  float annualFee = 6;
  bool isActive = 7;
}

message PlanList {
  repeated Plan plan = 1;
}
```

Listing 4.5: .proto Plan class definition

In order to compile this file the following command was used:

```
protoc --proto_path=./resources/proto --java_out=./java
    .\resources\proto\PlanEntity.proto
```

The java class was saved in the path defined in the .proto file, verify listing 4.5.

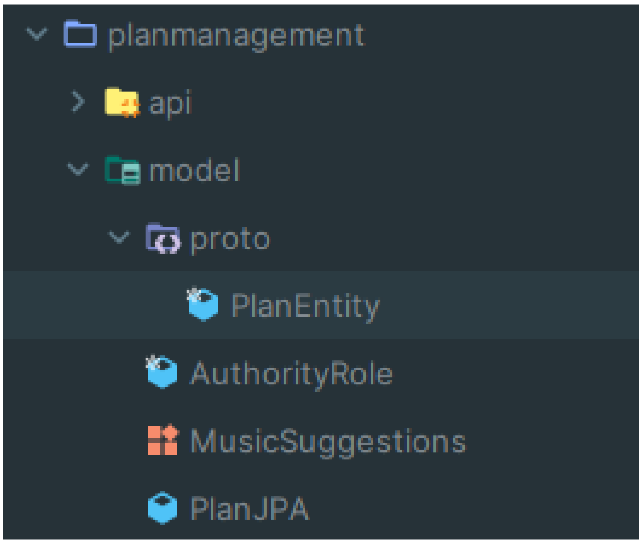


Figure 4.1: java class generated by protoc

This file contains a java class **PlanEntity** that is composed of two classes and two interfaces:

- Plan (Class)
- PlanOrBuilder (Interface)
- PlanList (Class)
- PlanListOrBuilder (Interface)

The purpose of the interface is to define methods to build the instances of the classes.

In order to have a body in a request being Protobuf a schema needs to be created to replace the old java ones, check listing 4.2. Another .proto file was created defining all the requests and then compiled to obtain the java code, analyze listing 4.6.

```
syntax = "proto3";
package example;
option java_package =
    "com.example.sisdi_plans.planmanagement.api.proto";

message EditPlanRequest {
  string description = 1;
  string numberOfMinutes = 2;
  string musicSuggestions = 3;
  string musicCollections = 4;
}
```

```

message CreatePlanRequest {
  string name = 1;
  string description = 2;
  int32 numberOfMinutes = 3;
  string musicSuggestions = 4;
  int32 musicCollections = 5;
  float monthlyFee = 6;
  float annualFee = 7;
}
```

Listing 4.6: .proto requests definitions

These steps were done for every microservice, access the full code in the appendix or in the GitHub [25].

4.5 Code Alterations

4.5.1 Spring Boot Application

There is a message converter that automatically converts a JSON body to a Protobuf message. This allows for the information to be sent in bytes or with JSON, maintaining backwards compatibility.

This converter is initialized in the main application code, verify listing 4.7.

```
@SpringBootApplication
public class PlansApplication {

   @Bean
   RestTemplate restTemplate(ProtobufHttpMessageConverter hmc) {
      return new RestTemplate(Arrays.asList(hmc));
   }

   @Bean
   ProtobufHttpMessageConverter protobufHttpMessageConverter() {
      return new ProtobufHttpMessageConverter();
   }

   public static void main(String[] args) {
      SpringApplication.run(PlansApplication.class, args);
   }

}
```

Listing 4.7: Protobuf message converter

Having the body from the requests converted to a Protobuf message we have the information parsed and ready to be transform to a java class in order to be saved in a JPA repository, for example.

4.5.2 Models

Having all the Protobuf classes generated, the mapper and the controller need to change to incorporate them. Parsing the generated class to the JPA entity or vice versa is a relatively simple transition. Using the builder interface it is easy to construct a new object or obtain information from an existing one. The biggest caviar was in the Enum. Because of the way they were created they store values as a string and not was a number, but the Protobuf class only has int numbers. So, to facilitate this process a simple extract from the value was implemented in all Enum, as visible in the listing 4.8.

```java
public enum MusicSuggestions {
   AUTOMATIC("Automatic"),
   PERSONALIZED("Personalized");

   private final String description;

   MusicSuggestions(String description) {
      this.description = description;
   }

   public String getDescription() {
      return description;
   }

   public static MusicSuggestions fromString(String text) {
      if (text != null) {
         for (MusicSuggestions musicSuggestions :
             MusicSuggestions.values()) {
            if (text.equalsIgnoreCase(musicSuggestions.description)) {
               return musicSuggestions;
            }
         }
      }
      throw new IllegalArgumentException("Invalid Music Suggestions: "
          + text);
   }

   public static MusicSuggestions fromInt(int value) {
      switch (value){
         case 0:
            return MusicSuggestions.AUTOMATIC;
         case 1:
            return MusicSuggestions.PERSONALIZED;
         default:
            return null;
      }
   }

   public static int toInt(MusicSuggestions musicSuggestions) {
      switch (musicSuggestions){
         case AUTOMATIC:
            return 0;
         case PERSONALIZED:
            return 1;
         default:
            return -12;
      }
   }

   @Override
   public String toString() {
      return getDescription();
   }
}
```

Listing 4.8: MusicSuggestions Enum alteration

4.5.3 Mappers

With a way to extract the Enum from the object, parsing was just simple get values and set values for new objects. For creating a list of plans, because the Protobuf message, `PlanList`, as a repeated message `Plan` a normal java `List<Plan>` can be easily stored in `PlanList`, as described in the listing 4.9.

```

    // Protobuf entity to JPA entity
    public PlanJPA toJPAEntity(Plan dto) {
        String name = dto.getName();
        String description = dto.getDescription();
        MusicSuggestions musicSuggestions =
            MusicSuggestions.fromInt(dto.getMusicSuggestionsValue());
        int musicCollections = dto.getMusicCollections();
        float annualFee = dto.getAnnualFee();
        float monthlyFee = dto.getMonthlyFee();
        int numberOfMinutes = dto.getNumberOfMinutes();
        PlanJPA jpaEntity = new PlanJPA(
            name,
            description,
            numberOfMinutes,
            musicSuggestions,
            musicCollections,
            monthlyFee,
            annualFee
            );
        jpaEntity.setActive(dto.getIsActive());
        return jpaEntity ;
    }

    // JPA entity to Protobuf entity
    public Plan toDTOEntity(PlanJPA jpaEntity) {
        return Plan.newBuilder()
                .setName(jpaEntity.getName())
                .setDescription(jpaEntity.getDescription())
                .setIsActive(jpaEntity.getIsActive())
                .setMusicCollections(jpaEntity.getMusicCollections())
                .setAnnualFee(jpaEntity.getAnnualFee())
                .setMonthlyFee(jpaEntity.getMonthlyFee())
                .setMusicSuggestionsValue(
                    MusicSuggestions.toInt(jpaEntity.getMusicSuggestions())
                )
                .setNumberOfMinutes(jpaEntity.getNumberOfMinutes())
                .build();
    }

    // List<Plan> from a List<PlanJPA>
    public List<Plan> toDTOEntityList(List<PlanJPA> jpaList) {
        List<Plan> dtos = new ArrayList<>();
        for(PlanJPA jpa: jpaList) {
            dtos.add(toDTOEntity(jpa));
        }
        return dtos;
    }

    // PlanList from a List<Plan>
    public PlanList toDTOPlanList(List<Plan> dtos) {
        return PlanList.newBuilder()
                .addAllPlan(dtos)
                .build();
    }
```

Listing 4.9: PlanDTOMapper

4.5.4 Internal Communication

As mentioned before, when two instances made an internal request they used an external library to parse the entity received, shown in listing 4.1. With the ability to instantiate an object from an array of bytes, this process is streamlined. Verify listing 4.10.

```java
public List<PlanJPA> getAllPlans() throws Exception {
    String url = this.getBaseUrl() + "/internal/all";

    try (CloseableHttpClient httpClient =
        HttpClients.createDefault()) {
        HttpGet httpRequest = new HttpGet(url);
        try (CloseableHttpResponse response =
            httpClient.execute(httpRequest)) {
            if (response.getStatusLine().getStatusCode() ==
                HttpStatus.SC_OK) {
                // transform the PlanList to a List<PlanJPA>
                return mapper.toJPAEntityList(mapper.toDTOEntityList(
                    // parse the bytes array to a PlanList
                    PlanList.parseFrom(
                        // convert the response to bytes
                        EntityUtils.toByteArray(response.getEntity()))
                        )
                    );
            }
        }
    }
    return null;
}
```

Listing 4.10: snippet of code showing an entity being convert to an List<PlanJPA> using a bytes array

Because the message converter can convert JSON to Protobuf there is no change in the endpoint. Basically, everything works as it should and the new application is backwards compatible. When returning a request the message comes in binary, but if in the controller the content type is defined as "application/json" the converter automatically parses the binary message to JSON, but this process may slow down the overall performance.

4.5.5 Gateway

A simple gateway was design and deployed in order to route out request to the correct microservice. Spring Cloud Gateway provides an easy way to deploy a gateway. The listing 4.11 display the needed dependencies to have a simple gateway.

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-webflux</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
```

Listing 4.11: gateway dependencies

JSON Project

With Spring Cloud gateway it is easy to define a simple route locator. Because the requests only need to be redirected the only code necessary is the one visible in 4.12. As it is possible to verify, the requests will only be sent to one instance. This problem could be solved with a load balancer, for example, Spring Cloud load balancer. Despite trying, that load balancer did not work with the static IPs. Because it did not seem to affect the overall test, since every project would lack the same thing, a load balancer was not established.

```
@SpringBootApplication
public class GatewayApplication {

    public static void main(String[] args) {
     SpringApplication.run(GatewayApplication.class, args);
   }

   @Bean
   public RouteLocator myRoutes(RouteLocatorBuilder builder) {
     return builder.routes()
           .route("plan-service", r -> r.path("/api/plans/**")
                .uri("http://localhost:8080"))
           .route("subscription-service", r ->
               r.path("/api/subscriptions/**")
                .uri("http://localhost:8081"))
           .route("user-service", r -> r.path("/api/public/**")
                .uri("http://localhost:8083"))
           .build();
   }
}
```

Listing 4.12: JSON gateway code

Protobuf Project

The gateway for Protobuf project had two versions. One that made sure Protobuf was only used in the internal connections of the project and another that permitted external communications as well.

V1

The gateway received the information in JSON and responded in JSON. To be able to do that the gateway was responsible for converting the Protobuf bodies to JSON. The conversion between the body is done with the same converter referred to before, `ProtobufHttpMessageConverter`. Also, with it a Protobuf message can be easily converted to JSON by accepting/setting the content type of the response as "application/json". With a custom gateway filter, every route can return the content type as indicated instead of "application/x-protobuf". Because an endpoint returns bytes instead of a Protobuf message, direct conversion from bytes to aJSON string is needed, as visible in the listing **??**.

```
...
    .route("plan-service", r -> r.path("/api/plans/**")
           .filters(f ->
                  f.addRequestHeader("Content-Type",
                      "application/x-protobuf")
                         .filter(modifyResponseContentTypeFilter.apply(new
                             ModifyResponseContentTypeFilter.Config()))

           )
           .uri("http://localhost:8080")
    )

    .route(r -> r.path("/api/subscriptions/userPlanDetails")
           .filters(f ->
                  f.addRequestHeader("Content-Type",
                      "application/x-protobuf")
                         .modifyResponseBody(byte[].class,
                             String.class,
                             MediaType.APPLICATION_JSON_VALUE,
                                (exchange, bytes) -> {
                                    if (bytes == null) {
                                        return Mono.empty();
                                    }
                                    PlanEntity.Plan plan = null;
                                    try {
                                        plan =
                                            PlanEntity.Plan.parseFrom(bytes);
                                        plan.toByteArray();
                                    } catch
                                        (InvalidProtocolBufferException
                                        e) {
                                        return Mono.error(new
                                            RuntimeException("Failed to
                                            parse PlanEntity.Plan from
                                            byte array", e));
                                    }
                                    String json;
                                    try {
                                        json =
                                            JsonFormat.printer().print(plan);
                                    } catch
                                        (InvalidProtocolBufferException
                                        e) {
                                        return Mono.error(new
                                            RuntimeException("Failed to
                                            convert PlanEntity.Plan to
                                            JSON", e));
                                    }
                                    return Mono.just(json);
                                }
                        )
            )
            .uri("http://localhost:8081")
    )
...


// Custom Gateway Filter
@Component
public static class ModifyResponseContentTypeFilter extends
    AbstractGatewayFilterFactory<ModifyResponseContentTypeFilter.Config>
    {

    public ModifyResponseContentTypeFilter() {
        super(Config.class);
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            exchange.getRequest().mutate().header(HttpHeaders.ACCEPT,
                "application/json").build();
            return chain.filter(exchange);
        };
    }

    public static class Config {
    }
}
```

Listing 4.13: Protobuf v1 gateway code

V2

With external Protobuf communication the only real change, in comparison to the JSON project, is setting the request content type header was "application/x-protobuf" after converting the body from JSON, as visible in the listing 4.14.

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
   return builder.routes()

        .route("plan-service", r -> r.path("/api/plans/**")
              .filters(f ->
                   f.addRequestHeader("Content-Type",
                       "application/x-protobuf")
              )
              .uri("http://localhost:8080")
        )
        .route(r -> r.path("/api/subscriptions/**")
              .filters(f ->
                   f.addRequestHeader("Content-Type",
                       "application/x-protobuf")
              )
              .uri("http://localhost:8081")
        )
        .route(r -> r.path("/api/public/**")
              .filters(f ->
                    f.addRequestHeader("Content-Type",
                        "application/x-protobuf")
              )
              .uri("http://localhost:8083")
        )
        .build();
}
```

Listing 4.14: Protobuf v2 gateway code

Chapter 5

Tests

This chapter covers all the tests performed on both solutions, including Protobuf with external communications using Protobuf. The main application used was JMeter.

5.1 Tests Configuration

In order to obtain various results there were performed four types of tests:

- Baseline - to verify how would the application work with minimal load
- Load - to analyze how the application would handle a little more load
- Stress - to verify how would the application react to an overwhelming load
- Soak - to test how would the application react a scaling load

Both versions of the project underwent the same tests. Each type of test used the same HTTP endpoints, except the soak test. Each type of test did the same request, and the only change was the load on each one. Changing the amount of Threads sending requests and the amount of time the tests underwent.

To create some data and at the same time save some information about the performance of the applications, they underwent a simple test, check table 5.1 for configuration, that had the purpose of creating subscriptions and plans.

	Creating Data
Threads	335
Ramp-up (s)	10
Loop Count	4

Table 5.1: JMeter Configuration for Creating Data

Because it is more probable that a GET request is used than another method, the emphasis of this tests focus on the performance of the projects to an overload on the queries.

Two main different endpoints were used for the tests with various purposes, for a better assessment of the results:

- GET Plan - received by the first instance of the Plans microservice, it retrieves all plans, meaning it needs to ask the second instance for their database information. This is perfect for analyzing the difference between the bytes received.
- GET Specific Plan - sent to the second Subscription microservice, it asks for the plan information that the user has associated with its subscription. Because the jwt token was from a user from the first microservice, the second one has to ask the first one for the information. The plan in question is saved on the database of the second instance, so the first one will need to retrieve that information. Because the request passes through various stages before returning something, it helps simulate a situation where four instances are working for a single request.

For creating data the endpoints the following endpoints were used:

- POST Subscriber
- POST Plan

The soak test had two extra endpoints:

- POST Login
- GET Subscriptions by Plan

In JMeter Thread Group the configuration available to be used:

- **Number of Threads (Users)** - this number represents the number of users connected to the server and realize requests
- **Ramp-up period (seconds)** - the number of seconds it takes to initialize all threads

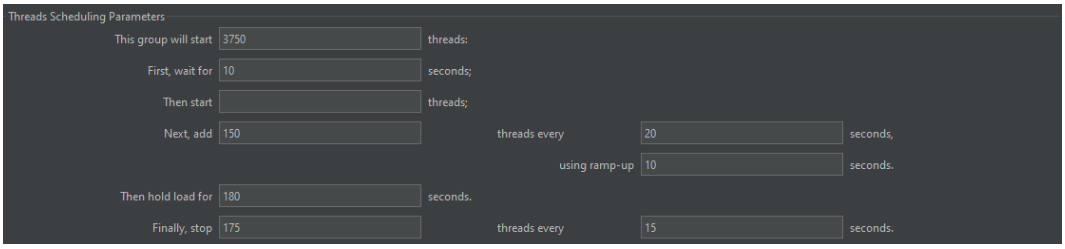
- **Duration (seconds)** - the duration that the tests run

The table 5.2 retracts the configuration for each thread group (test).

	Baseline	**Load**	**Stress**
Threads	10	600	3000
Ramp-up (s)	5	30	100
Duration (s)	180	360	600

Table 5.2: JMeter Configuration

Soak Test is a different thing. It makes use of a plugin that allows JMeter to escalate the thread group. The figure 5.1 represents the configuration. The total number of threads defined is 3750. After 10 seconds of the start of the test, JMeter initializes 0 threads and adds 150 every 20 seconds. These 150 threads will take 10 seconds to ramp up. When the full 3750 threads are up and running, JMeter holds them for 180 seconds. In the end, they stop 175 threads every 15 seconds.



Threads Scheduling Parameters

This group will start 3750 threads;

First, wait for 10 seconds;

Then start threads;

Next, add 150 threads every 20 seconds,

using ramp-up 10 seconds.

Then hold load for 180 seconds.

Finally, stop 175 threads every 15 seconds.

Figure 5.1: JMeter Soak Test Configuration

The graph represented in figure 5.2 demonstrates the relation between the elapsed time and the number of active threads count.

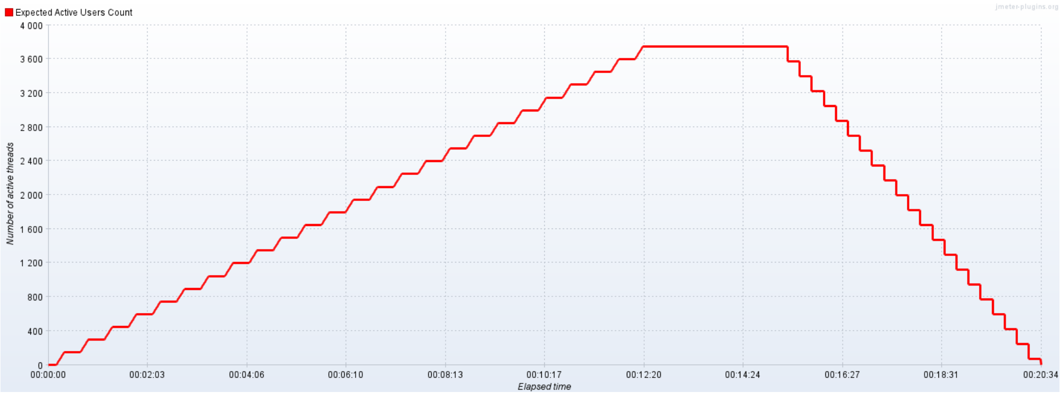
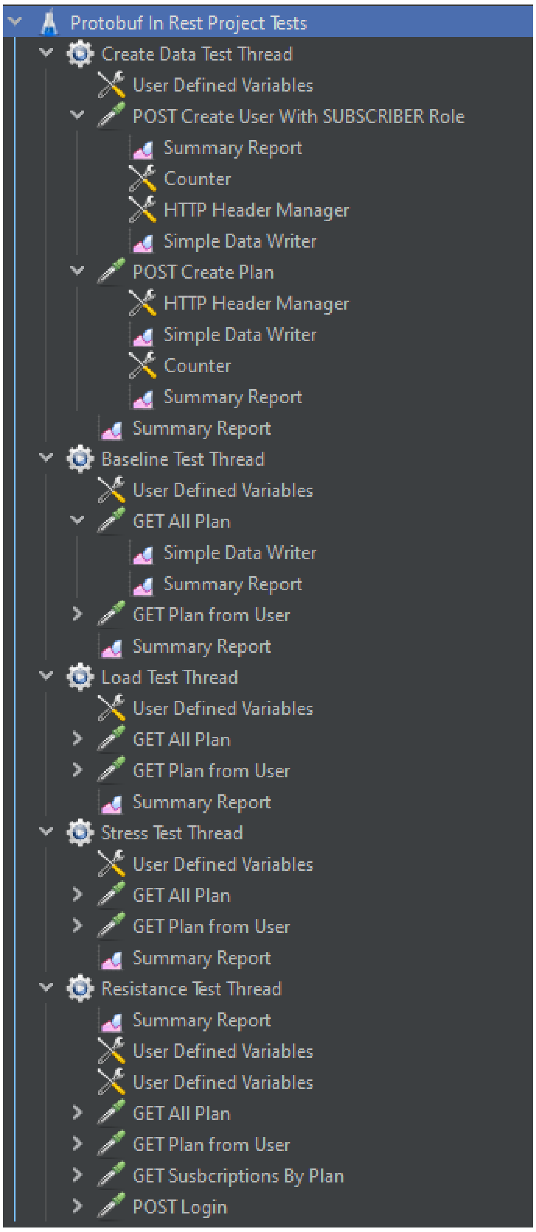


Figure 5.2: JMeter Soak Test Graph

Has visible in figure 5.3, each thread group has the same four "HTTP Request". Some of the "HTTP Request" had a "HTTP Header Manager" to add the content type and the authorization token. With the "Counter" it is possible to increase a variable value, this is used to change the unique identifier in the "POST" methods so it does not result in error. "Summary Report" generates a table with some information. "Simple Data Writer" writes all information to a .csv file for further analyzes.



```
˅ Protobuf In Rest Project Tests
    ˅ Create Data Test Thread
        User Defined Variables
        ˅ POST Create User With SUBSCRIBER Role
            Summary Report
            Counter
            HTTP Header Manager
            Simple Data Writer
        ˅ POST Create Plan
            HTTP Header Manager
            Simple Data Writer
            Counter
            Summary Report
        Summary Report
    ˅ Baseline Test Thread
        User Defined Variables
        ˅ GET All Plan
            Simple Data Writer
            Summary Report
        > GET Plan from User
        Summary Report
    ˅ Load Test Thread
        User Defined Variables
        > GET All Plan
        > GET Plan from User
        Summary Report
    ˅ Stress Test Thread
        User Defined Variables
        > GET All Plan
        > GET Plan from User
        Summary Report
    ˅ Resistance Test Thread
        Summary Report
        User Defined Variables
        User Defined Variables
        > GET All Plan
        > GET Plan from User
        > GET Susbcriptions By Plan
        > POST Login
```

Figure 5.3: JMeter Organization Tree

5.2 JMeter Tests Results - With Gateway

Throughout this subsection, the pink bar represents the Protobuf project that used JSON as the external format. the blue bar for the JSON project and the salmon bar for the other version of the Protobuf project. Also the Protobuf project will be referred to as ProtoPj, the second version as ProtoV2Pj and the JSON as JsonPj. The tests are organised by the order in which they were executed.

5.2.1 Create Data Tests

The tests performed in this section had the objective of analyse the performance of the projects when it comes to handle a post request. It also created data for later tests.

POST Subscriber

The graphs in figure 5.4 and 5.5 describe the elapsed time and the throughput performance during the creation of a new account as a subscriber. The mean value of the elapsed time is almost the same for everyone, but the smaller is JsonPj. Despite having a smaller mean, it had the highest maximum value of the three. For a few decimal values it had a higher throughput, followed by the ProtoV2Pj project.

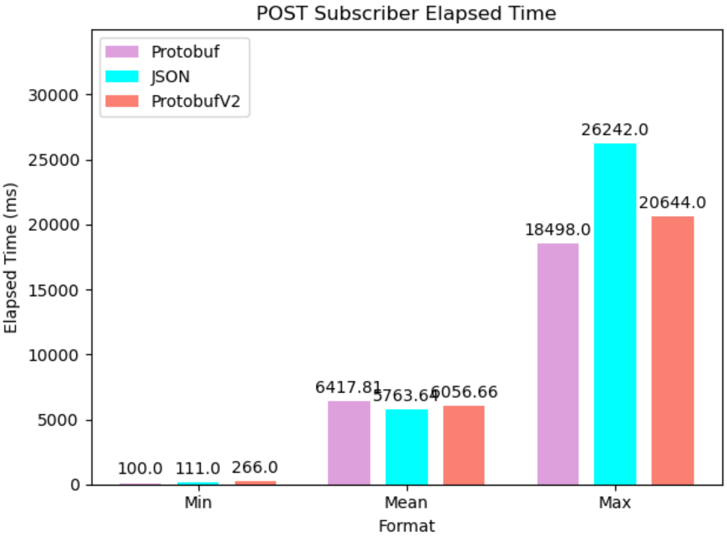


Figure 5.4: POST Subscriber Elapsed Time

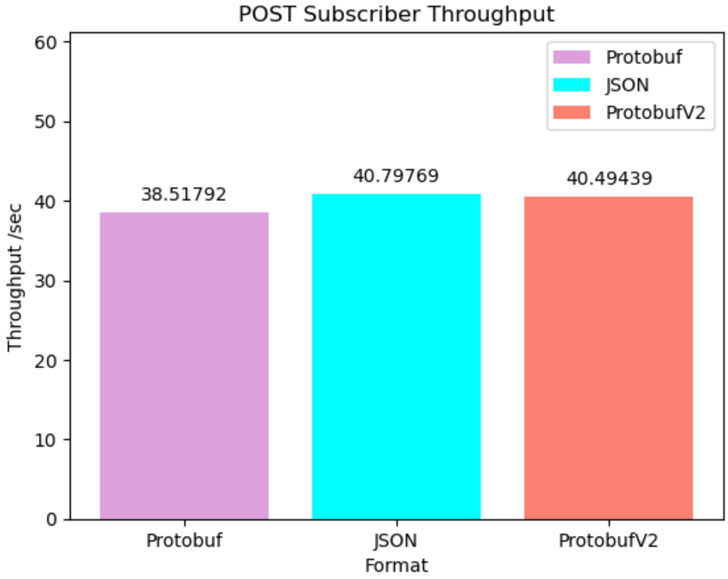


Figure 5.5: POST Subscriber Throughput

POST Plan

The graphs in figure 5.6 and 5.7 describe the elapsed time and the throughput performance during the creation of a new plan made by a marketing director. For the throughput, the test follows the same pattern, the JsonPj project has more but only by a few decimal values, then the ProtoV2Pj and last, not by a lot, ProtoPj. For the elapsed time, ProtoV2Pj has the smallest mean, and the second smallest max, losing by +/- 100 ms in comparison to JsonPj.

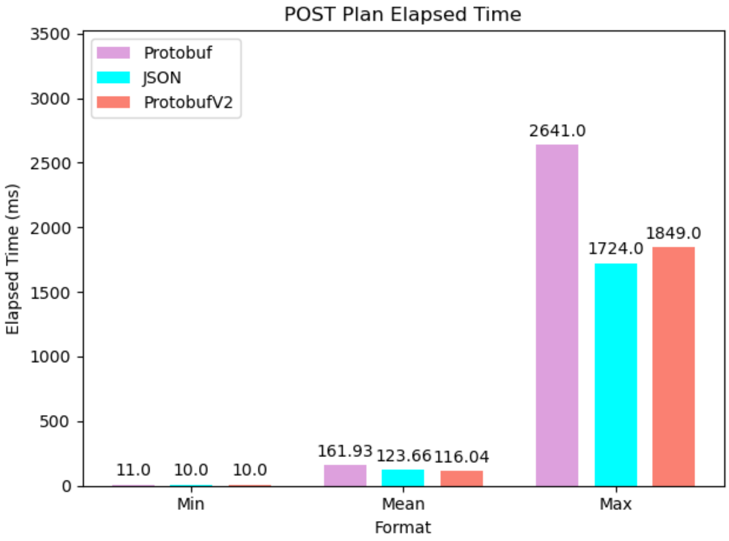


Figure 5.6: POST Plan Elapsed Time

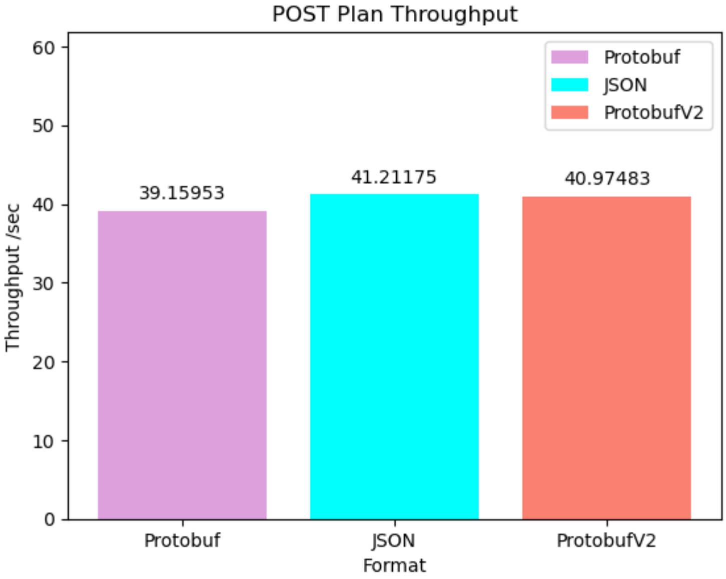


Figure 5.7: POST Plan Throughput

5.2.2 Baseline Tests

Baseline tests have the intuit to evaluate a system's initial performance and characteristics. It serves as a reference point, establishing a benchmark for future comparisons [27].

GET Plan

With the creation of a dataset with the same number of plans for each project, the tests can return enough information to compare the performance and the payload size in bytes.

The graphs in figure 5.8 and 5.9 describe the elapsed time and the throughput performance during queries for all the plans. ProtoV2Pj has a higher throughput than JsonPj and ProtoPj. Despite the difference being only +/- 7 for JsonPj, ProtoPj stays in last with +/- 40 less than ProtoV2Pj, a big difference. Elapsed time mean shows that it took almost the double of the time for ProtoPj to finish a request in comparison to the others. For the min and max values ProtoPj had the worst, ProtoV2Pj and JsonPj had almost the same, being that JsonPj had slightly better values.

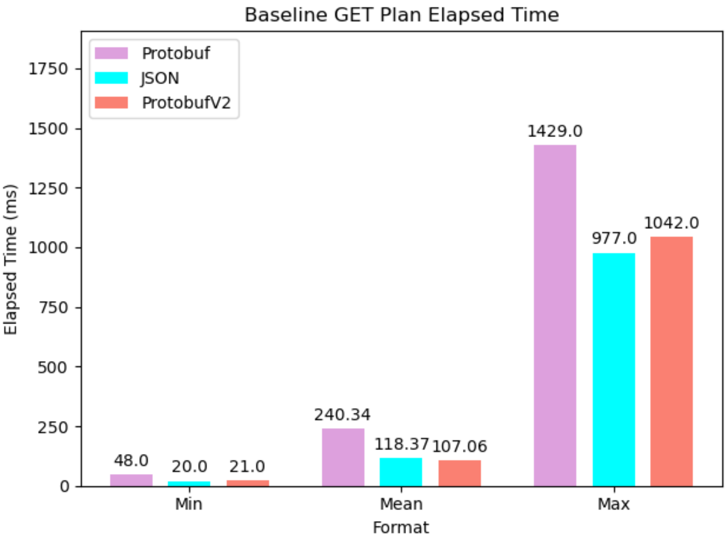


Figure 5.8: Baseline GET Plan Elapsed Time

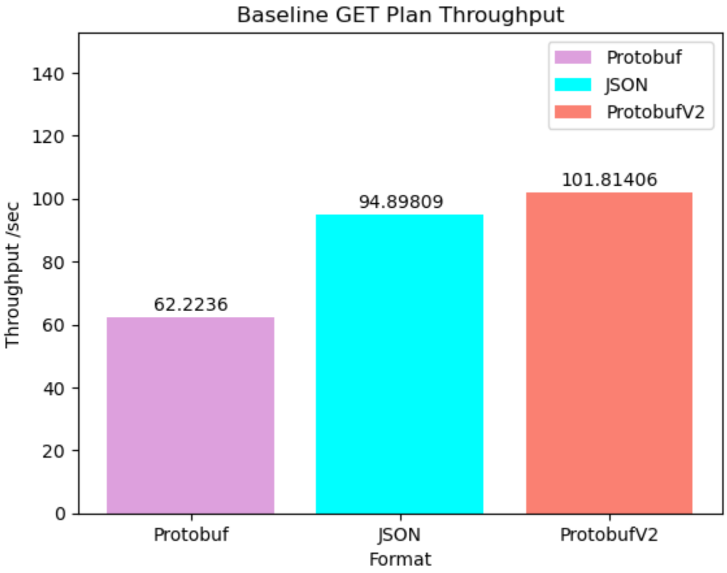


Figure 5.9: Baseline GET Plan Throughput

The graph in figure 5.10 exhibit the amount of bytes registered for each project. As expected ProtoV2Pj had the smallest value, less than half of the highest value. ProtoPj had a smaller value than JsonPj but that may be because of the way they format the response, ProtoPj has a little bit less information than JsonPj.

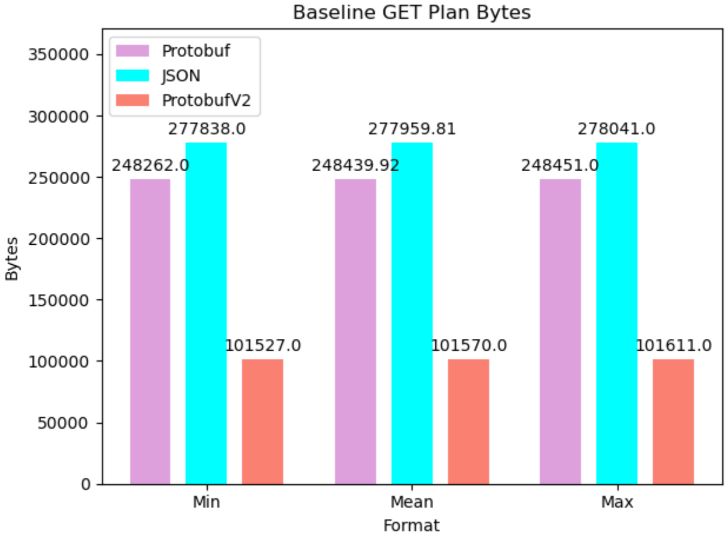


Figure 5.10: Baseline GET Plan Bytes

GET Specific Plan

The graphs in figure 5.11 and 5.12 describe the elapsed time and the throughput performance during query for the specific plan of a subscriber. Following the same pattern, ProtoV2Pj holds the best results, ProtoPj, despite being the one with the worst results, instead of the max value, has a smaller difference than the second-best, JsonPj. The throughput is almost the same as the other request.

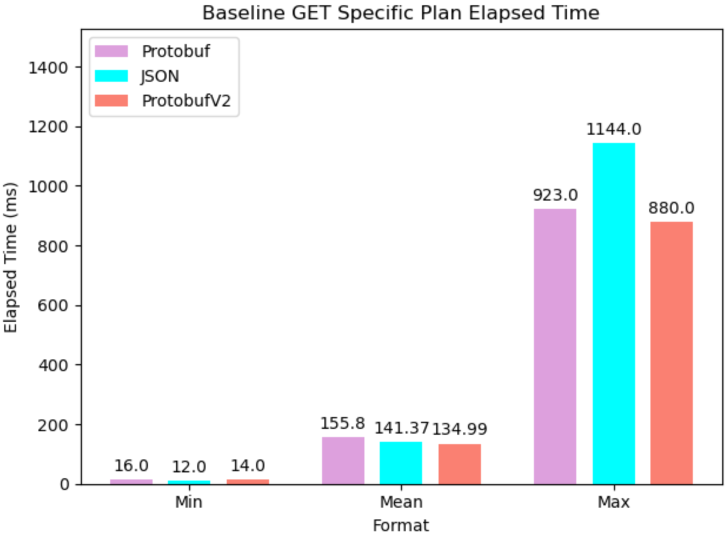


Figure 5.11: Baseline GET Specific Plan Elapsed Time

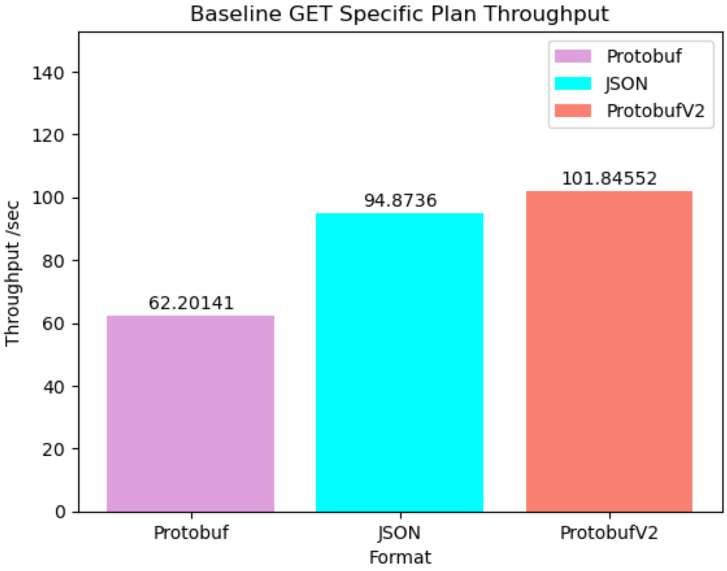


Figure 5.12: Baseline GET Specific Plan Throughput

5.2.3 Load Tests

Load tests are a type of performance testing that focuses on the behaviour and performance of an application under normal load conditions. It simulates the expected workload. It helps understand if the application can handle the projected load without compromising speed or user experience [28].

GET Plan

The graphs in figure 5.13 and 5.14 describe the elapsed time and the throughput performance during queries for all the plans. JsonPj got the highest value for the mean and the max. The min value for the elapsed time is equal to ProtoPj. Although ProtoV2Pj has the smallest min and max it loses, by a tiny difference in the mean for ProtoPj. ProtoPj has a slightly higher throughput, being on first by a small difference for the ProtoV2Pj.

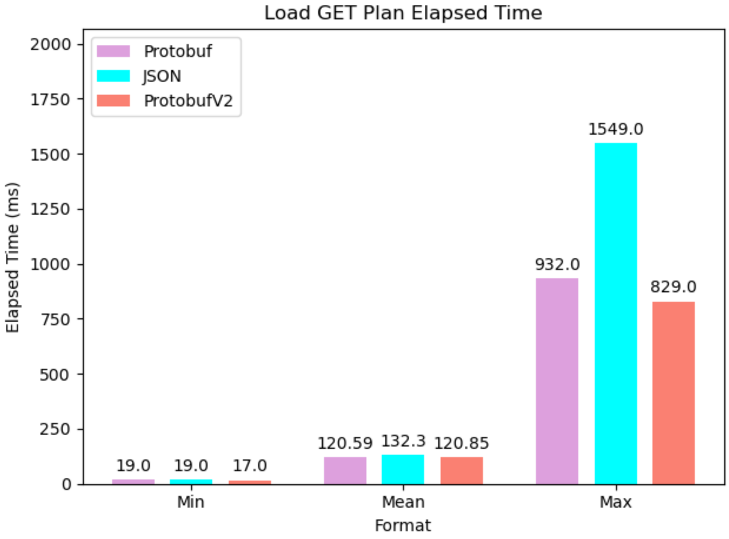


Figure 5.13: Load GET Plan Elapsed Time

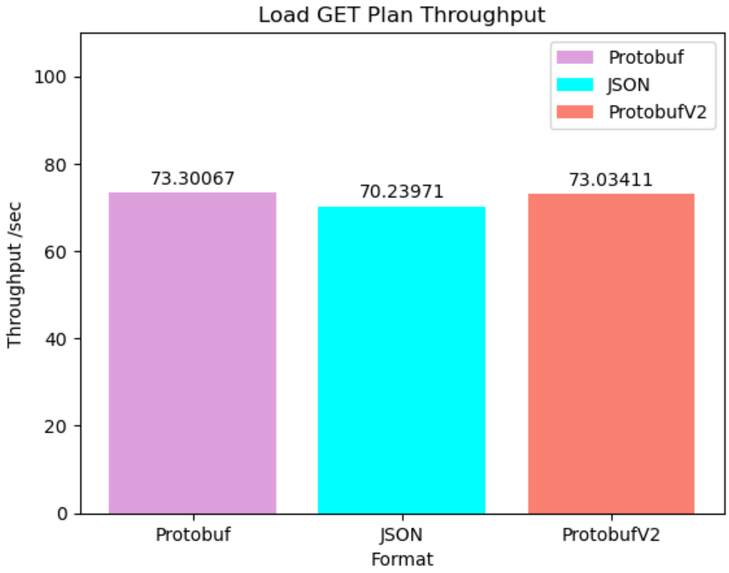


Figure 5.14: Load GET Plan Throughput

GET Specific Plan

The graphs in figure 5.15 and 5.16 describe the elapsed time and the throughput performance during query for the specific plan of a subscriber. Like the other request, ProtoPj has the best values for the mean, max and min. ProtoV2Pj comes after with the second-best times and JsonPj stays in last. In this one, ProtoV2Pj has a bigger throughput, by a small difference, than ProtoPj.

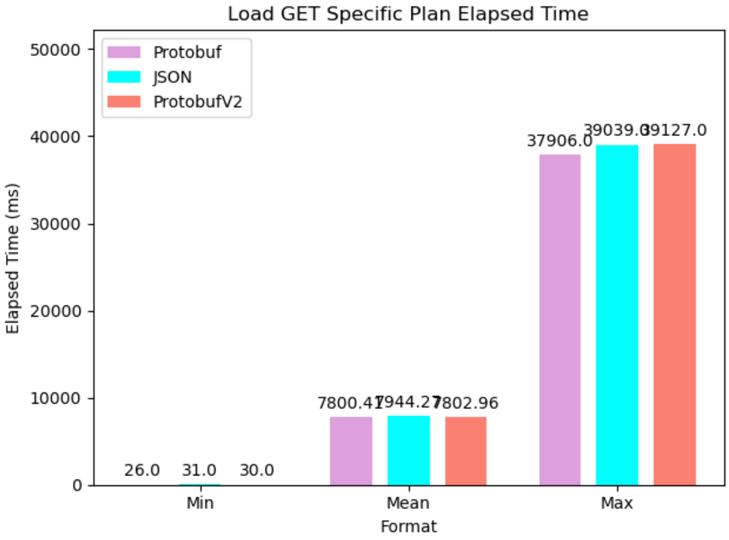


Figure 5.15: Load GET Specific Plan Elapsed Time

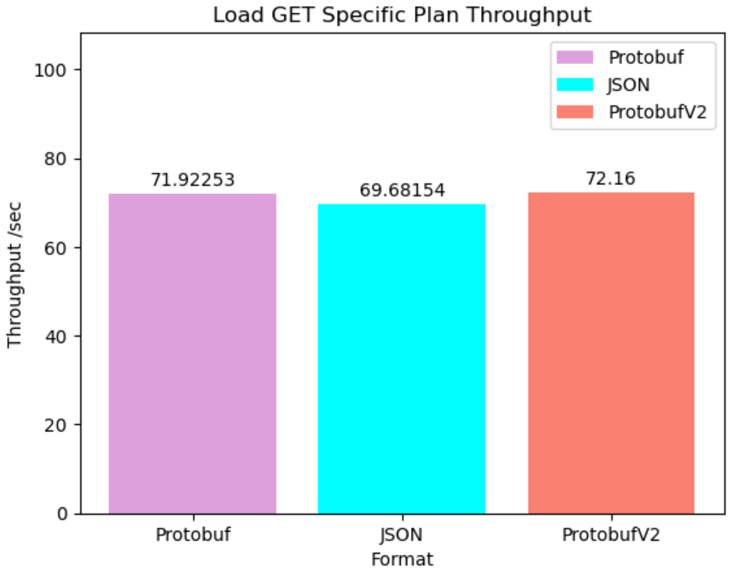


Figure 5.16: Load GET Specific Plan Throughput

5.2.4 Stress Tests

Stress tests are designed to push the application beyond its normal load conditions, this way, breaking points of the system can be identified. It helps to understand how the designed application behaves under extreme stress [28].

GET Plan

The graphs in figure 5.17 and 5.18 describe the elapsed time and the throughput performance during queries for all the plans. ProtoV2Pj has the smaller mean and max value, having his min equal to the JsonPj. ProtoPj has the smallest min. They have small differences in the throughput, although JsonPj stays on top after ProtoV2Pj.

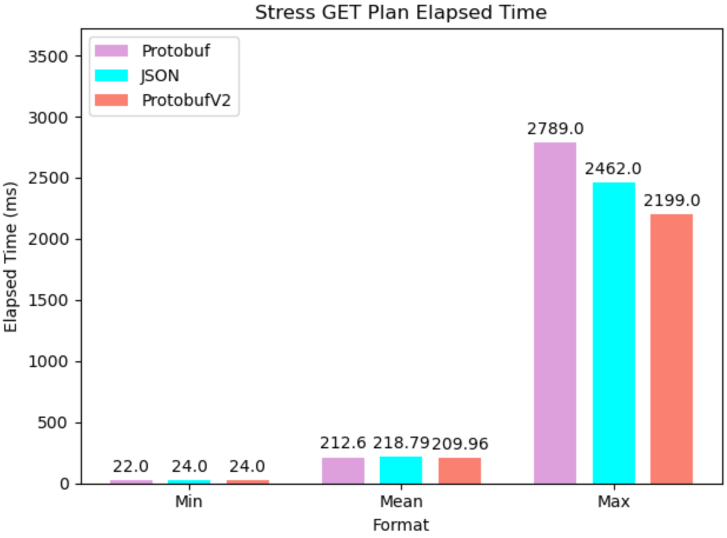


Figure 5.17: Stress GET Plan Elapsed Time

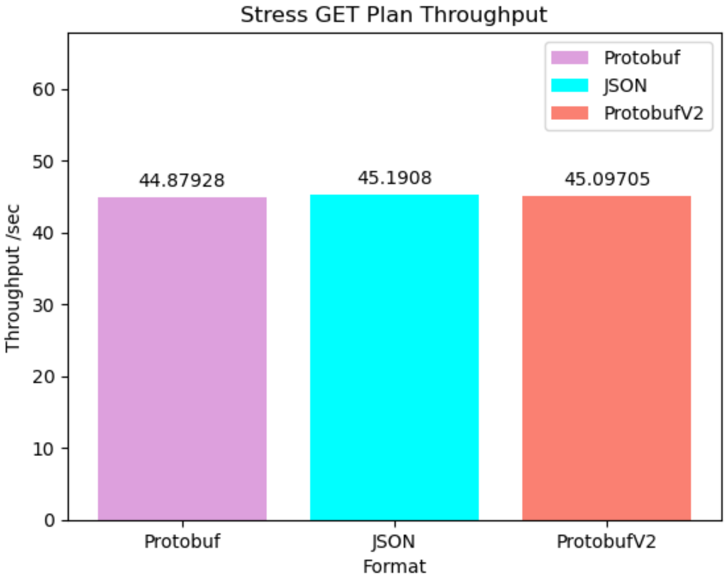


Figure 5.18: Stress GET Plan Throughput

GET Specific Plan

The graphs in figure 5.19 and 5.20 describe the elapsed time and the throughput performance during query for the specific plan of a subscriber. ProtoV2Pj has the biggest min even though the mean and the max are the lowest. They have almost the same value. The same thing happens in the throughput. They have almost the same value, still, ProtoV2Pj has the highest followed by ProtoPj.

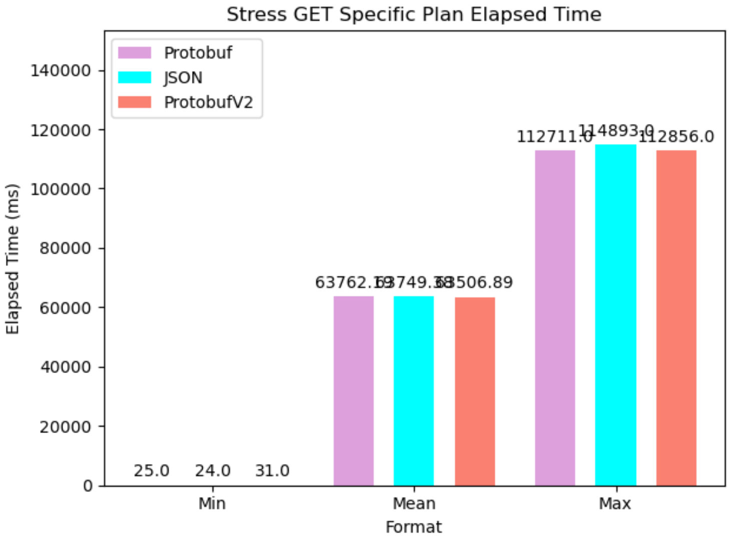


Figure 5.19: Stress GET Specific Plan Elapsed Time

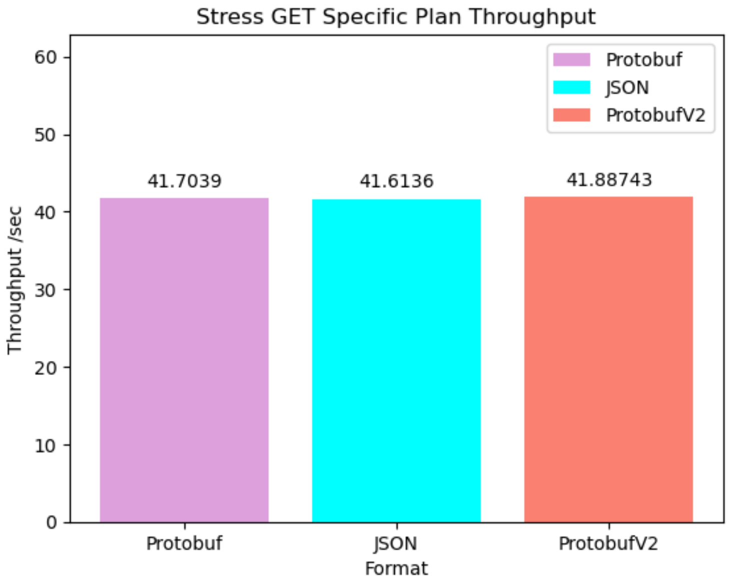


Figure 5.20: Stress GET Specific Plan Throughput

Figure 5.21 shows the percentage of error for this particular test. JsonPj had the highest percentage of errors, followed by ProtoPj. Although the percentage is very low it still makes a difference.

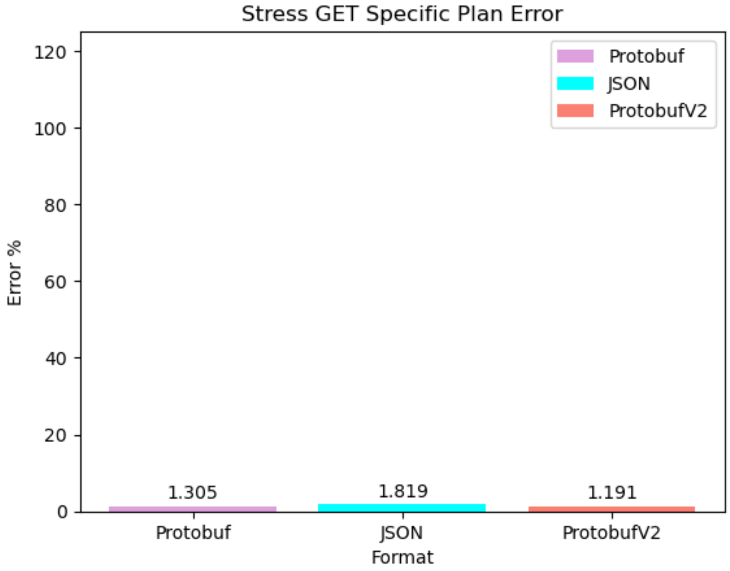


Figure 5.21: Stress GET Plan Error %

5.2.5 Soak Tests

Soak Tests are tests performed on the application to evaluate its endurance and responsiveness. It is meant to find any performance issues or bottlenecks [29].

Normally the soak tests should run for a very long period. Because of limited resources, it was not possible. Instead, it is a ramp-up period and threads until the max thread numbers, then it holds that for a defined duration. At the end, it ramps down the number of threads until it reaches 0.

GET Plan

The graphs in figure 5.22 and 5.23 describe the elapsed time and the throughput performance during queries for all the plans. JsonPj has the lowest mean and max. ProtoV2Pj comes in after and has the lowest min. ProtoPj has more the triple the mean time in comparison to the other two. The max value is 2.5x times more than the ProtoV2Pj.

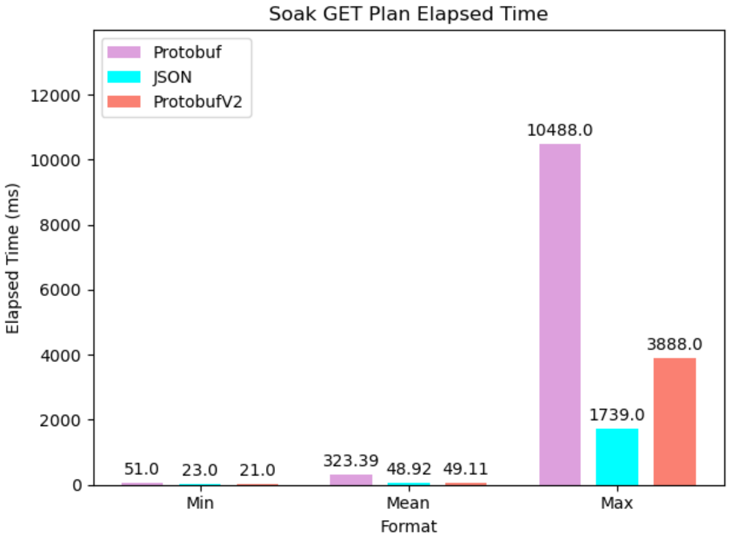


Figure 5.22: Soak GET Plan Elapsed Time

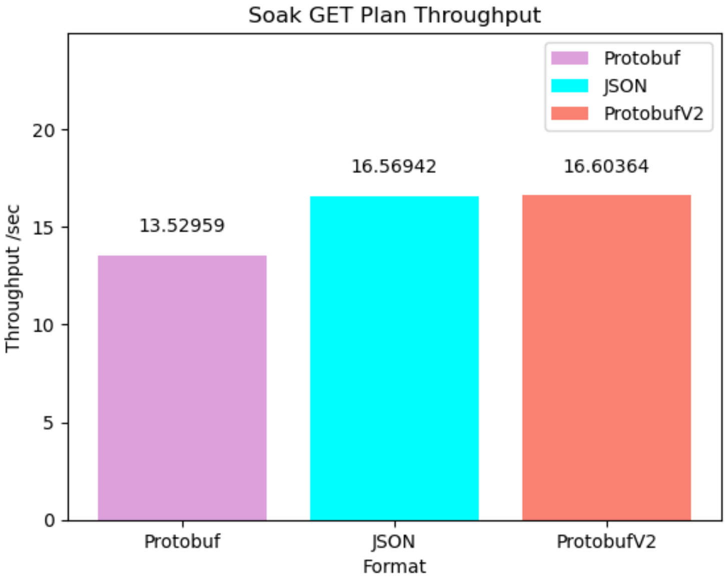


Figure 5.23: Soak GET Plan Throughput

GET Specific Plan

The graphs in figure 5.24 and 5.25 describe the elapsed time and the throughput performance during query for the specific plan of a subscriber. ProtoV2Pj has the best times in comparison to the other two, except in the max, but only for a little difference. ProtoPj stays with the exuberant times in relation to the other ones. ProtoV2Pj throughput is faintly better than JsonPj.

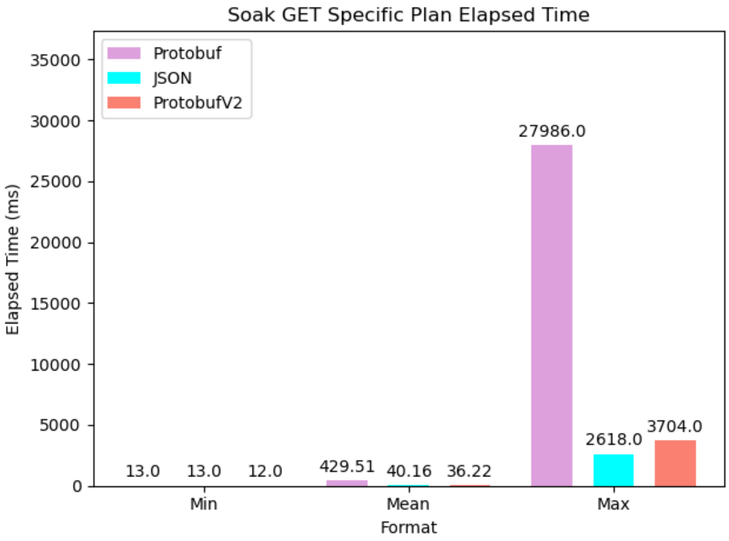


Figure 5.24: Soak GET Specific Plan Elapsed Time

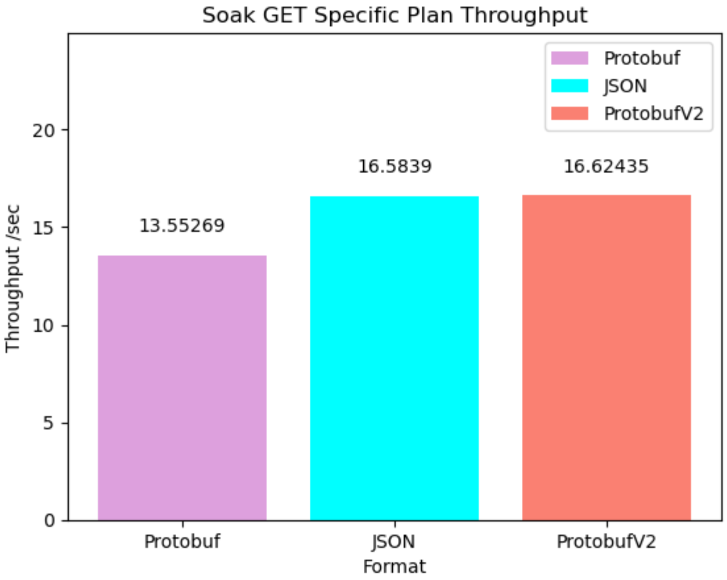


Figure 5.25: Soak GET Specific Plan Throughput

GET Details

The graphs in figure 5.26 and 5.27 describe the elapsed time and the throughput performance during query for all the subscriptions associated with a plan. Following the same pattern as the last two, ProtoV2Pj still has the best results, and the discrepancy between JsonPj and ProtoPj is a big one.

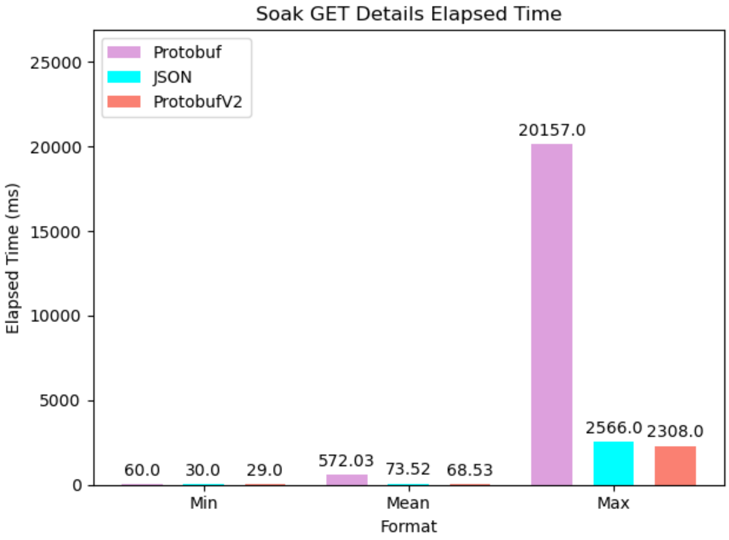


Figure 5.26: Soak GET Details Elapsed Time

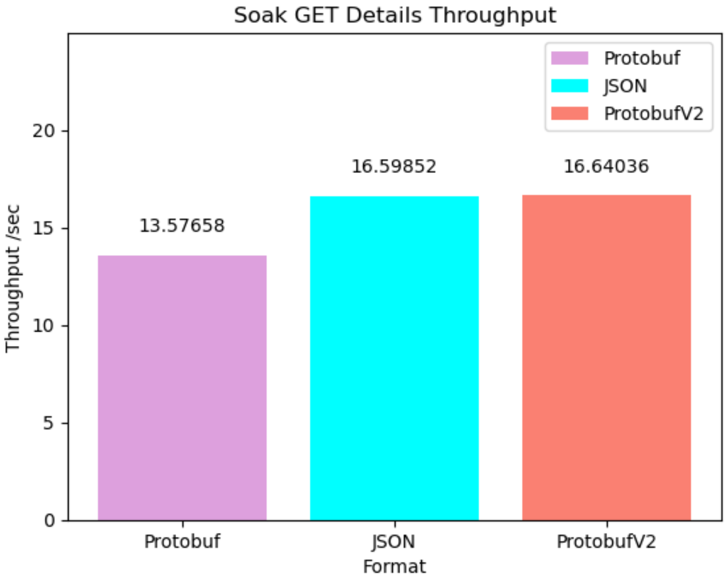


Figure 5.27: Soak GET Details Throughput

POST Login

The graphs in figure 5.28 and 5.29 describe the elapsed time and the throughput performance during users logging in. Once again the pattern repeats itself, ProtoV2Pj is, to some degree, better than JsonPj and both have a big advantage against ProtoPj.

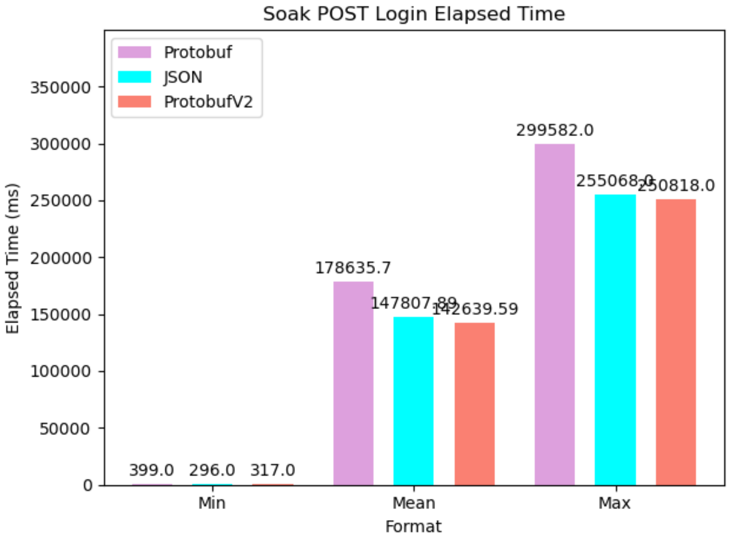


Figure 5.28: Soak POST Login Elapsed Time

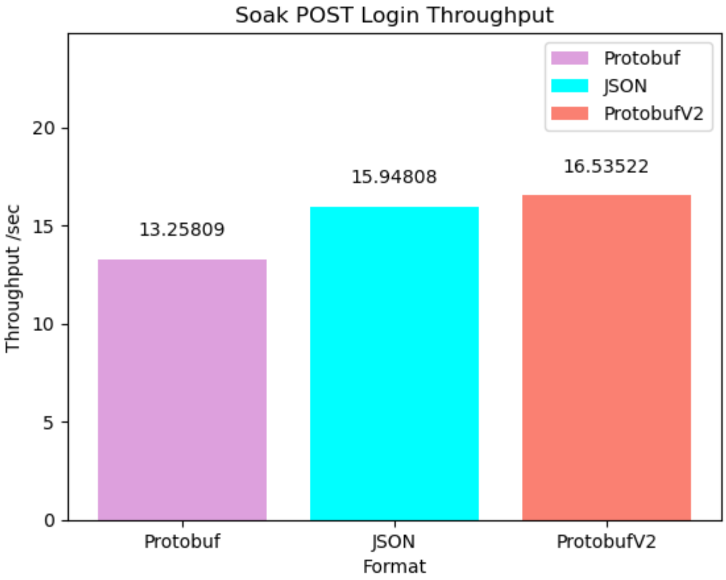


Figure 5.29: Soak POST Login Throughput

Looking at the errors obtained in this endpoint and described by the figure 5.30, ProtoPj was the project with the most percentage error, JsonPj comes second but with a difference of 8 % and then ProtoV2Pj with a difference of 0.7 %.

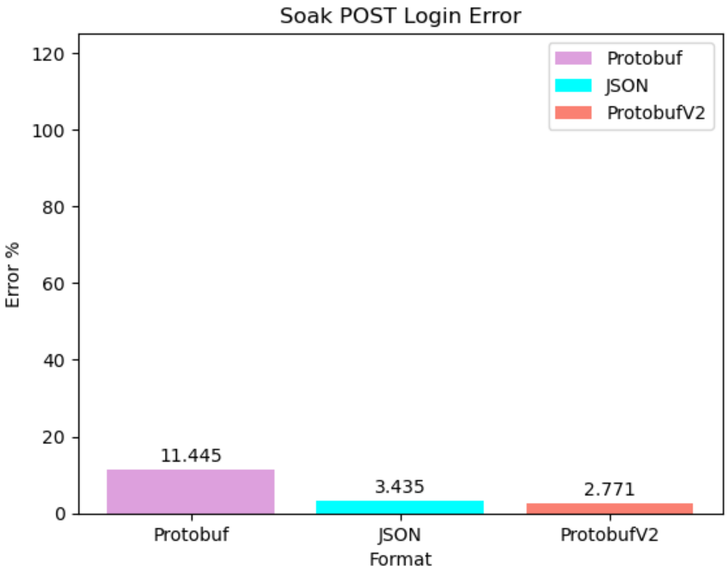


Figure 5.30: Soak POST Login Error %

5.3 JMeter Tests Results - Without Gateway

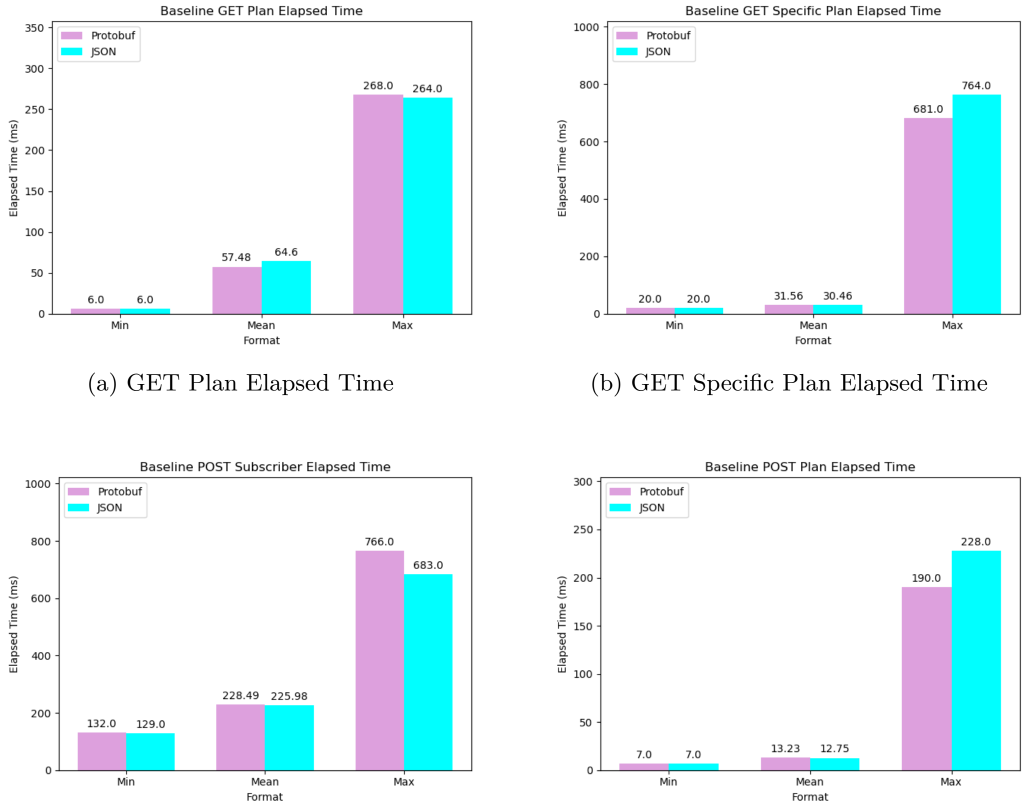
Before both projects had a gateway, almost all the tests were done and the results saved. In this state the Protobuf project, let's call it ProtoPj, had all internal and external communication using Protocol Buffers (Protobuf). The body of a request could be sent, and was, in JSON and when received would be automatically converted to a Protobuf message. The requests did not pass through a gateway, instead, each endpoint was specified.

The conditions for the tests were the same, only the endpoint changed. In addition to the two GETs, each thread group had two POST requests, the same POST request used to create some test data.

For these results, the pink bar represents the ProtoPj project and the blue JsonPj.

5.3.1 Baseline Tests - Without Gateway

Figure 5.31 agglomerates the four graphs related to the elapsed time of each endpoint. Analysing them, it is possible to gather that they are almost equal. For two of the graphs, 5.31b and 5.31d, ProtoPj has less max values and has the same value for the min. The mean is almost the same, but JsonPj is the lowest time with a small difference. In fact, only in graph 5.31a the mean is lowest with ProtoPj.



(a) GET Plan Elapsed Time

(b) GET Specific Plan Elapsed Time

(c) POST Subscriber Elapsed Time

(d) POST Plan Elapsed Time

Figure 5.31: Baseline Test without Gateway Elapsed Time Results

The throughput value of both projects was almost the same for each endpoint. ProtoPj had a higher rate than JsonPj, as it is visible in the table 5.3.

	Protobuf Project	JSON Project
Throughput \sec	30	29.7

Table 5.3: Baseline Test without Gateway Throughput Table

Examining the figure 5.32 is possible to conclude that ProtoPj sent fewer bytes than JsonPj. Even though new plan was being created in every request ProtoPj stay with a very small size compared with JsonPj.

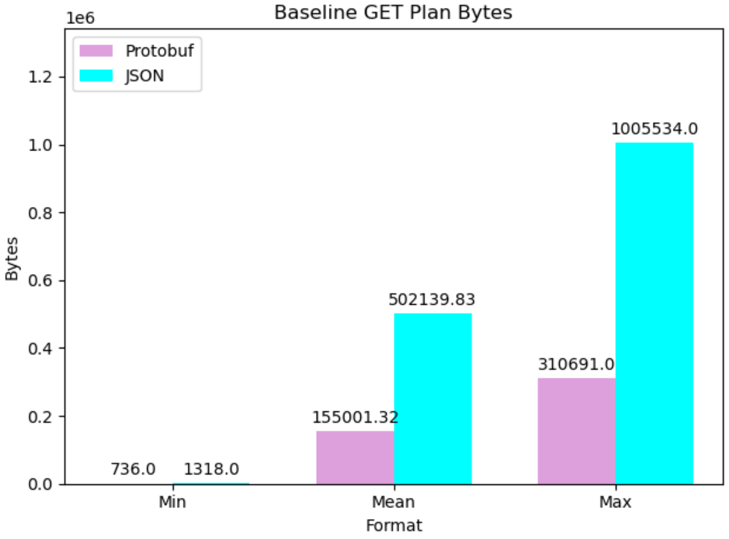
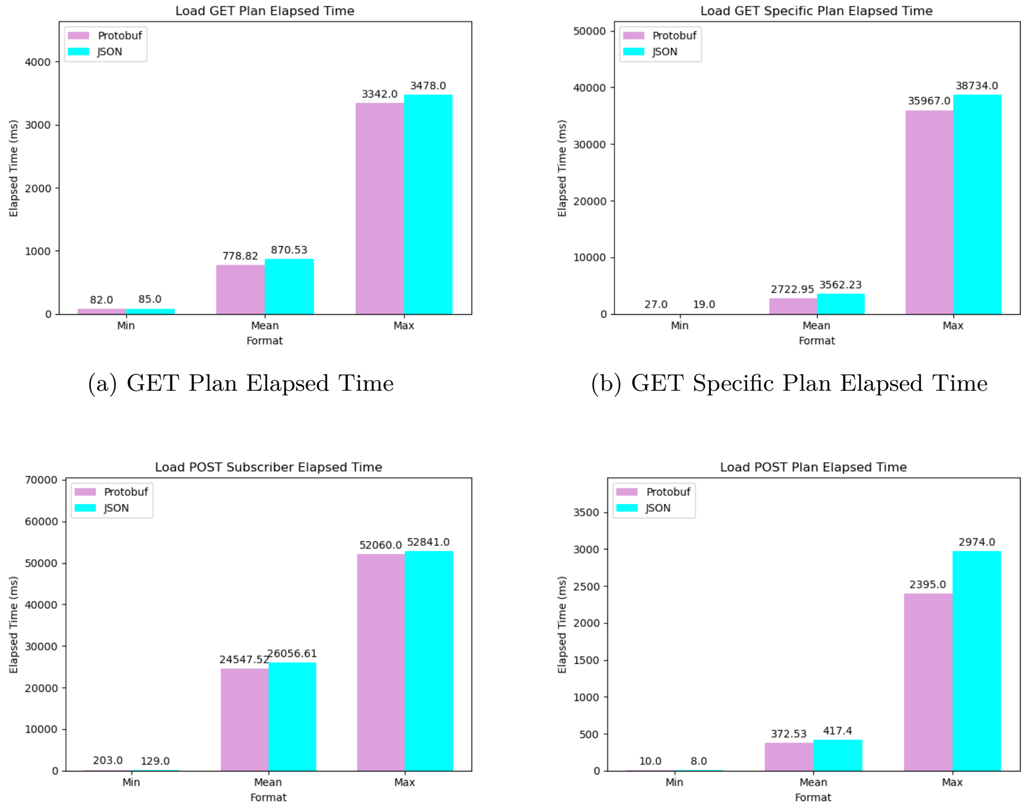


Figure 5.32: Baseline Test without Gateway - GET Plan Bytes

5.3.2 Load Tests - Without Gateway

Figure 5.33 agglomerates the four graphs related to the elapsed time of each endpoint. In all of them is understood that ProtoPj has a smaller mean and a smaller max. The only thing lacking is lower min values, but they are not so far apart from the JsonPj counterparts.



(a) GET Plan Elapsed Time

(b) GET Specific Plan Elapsed Time

(c) POST Subscriber Elapsed Time

(d) POST Plan Elapsed Time

Figure 5.33: Load Test without Gateway Elapsed Time Results

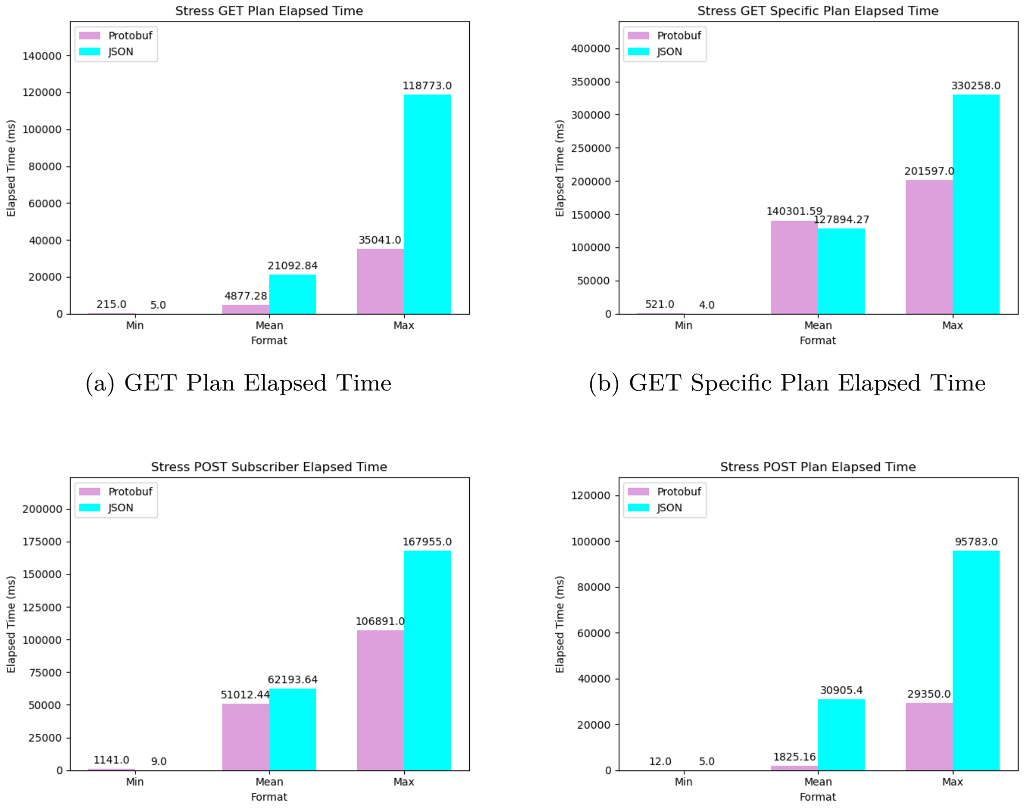
ProtoPj maintains superiority in the throughput, as visible in the 5.4. The throughput values were almost the same for every request.

	Protobuf Project	JSON Project
Throughput /sec	20.6	18.9

Table 5.4: Load Test without Gateway Throughput Table

5.3.3 Stress Tests - Without Gateway

Figure 5.34 agglomerates the four graphs related to the elapsed time of each endpoint. In almost all of them is understood that ProtoPj has a smaller mean and a smaller max. JsonPj also has all the min values to the lowest but has the max value the highest with a big difference. In the graph 5.34d the difference between mean values is devastating.



(a) GET Plan Elapsed Time

(b) GET Specific Plan Elapsed Time

(c) POST Subscriber Elapsed Time

(d) POST Plan Elapsed Time

Figure 5.34: Stress Test without Gateway Elapsed Time Results

ProtoPj maintains superiority in the throughput, as visible in the 5.5. The throughput values were almost the same for every request, and the difference between them increased.

	Protobuf Project	JSON Project
Throughput /sec	14.2	9.7

Table 5.5: Stress Test without Gateway Throughput Table

One more piece of data, crucial to this group of tests, is the error percentage. ProtoPj had the smallest error rate and JsonPj had some absurd ones. For two of the endpoints, ProtoPj did not present any errors. The table 5.6 exhibits all the error rates. JsonPj did not handle the stress test very well with four individual requests being made at the same time.

Endpoint	Protobuf Error %	JSON Error %
GET Plan	0	42.65
GET Specific Plan	13.56	79.91
POST Subscriber	7.7	86.1
POST Plan	0	31.06

Table 5.6: Stress Test without Gateway Error Rates

5.4 Hypothesis Tests

It is possible to do a statistical analysis of the data obtained through the JMeter tests. One of the possible tests that can be made, and the only one done, is the Wilcoxon Test.

Wilcoxon Test is a nonparametric statistical test that compares two paired groups. The tests essentially calculate the difference between sets of pairs and analyze these differences to establish if they are statistically significantly different from one another [30].

The library scipy.stats [31] for python provides a function to realise this. With this function, it is possible to verify if there is a significant difference between the values and if the values are significantly smaller or higher.

Because the data being analysed was composed of a large sample size, the significance level (alpha) chosen was 0.05. If the p-value calculated by the tests is smaller than the significance level, H0 (null hypothesis) can be rejected.

5.4.1 Results - With Gateway

With the gateway there were 3 projects, meaning 3 different datasets, so there are more than two hypotheses to consider. One pair to compare the performance ProtoV2Pj project versus JsonPj, another for ProtoPj versus JsonPj, and, in some cases, ProtoV2Pj versus ProtoPj. The last one is only considered if H1 (alternative hypothesis) is accepted for the other two. The H0 and H1 considered are identified in the table. **??**.

Elapsed Time	H0	H1
ProtoPj vs JsonPj	ProtoPj results are not significantly smaller than JsonPj	ProtoPj results are significantly smaller than JsonPj
ProtoV2Pj vs JsonPj	ProtoV2Pj results are not significantly smaller than JsonPj	ProtoV2Pj results are significantly smaller than JsonPj
ProtoV2Pj vs ProtoPj	ProtoV2Pj results are not significantly smaller than ProtoPj	ProtoV2Pj results are significantly smaller than ProtoPj

Table 5.7: Hypotheses for elapsed time

For the comprising of the payload size the hypothesis consider are the ones represented in the table 5.8.

Bytes	H0	H1
ProtoPj vs JsonPj	ProtoPj payloads are not significantly smaller than JsonPj	ProtoPj payloads are significantly smaller than JsonPj
ProtoV2Pj vs Json	ProtoV2Pj payloads are not significantly smaller than JsonPj	ProtoV2Pj payloads are significantly smaller than JsonPj
ProtoV2Pj vs ProtoPj	ProtoV2Pj payloads are not significantly smaller than ProtoPj	ProtoV2Pj payloads are significantly smaller than ProtoPj

Table 5.8: Hypotheses for payload size

Create Data Tests

As is possible to see in the list 5.4.1, JsonPj has statistically the smaller elapsed time values, except for the POST Plan endpoint that ProtoV2Pj had the smallest values.

- POST Subscriber
  - **p_value - 0.99 $>$ 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p_value - 0.99 $>$ 0.5** - H0 is not rejected, in other words 'ProtoV2Pj results are not significantly smaller than JsonPj'.
- POST Plan

  - **p__value - 0.99 > 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 0.00135 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

Baseline Tests

As is possible to see in the list 5.4.1, ProtoV2Pj has statistically the smaller elapsed time values, JsonPj comes right next having the second lowest times. JsonPj has the biggest payload size and ProtoV2Pj has the lowest.

- GET Plan
  - **p__value - 1.0 > 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 5.9e-77 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.
  - **p__value - 0.0 < 0.5** - H0 is rejected, therefore 'ProtoPj payloads are significantly smaller than JsonPj'.
  - **p__value - 0.0 < 0.5** - H0 is rejected, therefore 'ProtoV2Pj payloads are significantly smaller than JsonPj'.
  - **p__value - 0.0 < 0.5** - H0 is rejected, therefore 'ProtoV2Pj payloads are significantly smaller than ProtoPj'.
- GET Specific Plan
  - **p__value - 1 > 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 2.4655e-17 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

Load Tests

As is possible to see in the list 5.4.1, ProtoPj and ProtoV2Pj have the smallest time, but the difference between them are not significant.

- GET Plan
  - **p__value - 1.305e-76 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p__value - 1.179e-72 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

  - **p__value - 0.54 > 0.5** - H0 is not rejected, in other words 'ProtoV2Pj results are not significantly smaller than ProtoPj'.

- GET Specific Plan
  - **p__value - 0.00034 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p__value - 0.0077 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.
  - **p__value - 0.838 > 0.5** - H0 is not rejected, in other words 'ProtoV2Pj results are not significantly smaller than ProtoPj'.

Stress Tests

As is possible to see in the list 5.4.1, ProtoPj has significantly the smallest elapsed time, and for the GET Plan endpoint ProtoPj has significant lower elapsed times.

- GET Plan
  - **p__value - 4.071e-07 < 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 9.242e-16 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.
  - **p__value - 0.119 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than ProtoPj'.
- GET Specific Plan
  - **p__value - 0.99 > 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 0.114 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

Soak Tests

As is possible to see in the list 5.4.1, ProtoPj has significantly the smallest elapsed time in every request and ProtoPj has the highest values.

- GET Plan
  - **p__value - 1 > 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.

  - **p__value - 4.145e-303 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

- GET Specific Plan
  - **p__value - 1 $>$ 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
  - **p__value - 0.0 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.
- GET Details
  - **p__value - 1 $>$ 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p__value - 6.843e-303 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.
- POST Login
  - **p__value - 1 $>$ 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p__value - 0.0 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj results are significantly smaller than JsonPj'.

5.4.2 Results - Without Gateway

When analyzing the projects without the gateway only one pair of hypotheses is used, the one defined in the table 5.9.

Elapsed Time	H0	H1
ProtoPj vs JsonPj	ProtoPj results are not significantly smaller than JsonPj	ProtoPj results are significantly smaller than JsonPj

Table 5.9: Hypotheses for elapsed time

For the tests between bytes, the hypotheses pair used was the one defined in the table 5.10.

Bytes	H0	H1
ProtoPj vs JsonPj	ProtoPj payload sizes are not significantly smaller than JsonPj	ProtoPj payload sizes are significantly smaller than JsonPj

Table 5.10: Hypotheses for payload size

Baseline Tests

As is possible to see in the list 5.4.2, ProtoPj has statistically the smaller payload sizes. ProtoPj only has significantly smaller elapsed times in the GET Plan endpoint.

- GET Plan
  - **p_value - 2.525e-116 $<$ 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p_value - 0 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj payload sizes are significantly smaller than JsonPj'.
- GET Specific Plan
  - **p_value - 0.99 $>$ 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
- POST Subscriber
  - **p_value - 0.98 $>$ 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
- POST Plan
  - **p_value - 1 $>$ 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.

Load Tests

As is possible to see in the list 5.4.2, ProtoPj has statistically the smaller payload sizes and elapsed time.

- GET Plan
  - **p_value - 6.17e-74 $<$ 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p_value - 0 $<$ 0.5** - H0 is rejected, in other words 'ProtoV2Pj payload sizes are significantly smaller than JsonPj'.
- GET Specific Plan
  - **p_value - 1.312e-167 $<$ 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
- POST Subscriber

  - **p_value - 1.937e-222 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.

- POST Plan
  - **p_value - 4.774e-25 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.

Stress Tests

As is possible to see in the list 5.4.2, ProtoPj has statistically the smaller payload sizes and elapsed time, except for the endpoint GET Specific Plan. Although they are not significantly smaller in that endpoint, it is probably because 79.91% of the JsonPj requests returned error.

- GET Plan
  - **p_value - 8.67e-74 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
  - **p_value - 0 < 0.5** - H0 is rejected, in other words 'ProtoV2Pj payload sizes are significantly smaller than JsonPj'.
- GET Specific Plan
  - **p_value - 1 > 0.5** - H0 is not rejected, in other words 'ProtoPj results are not significantly smaller than JsonPj'.
- POST Subscriber
  - **p_value - 0 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.
- POST Plan
  - **p_value - 4.774e-25 < 0.5** - H0 is rejected, in other words 'ProtoPj results are significantly smaller than JsonPj'.

5.5 Results Discussion

Having in mind the results registered in section 5.2 and section 5.3, and the analysis in section 5.4 it is possible to conclude that Protobuf is overall a good option for REST applications.

Using only Protobuf for internal communication is not the best solution. Although in some cases the performance is equal to the JSON, it lacks in some parts. Because the project converted his data do JSON before answering a request, adds an extra layer that reduces the overall performance and is more prone to error. Also, there is no big advantage in the small size of the Protobuf message.

On the other hand, when using Protobuf for both internal and external, and, having in mind that the request body was in JSON, has some big benefits. Not only the overall application is less prone to errors, but it also has more throughput and in general less elapsed time.

In the tests with a gateway, the difference between JSON and Protobuf is not graphically the biggest, but the tests without a gateway show a better performance with big differences for Protobuf. Something that is proved by the statistical analysis that tells that Protobuf has significantly smaller values. That permits us to conclude that using Protobuf for the external and internal communication of the microservices allows for a better performance and smaller data size.

With Protobuf, considering the case where the project uses JSON for external communication, the project is less prone to errors. Those errors, probably caused by the conversion of data, might prove that Protobuf has a more efficient serializadion/deserilization process than JSON.

As demonstrated, the payload size of the projects that utilized Protobuf for external communication is significantly less than JSON. This is a positive, especially for applications that need to return extremely large datasets.

Chapter 6

Conclusion

This chapter has as its objective to expose the objectives achieved, and a suggestion for future work related to this project.

6.1 Objectives Achieved

During the internship, it was possible to learn some new concepts that were not taught during the degree, for example, Protocol Buffers (Protobuf), Gateway pattern and performance tests. Other concepts like microservice and Spring framework were deepened.

The main objective of this internship was to find the possible benefits in performance, efficient serialization and reduced payload size when using Protobuf in REST applications instead of the normally used JSON. This was accomplished by migrating an existent project, that used JSON with REST and was divided by microservices, to Protobuf. That process consisted of changing the internal connection between microservices and establishing a gateway to route the request. It was possible to establish two versions of the Protobuf project with the gateway. One used JSON as external communication and the other used Protobuf.

A series of tests were realized to obtain the necessary data to compare all projects. These test results included data register not only from the two versions of the Protobuf project and from the JSON project already mentioned but also from data retrieve when the project did not have a gateway established. When those tests were made, the Protobuf project had the external communication being done with Protobuf.

Analysing the results from all tests and the results from the hypothesis tests, it was possible to conclude that Protobuf is best worth using when all communication in the project is done with Protobuf, because is significantly better than JSON.

All the work and documentation is available at a public available repository at GitHub [25] of my ownership.

6.2 Future Work

In a future work, this study is anticipated to continue on a larger scale. A more extensive project is expected to make it easier to verify differences between projects. The focus will be on studying the benefits of Protobuf in a large REST project.

Another future work would be developing a method to simplify the migration process from Protobuf to JSON.

References

[1] baeldung, “Spring rest api with protocol buffers.” https://www.baeldung.com/spring-rest-api-with-protocol-buffers, January 2024. [Cited on page 1]

[2] G. Kaur and M. M. Fuad, “An evaluation of protocol buffer,” in *Proceedings of the IEEE SoutheastCon 2010 (SoutheastCon)*, pp. 459–462, 2010. [Cited on page 1]

[3] Kong, “What is grpc? use cases and benefits.” https://konghq.com/learning-center/api-management/what-is-grpc, April 2022. [Cited on page 1]

[4] Seara.com, “Protobuf lab session.” https://farfetchtechblog.com/en/blog/post/protobuf-lab-session/, 2020. Section: Technology. [Cited on page 2]

[5] J. Vieira, “Github - joaovieira17/joaov_grpc_communication: Application development using grpc for inter-service communication and performance comparison with rest.” https://github.com/joaovieira17/joaov_grpc_communication, 2023. [Cited on page 2]

[6] R. Gancarz, “Linkedin adopts protocol buffers for microservices integration and reduces latency by up to 60%.” https://www.infoq.com/news/2023/07/linkedin-protocol-buffers-restli/, July 2023. [Cited on page 2]

[7] HazelCast, “What is serialization and how does it work?.” https://hazelcast.com/glossary/serialization/, January 2024. [Cited on pages 5 and 6]

[8] HazelCast, “Data serialization process.” https://hazelcast.com/wp-content/uploads/2024/01/glossary-serialization-2.svg, 2024. [Cited on pages ix and 6]

[9] E. R. Harold and W. S. Means, *XML in a Nutshell.* O’Reilly Media, Inc., 2004. [Cited on page 6]

[10] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, “Comparison of json and xml data interchange formats: A case study,” in *22nd International Conference on Computer Applications in Industry and Engineering 2009, CAINE 2009*, pp. 157–162, 2009. [Cited on pages 6 and 7]

[11] A. Sumaray and S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, pp. 1–6, ACM, February 2012. [Cited on page 7]

[12] Google, "Protocol buffers." `https://protobuf.dev/`. [Cited on pages xi, 7, 9, and 10]

[13] HackingNote, "Proto2 vs proto3." `https://www.hackingnote.com/en/versus/proto2-vs-proto3/`, February 2022. [Cited on page 8]

[14] D. Shvaika, A. Shvaika, and V. Artemchuk, "Data serialization protocols in iot: problems and solutions using the thingsboard platform as an example," in *4th Edge Computing Workshop*, (Zhytomyr, Ukraine), 2024. [Cited on page 12]

[15] C. Currier, *Protocol Buffers*, pp. 223–260. Springer Nature, 2022. [Cited on pages ix, 12, and 13]

[16] C. Richardson, "What are microservices?." `https://microservices.io/`. [Cited on pages 13 and 14]

[17] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015. [Cited on page 13]

[18] J. Networks, "What is a cloud microservice?." `https://www.juniper.net/us/en/research-topics/what-is-a-cloud-microservice.html`. [Cited on pages ix and 13]

[19] gRPC, "Introduction to grpc." `https://grpc.io/img/landing-2.svg`, February 2023. [Cited on pages ix and 15]

[20] L. Gupta, "What is rest?: Rest api tutorial." `https://restfulapi.net/`, 2018. [Cited on page 15]

[21] The Apache Software Foundation, "Apache jmeter." `https://jmeter.apache.org/`. [Cited on page 16]

[22] Postman, "What is Postman? Postman API Platform." `https://www.postman.com/product/what-is-postman/`. [Cited on page 16]

[23] T. Ribeiro, "sidis-isep." `https://bitbucket.org/tamr65/sidis-isep/src/master/`, 2023. [Cited on page 17]

[24] C. Richardso, "Microservices Pattern: Pattern: Api Gateway / Backends for Frontends." `https://microservices.io/patterns/apigateway.html`. [Cited on page 25]

[25] T. Ribeiro, "Github - tam65r/protobuf-in-REST-applications." `https://github.com/tam65r/protobuf-in-REST-applications`. [Cited on pages 29, 35, and 78]

[26] Google, "GitHub - google/gson: A java serialization/deserialization library to convert java objects into json and back." `https://github.com/google/gson`, n.d. [Cited on page 29]

[27] A. Thomas, "Baseline testing: What is it, and Why is it important?." `https://testsigma.com/blog/baseline-testing/`, jun 30 2023. [Cited on page 51]

[28] F. Salami, "Load Testing vs Stress Testing." `https://testsigma.com/blog/load-testing-vs-stress-testing/`, sep 26 2023. [Cited on pages 54 and 56]

[29] S. Wickramasinghe, "Soak Testing – What it is How to Do It with Example?." `https://testsigma.com/blog/soak-testing/`, nov 7 2023. [Cited on page 59]

[30] A. Hayes, "Wilcoxon Test: Definition in Statistics, Types, and Calculation," *Investopedia*, jun 25 2010. [Cited on page 69]

[31] "scipy.stats.wilcoxon — SciPy v1.13.1 Manual." `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html`. [Cited on page 69]

Appendix A

Code Definitions

A.1 Java JPA Entities and Request Bodies

```java
@Entity
@Table(name = "subscription")
public class SubscriptionJPA {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String subscriberID;

    @Column(nullable = false)
    @NotBlank
    @Size(min = 1, max = 32)
    private String plan;

    @Nullable
    @Column(nullable = false)
    private LocalDateTime initialDate;


    private boolean isActive;

    @Nullable
    private LocalDateTime finalDate;

    private FeeType feeType;

    private PaymentMethod paymentMethod;

    private boolean isRenewed;
    @Nullable
    private LocalDateTime endSubscriptionDate;
```

Listing A.1: Subscription JPA

```
public enum FeeType {
    ANNUAL("Annual"),
    MONTHLY("Monthly"),
    NA("N/A");


public enum PaymentMethod {
    CARD("Card"),
    MBWAY("MBWay"),
    NA("N/A");
```

Listing A.2: Subscription Enum

```
@Entity
@Table(name = "users")
@EntityListeners(AuditingEntityListener.class)
public class UserJPA implements UserDetails {

     @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     private Long id;

     @Column(nullable = false, updatable = false)
     @NotNull
     @NotBlank
     @Size(min = 1, max = 32)
     private String name;

     @Setter
     @Getter
     private boolean enabled = true;

```

```
    @Column(nullable = false)
    private String citizenCardNumber;

    @Column(nullable = false)
    private LocalDateTime birthday;

    @Column(nullable = false)
    private String phoneNumber;
    @Column(nullable = false)
    private Gender sex;

    @Column(unique = true, updatable = false, nullable = false)
    @Email
    @Getter
    @NotNull
    @NotBlank
    private String username;

    @Column(nullable = false)
    @Getter
    @NotNull
    @NotBlank
    private String password;

    @ElementCollection
    @Getter
    private final Set<AuthorityRole> authorities = new HashSet<>();
```

Listing A.3: User JPA

```
public enum Gender {
   FEMALE("Female"),
   MALE("Male"),
   OTHER("Other");
```

Listing A.4: User Enum

A.2 Proto Schema Definitions

A.2.1 DTO

```
syntax = "proto3";
package example;
option java_package =
    "com.example.sisdi_users.usermanagement.model.proto";
```

```


message User {
  string name = 1;
  string citizenCardNumber = 3;
  string birthday = 4;
  string phoneNumber = 5;
  string username = 6;

  enum Gender {
    OTHER = 0;
    FEMALE = 1;
    MALE = 2;
  }


  Gender gender = 7;
  string role = 2;
}

message UserList {
  repeated User user = 1;
}
```

Listing A.5: User Entity DTO

```
syntax = "proto3";
package example;
option java_package = "com.example.sisdi_subscriptions.subscriptionmanagement.model.proto";


message Subscription {
  string subscriberID = 1;
  string plan = 2;
  string initialDate = 3;
  bool isActive = 4;
  string finalDate = 5;
  string endSubscriptionDate = 6;

  enum FeeType {
    FEE_TYPE_NA = 0;
    MONTHLY = 1;
    ANNUAL = 2;
  }

  enum PaymentMethod {
    PAYMENT_METHOD_NA = 0;
    MBWAY = 1;
    CARD = 2;
  }

  FeeType feeType = 7;
  PaymentMethod paymentMethod = 8;
}

message SubscriptionList {
  repeated Subscription subscriptions = 1;
}
```

Listing A.6: Subscription Entity DTO

A.2.2 Requests

```
syntax = "proto3";
package example;
option java_package =
    "com.example.sisdi_users.usermanagement.api.proto";


message CreateSubscriptionRequest {
  string username = 1;
  string plan = 2;
  string feeType = 3;
  string paymentMethod = 4;
  string initialDate = 5;
}


message CreateUserRequest {
  string username = 1;
  string password = 2;
  string name = 3;
  string citizenCardNumber = 4;
  string birthday = 5;
  string phoneNumber = 6;
  string sex = 7;
  string role = 8;
  string plan = 9;
  string feeType = 10;
  string paymentMethod = 11;
  string initialDate = 12;
}

message AuthRequest {
  string username = 1;
  string password = 2;
}
```

Listing A.7: User Requests

```
syntax = "proto3";
package example;
option java_package =
    "com.example.sisdi_subscriptions.subscriptionmanagement.api.proto";


message CreateSubscriptionRequest {
  string username = 1;
  string plan = 2;
  string feeType = 3;
  string paymentMethod = 4;
  string initialDate = 5;
}
```

Listing A.8: Subscription Requests