



[Protobuf and REST: assessing benefits and drawbacks]

Instituto Superior de Engenharia do Porto

2023/2024

José Miguel Bártolo Oliveira Sousa Guimarães

Supervisor: Isabel de Fátima Azevedo

ISEP INSTITUTO SUPERIOR
DE ENGENHARIA DO PORTO

Abstract

Today, due to the omnipresence of the internet in our digital lives, there is an ever-increasing demand for fast and reliable data transmission between machines running applications all over the world. A considerable amount of those applications use a software architectural style in the basis of their design known as REST. Most REST applications use JSON as their data serialization format of choice. A data serialization format is used as a way to convert data structures or object states into a very efficient format, that will then be transmitted to other applications and then reconstructed there, a process called data serialization/deserialization.

Protobuf is a data serialization format developed by Google, normally used in gRPC applications, but can also be used in REST. Protobuf is more lightweight than JSON, due to the extremely efficient way it encodes its data compared to JSON.

This project explores the possibility of using Protobuf in REST applications as its data serialization format in REST applications, as opposed to the more traditional use of JSON. This exploration was made by choosing a preexisting REST application using JSON, adapting it to use Protobuf, running a series of tests on both the original application using JSON and the new one using Protobuf, and then analyzing and taking conclusions from the tests results.

The test results showed that Protobuf usage over JSON can have a beneficial impact in the performance of the application, especially when the application is subjected to more intense concurrent usage.

Keywords: Microservices, REST, Data Serialization Format, JSON, Protobuf, JMeter

Contents

List of Figures	vii
List of Source Code	ix
1 Introduction	1
1.1 Contextualization	1
1.2 Problem Description	1
1.2.1 Objectives	1
1.2.2 Approach	2
1.2.3 Contributions	2
1.3 Report Structure	2
2 State of the Art	3
2.1 REST	3
2.2 Microservices	4
2.3 Structured Data Serialization	5
2.3.1 XML	5
2.3.2 JSON	6
2.3.3 Protobuf	7
2.4 Tools	10
2.4.1 Postman	10
2.4.2 Protoman	10
2.4.3 JMeter	11
3 From Analysis to Test	13
3.1 Project Analysis and Development	13
3.2 Testing	27
3.2.1 Tests Configuration	27
3.2.2 Data Creation Tests	30
3.2.3 Baseline Tests	34
3.2.4 Load Tests	37
3.2.5 Stress Tests	40
3.2.6 Soak Tests	43
4 Conclusions	47
4.1 Objectives Achieved	47
4.2 Contributions	47
Bibliography	49

List of Figures

2.1	Diagram of the basic interactions of a REST API.[6]	4
2.2	API call made with Protoman, showing the request and response body human-readable.	11
3.1	Folder structure of the project	14
3.2	PetClinic's Domain Model[21]	14
3.3	Use Case Diagram	15
3.4	Component Diagram	16
3.5	Error calling addPet, even when submitting supposedly valid information.	18
3.6	getOwnersPet returning 404 Not Found, even though the owner with ownerId '1' owns the pet with petId '1'.	18
3.7	Specialty with id '3' exists and should have been able to be deleted by this call.	19
3.8	Error calling addVisit, even when submitting supposedly valid information.	19
3.9	Sequence Diagram for listPetTypes	20
3.10	Sequence Diagram for listOwners	20
3.11	Sequence Diagram for addPetType	21
3.12	Sequence Diagram for addOwner	21
3.13	.proto file used in the Protobuf project being integrated into Protoman	25
3.14	Example of an API call in Protoman and the request collection in the sidebar	26
3.15	Example of a Thread Group Configuration	28
3.16	JMeter's Test Plan Structure	29
3.17	Script to create JSON body for addPetType call	30
3.18	Script to create Protobuf body for addPetType call	30
3.19	Script to create JSON body for addOwner call	31
3.20	Script to create Protobuf body for addOwner call	31
3.21	POST addPetType Elapsed Time	32
3.22	POST addPetType Throughput	32
3.23	POST addOwner Elapsed Time	33
3.24	POST addOwner Throughput	33
3.25	Baseline GET listPetTypes Elapsed Time	34
3.26	Baseline GET listPetTypes Throughput	35
3.27	Baseline GET listOwners Elapsed Time	35
3.28	Baseline GET listOwners Throughput	36
3.29	Load GET listPetTypes Elapsed Time	37
3.30	Load GET listPetTypes Throughput	38
3.31	Load GET listOwners Elapsed Time	38
3.32	Load GET listOwners Throughput	39
3.33	Stress GET listPetTypes Elapsed Time	40
3.34	Stress GET listPetTypes Throughput	41
3.35	Stress GET listOwners Elapsed Time	41
3.36	Stress GET listOwners Throughput	42

3.37	Parametrization of the Soak Tests	43
3.38	Graph representing the number of expected active users during the duration of the test	43
3.39	Soak GET listPetTypes Elapsed Time	44
3.40	Soak GET listPetTypes Throughput	44
3.41	Soak GET listOwners Elapsed Time	45
3.42	Soak GET listOwners Throughput	45

List of Source Code

2.1	XML code excerpt containing information about a pet	6
2.2	JSON code excerpt containing information about a pet	7
2.3	Protobuf code excerpt containing messages representing the field types mentioned above	8
2.4	Protobuf code excerpt containing messages representing the field labels mentioned above	9
2.5	Protobuf code excerpt containing messages to serialize information about a pet (representing the same data as in the XML and JSON examples) . . .	9
3.1	Example of a "list" method (listPetTypes)	17
3.2	Example of a "add" method (addPetType)	17
3.3	Example of a DTO (OwnerDto)	22
3.4	Examples of the messages above for the entity Pet	23
3.5	Examples of more of the messages above for the entity Pet	24
3.6	Method listPetTypes now using Protobuf	24
3.7	Method getPetType now using Protobuf	25

Chapter 1

Introduction

1.1 Contextualization

Applications and services developed with a microservices based architecture are extremely common in today's world, providing a myriad of advantages over the now supplanted monolithic architecture styled applications more commonly used in the past [1], but one key feature that is of extreme relevance to this project is the isolation of different responsibilities of a system in different parts, the aforementioned microservices.

For these isolated services to be able to communicate with each other, certain technologies are used, like REST (Representational State Transfer) and gRPC (Google Remote Procedure Call). Of these two, the architecture that is more prevalent is REST, using JSON as its data serialization format, but gRPC has been gaining traction in the past few years, mainly due to reports of it being more efficient than REST in high-demand environments [2]. Part of the reason of these advantages existing is the fact that gRPC typically uses Protobuf instead of JSON, a data serialization format that is more lightweight and permits faster serialization and deserialization than JSON [3].

However, Protobuf can also be used in REST APIs, and most of the studies related to the performance of these systems are made comparing applications using REST/JSON and gRPC/Protobuf.

1.2 Problem Description

The problem in question is being able to compare two REST projects that contain only a few microservices each at most, one using JSON and the other using Protobuf as its data serialization format, and derive conclusions about the potential benefits of Protobuf usage in REST over JSON.

1.2.1 Objectives

The main objective is to compare two REST projects with different data serialization formats, JSON and Protobuf, in order to assess potential advantages and disadvantages in using Protobuf in a REST project.

1.2.2 Approach

Firstly, a general investigation needs to be developed about the characteristics of JSON and Protobuf, the relevant differences between them in terms of their structure and the way they are implemented in REST.

After this investigation, a search for an adequate project to choose and adapt will start, having into account certain requirements, such as the fact that the project needs to use a REST architecture with JSON. The project in question also needs to provide a license to fork and edit it. Then the chosen project will be adapted to use Protobuf instead of JSON, using the investigations previously mentioned as a basis for this adaptation to be successful.

After the adaptation of the project, performance measuring tools like jMeter and Sonargraph will be used to obtain data about the differences in efficiency of the projects (the original one and the adapted one). Then the data collected and other considerations that are not measurable in a quantifiable way (like the difference in the ease of use of JSON and Protobuf) will be used to produce possible conclusions about the potential benefits of the usage of Protobuf instead of JSON.

The work developed will be synthesized in a report, and both this report and the adapted project will be available in a public repository.

1.2.3 Contributions

The main contribution that this project aims to provide is potentially valuable data about the usage of Protobuf in REST APIs. Success in demonstrating efficiency benefits of Protobuf usage in REST could imply several benefits:

1. Expanding the universe of approaches to API development by providing data that supports an underused approach.
2. Potential cost reductions to the resources needed to deploy applications to the public.
3. More efficient usage of resources can be a source of help to the environment.

1.3 Report Structure

This report is composed as follows by the following chapters:

1. **Introduction:** Introduction to the project, providing a contextualization for the project, its objectives and the approach to fulfill them.
2. **State of the Art:** Investigation of the state of the art, concerning other works and technologies related to the area of this project.
3. **From Analysis to Test:** Analysis of a chosen project/subsequent development and testing of both the projects.
4. **Conclusions:** Conclusions about the work developed.

Chapter 2

State of the Art

In order to be well-equipped to solve the difficulties and meet the objectives relative to the problem, an investigation of the state of the art of certain concepts and technologies related to the problem was made, as well as a synthesis of the investigation's findings.

This synthesis is organized and subdivided subsequently in the following topics:

- General resume of the main characteristics of REST.
- A brief resume of why are microservices important and, therefore, data serialization formats.
- Analysis of XML, a historically important data serialization format.
- Analysis of the two types of data serialization formats relevant to our problem (JSON and Protobuf), with a special focus on Protobuf.
- Main characteristics and purposes of some tools that will be used in the project.

2.1 REST

REST (Representational State Transfer) is a software architectural style used to develop web services by defining a set of protocols and a client-server style of communication, while also limiting the usage of bandwidth, making it very useful for the internet [4].

REST APIs usually operate based on the following principles [5]:

- **Stateless** - the server side is not responsible to hold the state of each one of the client applications; each client is responsible for the information relative to its interactions with the server.
- **Client/Server** - the client and server side are separated in different services, that can be implemented and deployed independently, as long as they conform to the requisites of another principle on this list: uniform interface.
- **Uniform Interface** - each application must conform to certain constraints, identified by REST's creator Roy Fielding as: identification of resources, manipulation of resources through representations, self-descriptive messages and hypermedia as the engine of application state (using links as the mean to connect to different resources).
- **Cacheable** - servers must declare the cacheability of each response's data, in order to minimize the server's load, improving the availability and reliability of the application overall.

- **Layered System** - enables certain network-based components to serve as intermediaries between the client and the server, enforcing security, response caching and load balancing.
- **Code on Demand** - servers can transfer executable programs to the clients if necessary or at least beneficial to the client-server interaction.

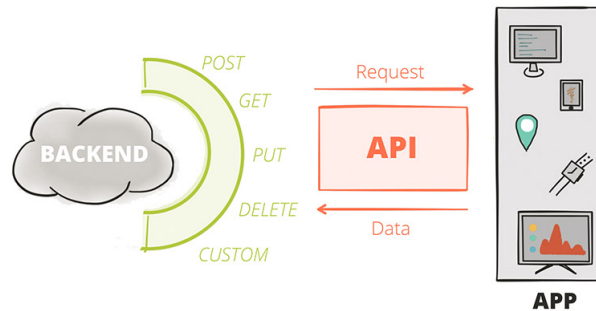


Figure 2.1: Diagram of the basic interactions of a REST API.[6]

2.2 Microservices

Microservices are a software architecture that appeared out of the necessity to correct some of the drawbacks of the monolithic architecture, mainly in the application scaling field, that can be very difficult in a monolithic architecture, because all processes are tightly coupled into a single service [7].

With a microservices architecture, the system is built using independent and loosely coupled components that run each application process as a different service. Updating and scaling an application to meet demand for specific functions it's much easier in this case, because the changes usually are only related to a specific service [7].

There are a lot of benefits to using a microservices architecture, such as [8]:

- Enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable.
- The microservices architecture enables teams to be autonomous.
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation.

However, there are some drawbacks of a microservices based architecture too: [8]:

- Finding the right set of services is challenging.
- Distributed systems are more complex, which makes development, testing and deployment difficult.
- Deploying features that span multiple services requires careful coordination.

- Deciding when to adopt the microservice architecture is difficult.

2.3 Structured Data Serialization

Structured data serialization is the process of converting a data object into a more appropriate format, normally a stream of bytes, in order to facilitate the process of data storage, that is, saving the object to a database, a file or even to the memory, or to facilitate the transmission of the data object over a network between different systems [9]. Data deserialization is the inverse process of data serialization, transforming the serialized data object into a data object again, ready to be used and manipulated again in the context of a programming language.

The choice of a serialized data format for a certain application depends on multiple different factors, such as data complexity, need for human readability, speed and storage space constraints [10].

Data serialization formats are necessary for microservices to be able to communicate with each other, of which there are two of special relevance to this project: JSON and Protobuf. The next three subsections will focus on each of them, as well as on the format XML, due to its historical importance to data serialization.

2.3.1 XML

XML was created in the mid-1990s at the World Web Consortium (W3C), due to a general dissatisfaction with the current existing formats at the time, such as SGML (Standard Generalized Markup Language), that never achieved wide success due to its complexity and high cost to utilize [11]. The main objective in creating XML was to make a simplified version of SGML, combining the flexibility of SGML with the simplicity of the HTML format [12].

The W3C had several principles in mind when developing XML, principles that still serve as the main core of the data formats used today [12].

- **Form should follow function** - the language should be flexible to adapt to the data, not the inverse.
- **A document should be unambiguous** - every possible state of a document must have only one single interpretation (or be considered invalid).
- **Separate markup from presentation** - document data should be separated from the presentation data in order to allow more flexibility in the presentation of the data.
- **Keep it simple** - the language should be easy to learn in order for it to gain widespread usage.
- **It should enforce maximum error checking** - a document should only be valid if it obeys certain syntax parameters that are enforced to try to prevent errors or data disorganization.
- **It should be culture-agnostic** - the language should support the use of any language and alphabet, not being restricted to only the Latin alphabet and the English language.

```
1  <pet>
2    <id>1</id>
3    <name>Riscas</name>
4    <birthDate>2022-02-01</birthDate>
5    <pet_type>
6      <id>1</id>
7      <name>cat</name>
8    </pet_type>
9    <owner>
10     <id>1</id>
11     <firstName>Paulo</firstName>
12     <lastName>Lopes</lastName>
13     <address>Rua Central</address>
14     <city>Lisboa</city>
15     <telephone>919874563</telephone>
16   </owner>
17 </pet>
```

Listing 2.1: XML code excerpt containing information about a pet

2.3.2 JSON

JSON, short for JavaScript Object Notation, is a data serialization format, whose creation is attributed to Douglas Crawford, even though Crawford admits that he is not the first person to have realized it, although he provided it with a name and a formalized grammar [13].

Eventually after its creation, it would become more popular, due to the ease of implementation and less verbosity (and thus, being more lightweight) compared to the most popular data serialization format of the time, XML [13]. Due to these advantages, JSON soon took XMLs place as the most dominant data serialization format for transferring data between services, a place that it maintains to this day.

JSON, as the name implies, originates from the JavaScript language and is represented using two primary data structures: ordered lists (recognized as arrays) and name/value pairs (recognized as objects) [14].


```
1  {
2    "idPet": 1,
3    "name": "Riscas",
4    "birthDate": "2022-02-01",
5    "petType": {
6      "id": 1,
7      "name": "cat"
8    },
9    "owner": {
10     "id": 1,
11     "firstName": "Paulo",
12     "lastName": "Lopes",
13     "address": "Rua Central",
14     "city": "Lisboa",
15     "telephone": "919874563"
16  }
```

Listing 2.2: JSON code excerpt containing information about a pet

2.3.3 Protobuf

Protobuf, short for Protocol Buffer, is a data serialization format, developed by Google, that is quite different from JSON and other data formats in several respects. It is platform and language independent and the data is serialized to a very efficient binary format, making it smaller and faster than most other data formats, including JSON [15].

However, unlike JSON, it has the characteristic of the serialized data not being human-readable, what can possibly be a disadvantage in certain scenarios, such as not being able to debug a problem by looking at the serialized data directly.

Protobuf's structure is defined in a .proto file, which is a readable text format. This file can then be used to automatically generate code for multiple services that can be using multiple different languages, while ensuring the compatibility between them and the capability of writing and reading data.

A .proto file is composed of multiple data structures called messages, that are themselves composed of multiple elements, each of them representing a certain type of data to be transmitted, each one of them defined by a field type, a name and a field number [16].

Field types can be specified in multiple ways depending on their purpose [16]:

- **Scalar Values**, that are used similarly to primitive types in OO languages, representing simple data like numerical values, strings and booleans.
- **Enums**, used to store a certain value out of a predefined list of possible values. An enum should start with a value tagged with 0, representing the default value in case one its not provided.
- **Nested messages**, a message defined within another message.

```
1  message Pet {
2
3      int32 id = 1;
4      string name = 2;
5      string birth_date = 3;
6
7      enum VaccinationStatus {
8
9          UNDEFINED = 0;
10         VACCINATED = 1;
11         NON_VACCINATED = 2;
12     }
13 }
14
15 message PetCharacteristics {
16
17     string color = 1;
18     string race = 2;
19     int32 weight = 3;
20
21 }
22
23 PetCharacteristics pet_characteristics = 4;
24
25 }
```

Listing 2.3: Protobuf code excerpt containing messages representing the field types mentioned above

Field numbers have to be assigned to each field of a message, according to the following set of rules and recommendations [16]:

- The field numbers used should have a value between 1 and 536,870,911, except for the number range of 19,000 to 19,999 that are reserved to the Protocol Buffers implementation. The protocol buffer compiler will complain if a field number within that range is used in a message.
- The given number must be unique among all fields for that message.
- You cannot use any previously reserved field numbers (numbers that are manually added to a list, impeding their usage, in order to avoid issues that can come from reusing the same field number that was used on a different field in the past) or any field numbers that have been allocated to extensions.
- Field numbers 1 to 15 should be used for the most frequent set out fields, because they only take one byte to encode, compared to the field number range of 16 to 2047, that take two bytes. Therefore, intelligent use of field numbers can reduce the size of the serialized data.

A very important feature to know about the definition of messages in .proto files is the possibility to specify field labels [16]:

- **optional**, used to make the usage of a certain field of a message optional.
- **repeated**, used to make the usage of a certain field repeatable zero or more times (essentially being an array).

- **map**, used to make a certain field encode a paired key/value.

```
1  message Pet {
2
3  int32 id = 1;
4  string name = 2;
5  string birth_date = 3;
6  repeated string visits = 4;
7  optional int32 owner_id = 5;
8  map <int32, string> visit = 6; // int32 stores the number id of a
9                                // visit and string a description of the visit.
10 }
11
```

Listing 2.4: Protobuf code excerpt containing messages representing the field labels mentioned above

If there is no field label assigned to a field, the field is defined as a singular field, having none or exactly one value [16].

```
1  message PetType {
2
3  int32 id = 1;
4  string name = 2;
5
6  }
7
8  message Owner {
9
10 int32 id = 1;
11 string first_name = 2;
12 string last_name = 3;
13 string address = 4;
14 string city = 5;
15 string telephone = 6;
16
17 }
18
19 message Pet {
20
21 int32 id = 1;
22 string name = 2;
23 string birth_date = 3;
24 PetType pet_type = 4;
25
26 }
```

Listing 2.5: Protobuf code excerpt containing messages to serialize information about a pet (representing the same data as in the XML and JSON examples)

2.4 Tools

2.4.1 Postman

Postman is an API platform for building and using APIs, simplifying each step of the API lifecycle and streamlining collaboration so the creation of better APIs can happen faster [17].

Postman provides the following advantages to its users [17]:

- **API Repository:** Easily stores, catalogs, and offers the possibility to collaborate around all API artifacts on one central platform.
- **Tools:** Includes a comprehensive set of tools that help accelerate the API lifecycle: from design, testing, documentation, and mocking to the sharing and discoverability of APIs.
- **Governance:** Postman's full-lifecycle approach to governance lets adopters shift left their development practices, resulting in better-quality APIs.
- **Workspaces:** Postman workspaces help you organize your API work and collaborate across your organization or across the world.
- **Integrations:** Postman integrates with the most important tools in your software development pipeline to enable API-first practices.

The main use of Postman was to test the various methods of the original project, that uses JSON as its data serialization format.

2.4.2 Protoman

Protoman is an API platform inspired by Postman, as its name implies, but focused on Protobuf based messages, being more well-equipped to handle those type of messages than Postman [18].

Protoman provides unique features that make it very well-equipped to handle Protobuf messages, such as [18]:

- Integrating the .proto file in the program, checking its validity in the process.
- The process of sending request bodies coded in Protobuf is made easier by using the .proto file and providing an UI to write correct Protobuf messages.
- Response bodies are shown in the Protoman UI as human readable and structured according to the messages provided in the .proto file.

The screenshot displays the Protoman API client interface. At the top, a dropdown menu is set to 'POST' and the URL 'http://localhost:9966/petclinic/pettyes' is entered. Below this, there are tabs for 'Headers', 'Body', and 'Expected Message', with 'Body' currently selected. Under the 'Body' tab, the 'Request Message' is set to '.org.springframework.samples.petclinic.protobuf.ProtoPetTypeAdd'. The request body is shown in a human-readable Protobuf format:

```
.org.springframework.samples.petclinic.protobuf.ProtoPetTypeAdd {  
  name: horse  
}
```

. Below the request, a horizontal line separates it from the 'Response' section. The 'Response' section also has 'Body' and 'Headers' tabs, with 'Body' selected. The response body is shown in a human-readable Protobuf format:

```
.org.springframework.samples.petclinic.protobuf.ProtoPetType {  
  id: 7  
  name: horse  
}
```

Figure 2.2: API call made with Protoman, showing the request and response body human-readable.

2.4.3 JMeter

The Apache JMeter application is open source software, a 100 per cent pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions [19].

JMeter can be used to simulate a heavy load on a server to test its strength or to analyze overall performance under different load types [19].

In the context of this project, it was used to perform load tests both on the original JSON project chosen and the Protobuf converted one, outputting data that gave relevant information about the differences between both applications.

Chapter 3

From Analysis to Test

This chapter will be dedicated to the following topics:

- Choosing and analyzing a preexisting REST project that uses JSON.
- Developing the chosen project to use Protobuf instead of JSON.
- Testing of both projects.

3.1 Project Analysis and Development

In order to be able to compare two REST projects, an open-source project that uses REST with JSON was chosen to be developed, substituting Protobuf with JSON: Spring PetClinic, specifically a version of it that already uses REST [20].

PetClinic is an application that offers multiple functionalities related to the management of a veterinary clinic, functionalities that have as its main goal to provide the possibility to manipulate data related to the different elements of the problem's domain. This specific version of PetClinic does not provide a UI, only provides a REST API, even though for our purposes the existence of a UI would not be of importance.

In order to be able to develop the project correctly and in the most correct possible way, first I must analyze the current structure of the project.

It is organized in multiple layers, with the following ones being the most important to us:

1. A **model layer**, with all the entities present in the domain problem, using JPA annotations in order to correctly map the entities to the repository.
2. A **mapper layer**, that provides an interface to transfer data objects into DTOs and vice versa.
3. Three different **repository layers** (JPA, Spring JDBC, Spring Data JPA). The one that is used by default is the Spring Data JPA one.
4. Three different **databases** (HSQL, MySQL and PostgreSQL). The one that is used by default is the HSQL one.
5. A **service layer**, responsible for being the intermediary of the controller and repository layers.
6. A **REST controller layer**, using DTOs (Data Transfer Objects) serialized using JSON to receive and send messages to and from external clients.

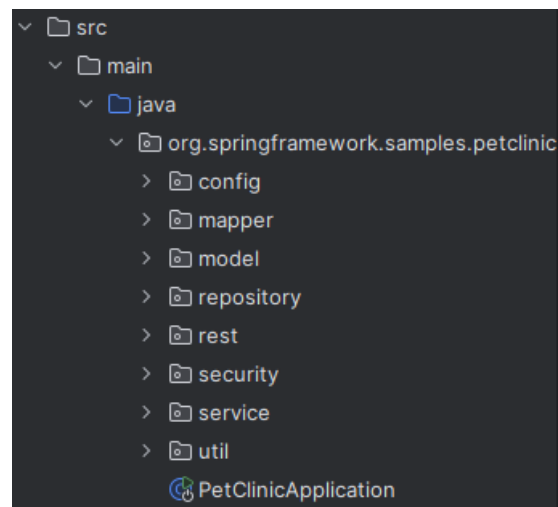


Figure 3.1: Folder structure of the project

Domain Model

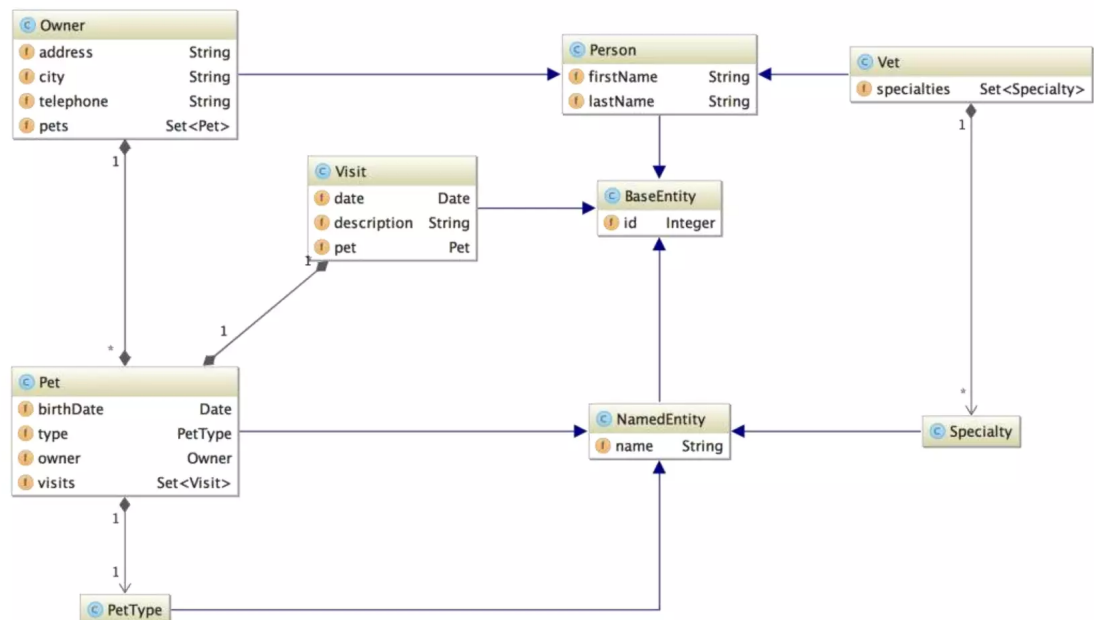


Figure 3.2: PetClinic's Domain Model[21]

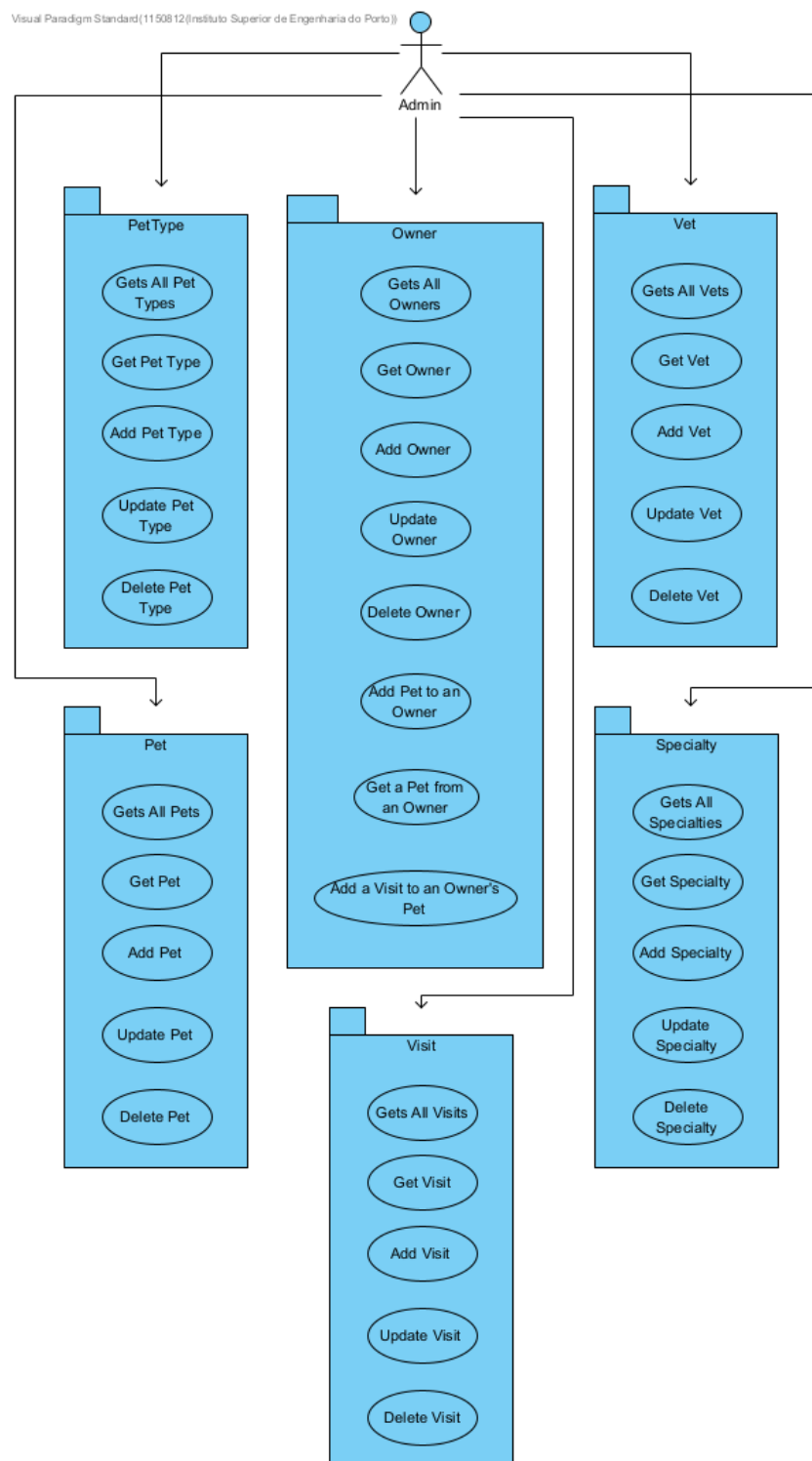


Figure 3.3: Use Case Diagram

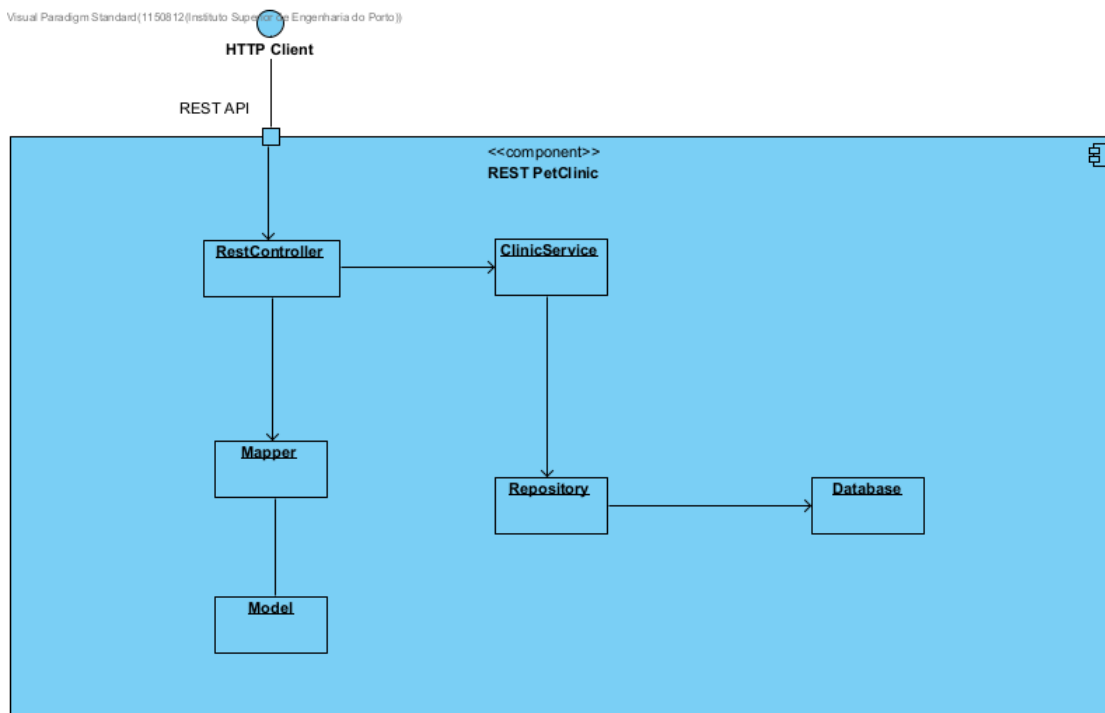


Figure 3.4: Component Diagram

The layer that is most relevant to us and where the bulk of the changes will happen is the REST controller layer. It is divided in six different classes and each one has various methods that I will need to adapt to use Protobuf, for a total of 33 methods. There are other controller classes in the project, but they aren't relevant to the purposes of this project. Here is a comprehensive list of the relevant REST controller classes:

1. PetTypeRestController
2. PetRestController
3. OwnerRestController
4. SpecialtyRestController
5. VetRestController
6. VisitRestController

Each one of these controllers contains their version of the following types of methods:

1. **list** (e.g. listPets) - lists all the existing instances of a certain entity.
2. **get** (e.g. getPet) - returns an instance of a certain entity.
3. **add** (e.g. addPet) - adds an instance of a certain entity.
4. **update** (e.g. updatePet) - updates an instance of a certain entity.
5. **delete** (e.g. deletePet) - deletes an instance of a certain entity.

The REST Controller `OwnerRestController` provides 3 extra methods that also need adaptation:

1. **addPetToOwner** - creates a pet and assigns it to a specific owner.
2. **addVisitToOwner** - creates a visit and assigns to a specific owner's pet.
3. **getOwnersPet** - returns a specific pet belonging to a specific owner.

```
1 public ResponseEntity <List <PetTypeDto>> listPetTypes() {  
2     List<PetType> petTypes = new ArrayList <>(this.clinicService.  
3     findAllPetTypes());  
4     if (petTypes.isEmpty()) {  
5         return new ResponseEntity <>(HttpStatus.NOT_FOUND);  
6     }  
7     return new ResponseEntity <>(petTypeMapper.toPetTypeDtos(petTypes  
, HttpStatus.OK);  
}
```

Listing 3.1: Example of a "list" method (listPetTypes)

```
1 public ResponseEntity <PetTypeDto> addPetType(PetTypeDto petTypeDto)  
2 {  
3     HttpHeaders headers = new HttpHeaders();  
4     if (Objects.nonNull(petTypeDto.getId()) && !petTypeDto.getId().  
5     equals(0)) {  
6         return new ResponseEntity <>(HttpStatus.BAD_REQUEST);  
7     } else {  
8         final PetType type = petTypeMapper.toPetType(petTypeDto);  
9         this.clinicService.savePetType(type);  
10        headers.setLocation(UriComponentsBuilder.newInstance().path(  
11        "/api/pettypes/{id}").buildAndExpand(type.getId()).toUri());  
12        return new ResponseEntity <>(petTypeMapper.toPetTypeDto(type)  
13        , headers, HttpStatus.CREATED);  
14    }  
15 }
```

Listing 3.2: Example of a "add" method (addPetType)

It's important to mention that some of the controller methods have implementation problems in this project, so we won't use them in the tests in the following chapter, but that's not detrimental to the objectives of our project because we don't need to use all the methods to test the differences between the projects, only enough methods that help us understand the differences between JSON and Protobuf. Nevertheless, these are the methods that were identified as faulty:

- **addPet** - it should create a pet that does not have an owner and visits, but the method expects that an owner and visits are associated to the pet created.

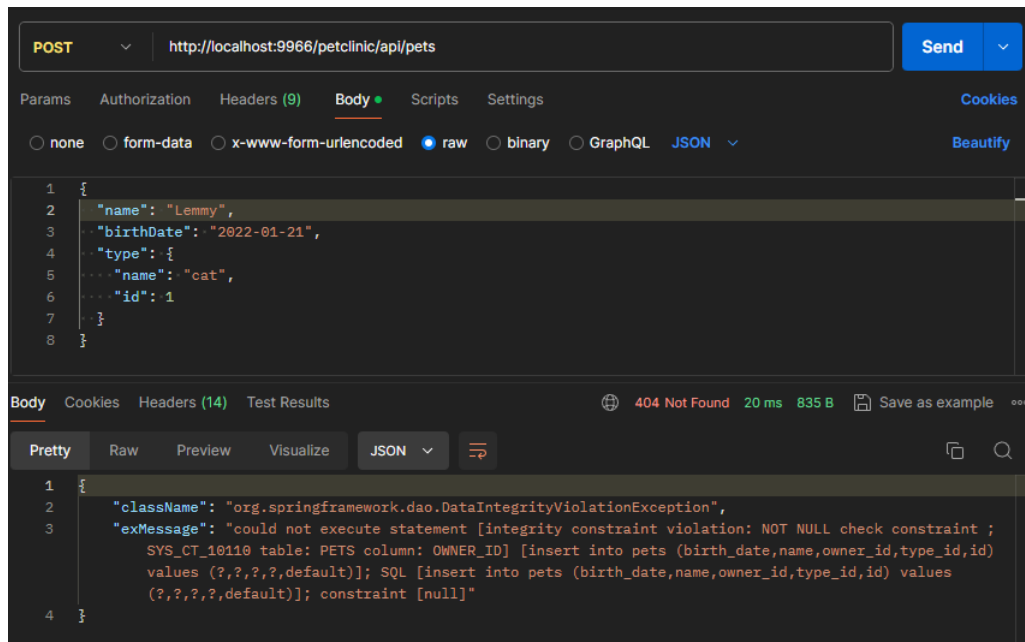


Figure 3.5: Error calling addPet, even when submitting supposedly valid information.

- **getOwnersPet** - this method receives via API call a petId, identifying a certain pet, and an ownerId, identifying a certain owner. Further along in the method it has a check to see if the owner associated to the pet identified by the petId is the same as the owner identified by the ownerId. This check always returns false, even in cases where the check should return true, therefore making this method unusable.

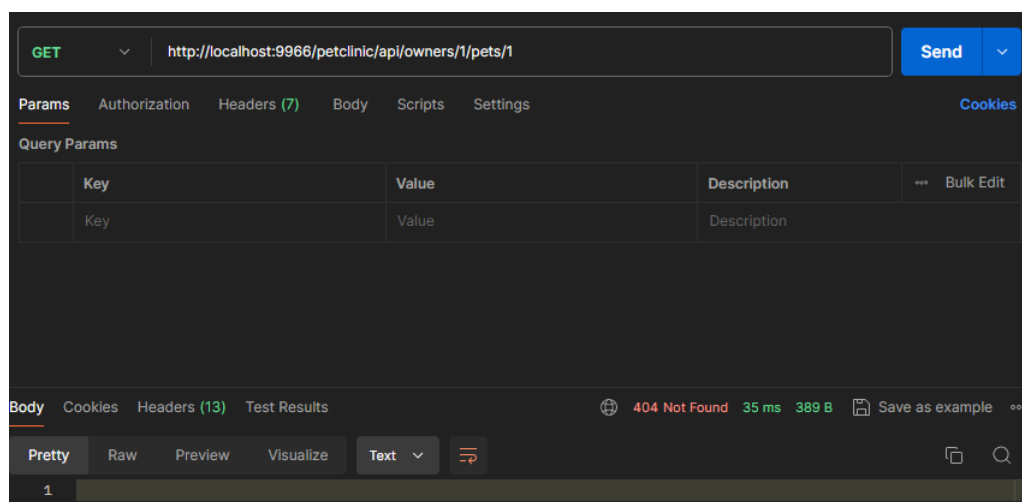


Figure 3.6: getOwnersPet returning 404 Not Found, even though the owner with ownerId '1' owns the pet with petId '1'.

- **deleteSpecialty** - can only delete specialties that aren't associated with a vet.

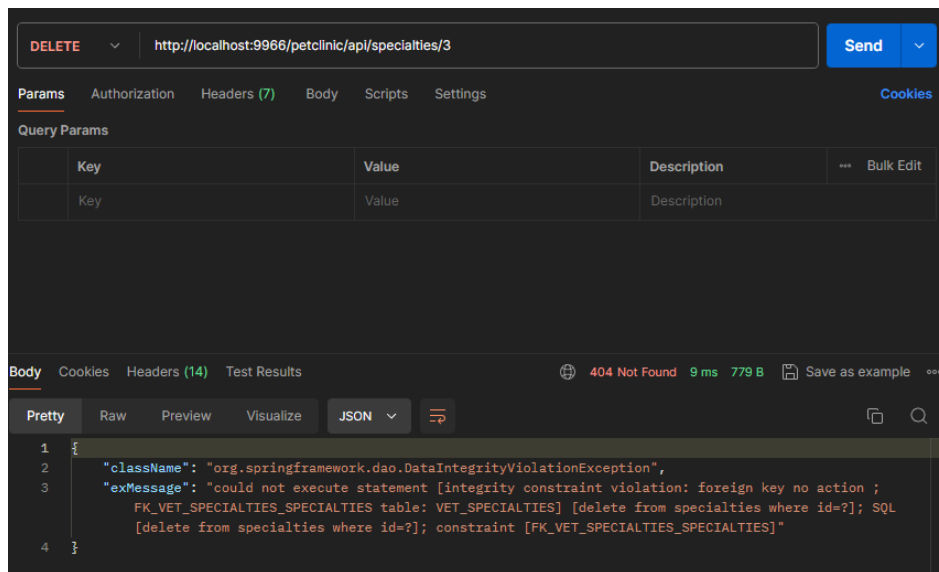


Figure 3.7: Specialty with id '3' exists and should have been able to be deleted by this call.

- **addVisit** - doesn't work in any context.

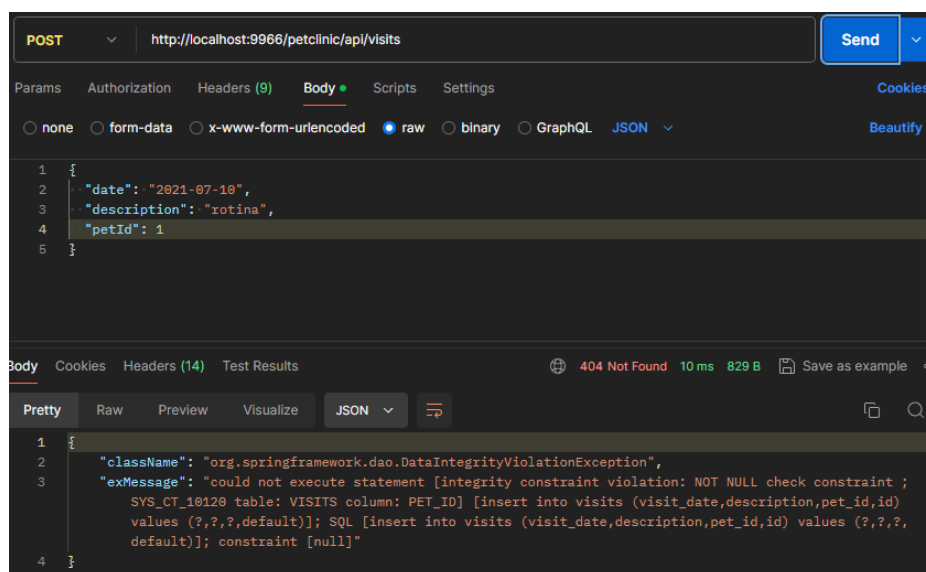


Figure 3.8: Error calling addVisit, even when submitting supposedly valid information.

The following images are sequence diagrams of API calls that are going to be used in the testing part of the project:

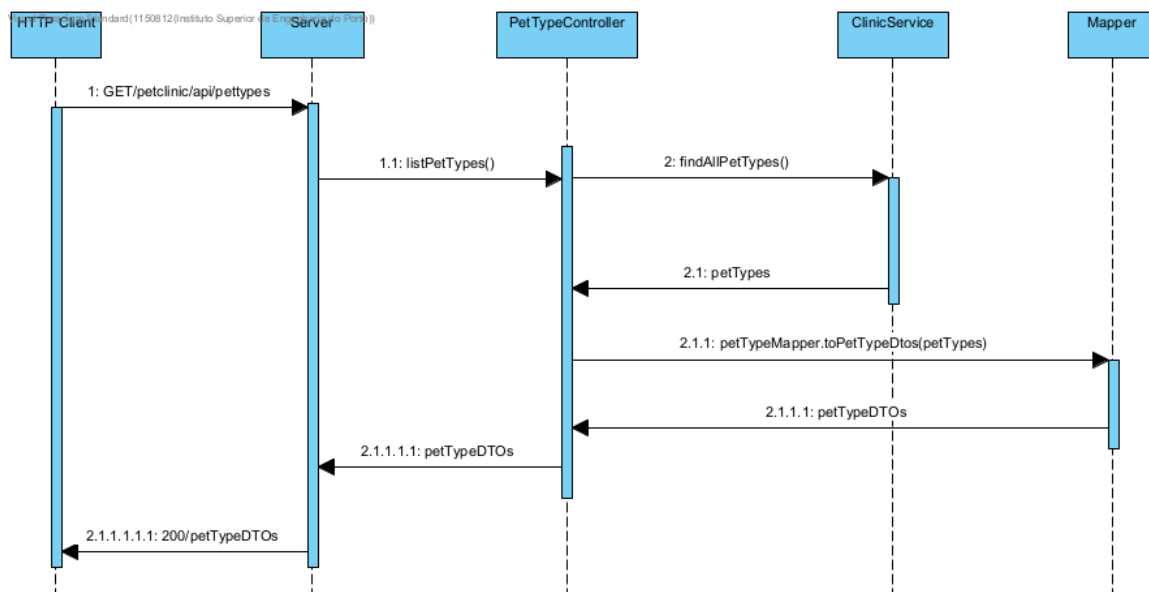


Figure 3.9: Sequence Diagram for listPetTypes

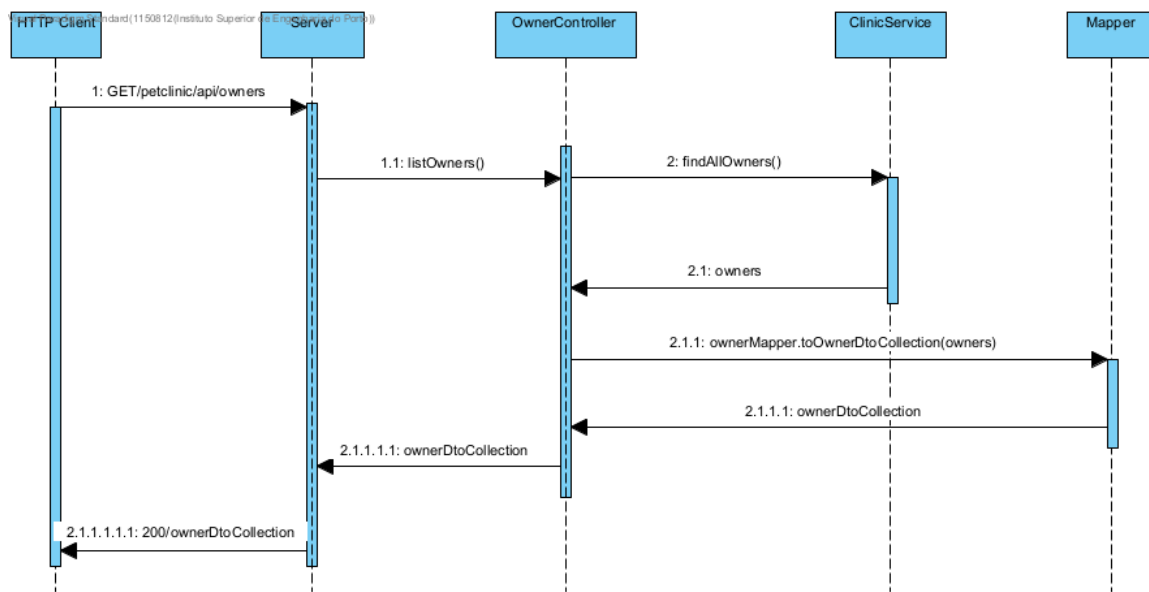


Figure 3.10: Sequence Diagram for listOwners

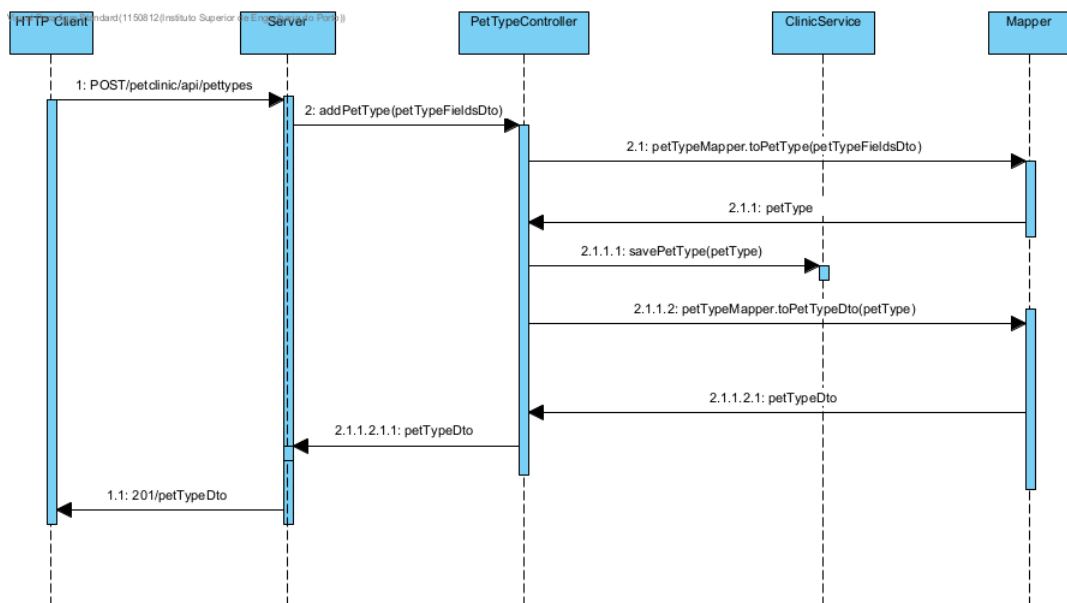


Figure 3.11: Sequence Diagram for addPetType

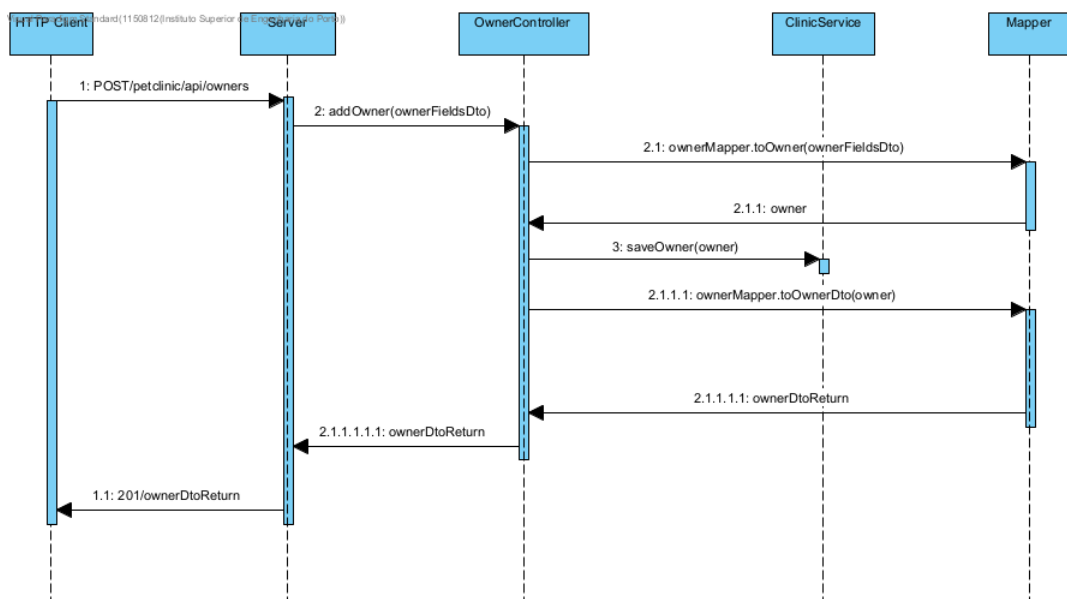


Figure 3.12: Sequence Diagram for addOwner

All the methods of these controllers, as said before, use DTOs serialized using JSON, and that's what we have to adapt, changing them to use Protobuf. To do so, an analysis of the structure of the DTOs used in the controllers must be made in order to later produce Protobuf classes that are in alignment with the existing DTOs.

For each of the six model classes that need to be mapped to a DTO (Pet, PetType, Owner, Vet, Specialty and Visit), there are the following three types of DTO in the project (except Specialty, that only has one DTO):

1. (x)Dto (e.g. PetDto) - DTO that maps all the variables of the respective entity.
2. (x)FieldsDto (e.g. PetFieldsDto) - DTO used to map only the primitive type variables of the respective entity.
3. (x)AllOfDto (e.g. PetAllOfDto) - DTO used to map reference type variables of the respective entity.

```
1 public class OwnerDto {  
2  
3     @JsonProperty("firstName")  
4     private String firstName;  
5  
6     @JsonProperty("lastName")  
7     private String lastName;  
8  
9     @JsonProperty("address")  
10    private String address;  
11  
12    @JsonProperty("city")  
13    private String city;  
14  
15    @JsonProperty("telephone")  
16    private String telephone;  
17  
18    @JsonProperty("id")  
19    private Integer id;  
20  
21    @JsonProperty("pets")  
22    @Valid  
23    private List<PetDto> pets = new ArrayList<>();  
24  
25    //getters and setters...
```

Listing 3.3: Example of a DTO (OwnerDto)

The first step of the practical part of the project is creating a .proto file with an equivalent structure to the DTOs used currently in the project. This structure will be composed of several messages, each one with the objective to encode information that will be received or transmitted in one or more controller methods. The .proto file created has 24 different types of messages, although some are very similar to each other, with only minor differences.

Most of the messages in the .proto file are the same 3 messages, but adapted to each of the 6 model entities, making 18 of the total 24 messages:

1. **Proto(x) (e.g. ProtoPet)** - holds the complete information of the entity in question.
2. **Proto(x)s (e.g. ProtoPets)** - holds the complete information of all instances of the entity in question.
3. **Proto(x)Add (e.g. ProtoPetAdd)** - message to add an instance of the entity in question.

There are other 6 messages that are related to more specific circumstances:

1. **ProtoPetOwner** - holds the basic information about the owner of a certain pet.
2. **ProtoPetVisit** - holds the basic information about a visit of a certain pet.
3. **ProtoOwnerPet** - holds the basic information about a pet of a certain owner.
4. **ProtoOwnerFindByName** - holds just the name of an owner to be searched for in the database.
5. **ProtoOwnerAddPet** - message to create and add a pet owned by a specific owner.
6. **ProtoOwnerAddVisit** - message to add a visit to a certain owner and pet.

```
1  message ProtoPet {
2
3      int32 id = 1;
4      string name = 2;
5      string birth_date = 3;
6      string pet_type = 4;
7      ProtoPetOwner owner = 5;
8      repeated ProtoPetVisit visits = 6;
9
10 }
11
12 message ProtoPets {
13
14     repeated ProtoPet pets = 1;
15
16 }
```

Listing 3.4: Examples of the messages above for the entity Pet

```

1  message ProtoPetAdd {
2
3      string name = 1;
4      string birth_date = 2;
5      int32 pet_type_id = 3;
6      int32 owner_id = 4;
7
8  }
9
10 message ProtoPetOwner {
11
12     string first_name = 1;
13     string last_name = 2;
14     string address = 3;
15     string city = 4;
16     string telephone = 5;
17
18 }
19
20 message ProtoPetVisit {
21
22     int32 id = 1;
23     string date = 2;
24     string description = 3;
25
26 }

```

Listing 3.5: Examples of more of the messages above for the entity Pet

This .proto file will then be compiled by a compiler called **protoc** that will generate several java classes, one for each message, that will then be used to replace the DTOs that we saw on the previous chapter, used in the methods of the REST controllers.

```

1  @RequestMapping("listPetTypes")
2  public ResponseEntity<ProtoPetTypes> listPetTypes() {
3      List<PetType> petTypes = new ArrayList<>(this.clinicService .
4      findAllPetTypes());
5      if (petTypes.isEmpty()) {
6          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
7      }
8
9      List<ProtoPetType> collection = new ArrayList<>();
10
11     for (PetType petType : petTypes) {
12         ProtoPetType petTypeProto = ProtoPetType.newBuilder().setId(
13         petType.getId()).setName(petType.getName()).build();
14         collection.add(petTypeProto);
15     }
16
17     ProtoPetTypes lp = ProtoPetTypes.newBuilder().addAllPetTypes(
18     collection).build();
19
20     return new ResponseEntity<>(lp, HttpStatus.OK);
21 }

```

Listing 3.6: Method listPetTypes now using Protobuf

```

1  @RequestMapping("getPetType/{petTypeId}")
2  public ResponseEntity<ProtoPetType> getPetType(@PathVariable("
petTypeId") Integer petTypeId) {
3      PetType petType = this.clinicService.findPetTypeById(petTypeId);
4      if (petType == null) {
5          return new ResponseEntity<>(HttpStatus.NOT_FOUND);
6      }
7
8      ProtoPetType petTypeProto = ProtoPetType.newBuilder().setId(
petType.getId()).setName(petType.getName()).build();
9
10     return new ResponseEntity<>(petTypeProto, HttpStatus.OK);
11 }
12

```

Listing 3.7: Method getPetType now using Protobuf

Modifications similar to the ones represented above were made for each one of the methods, taking into consideration the fact the data transmitted and received should be equivalent between the JSON project and the Protobuf project, as well as HTTP status returned.

Testing of these modifications was made with a tool called Protoman, a similar program to the more popular Postman, but designed to be easier to work with Protobuf messages.

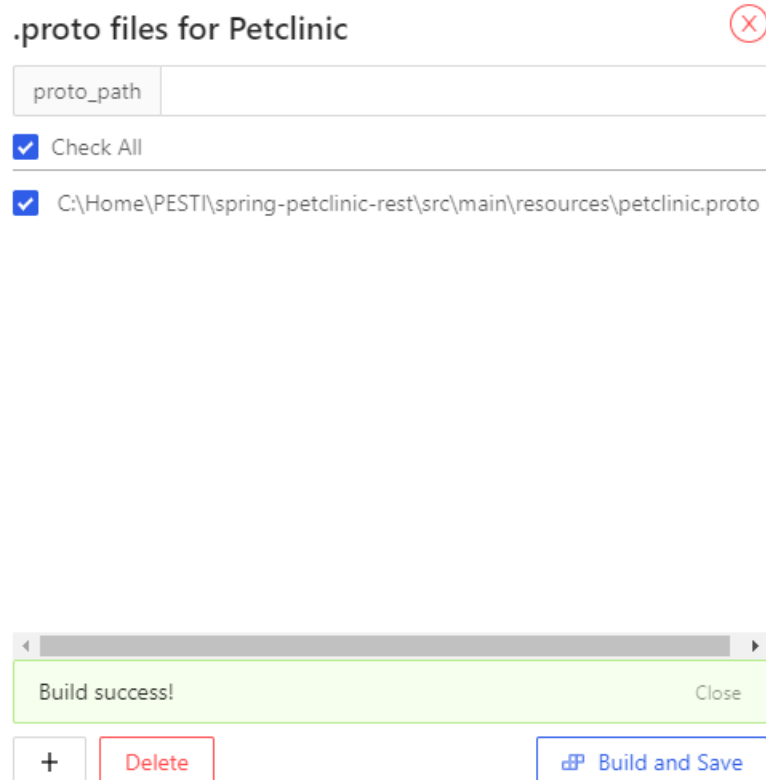


Figure 3.13: .proto file used in the Protobuf project being integrated into Protoman

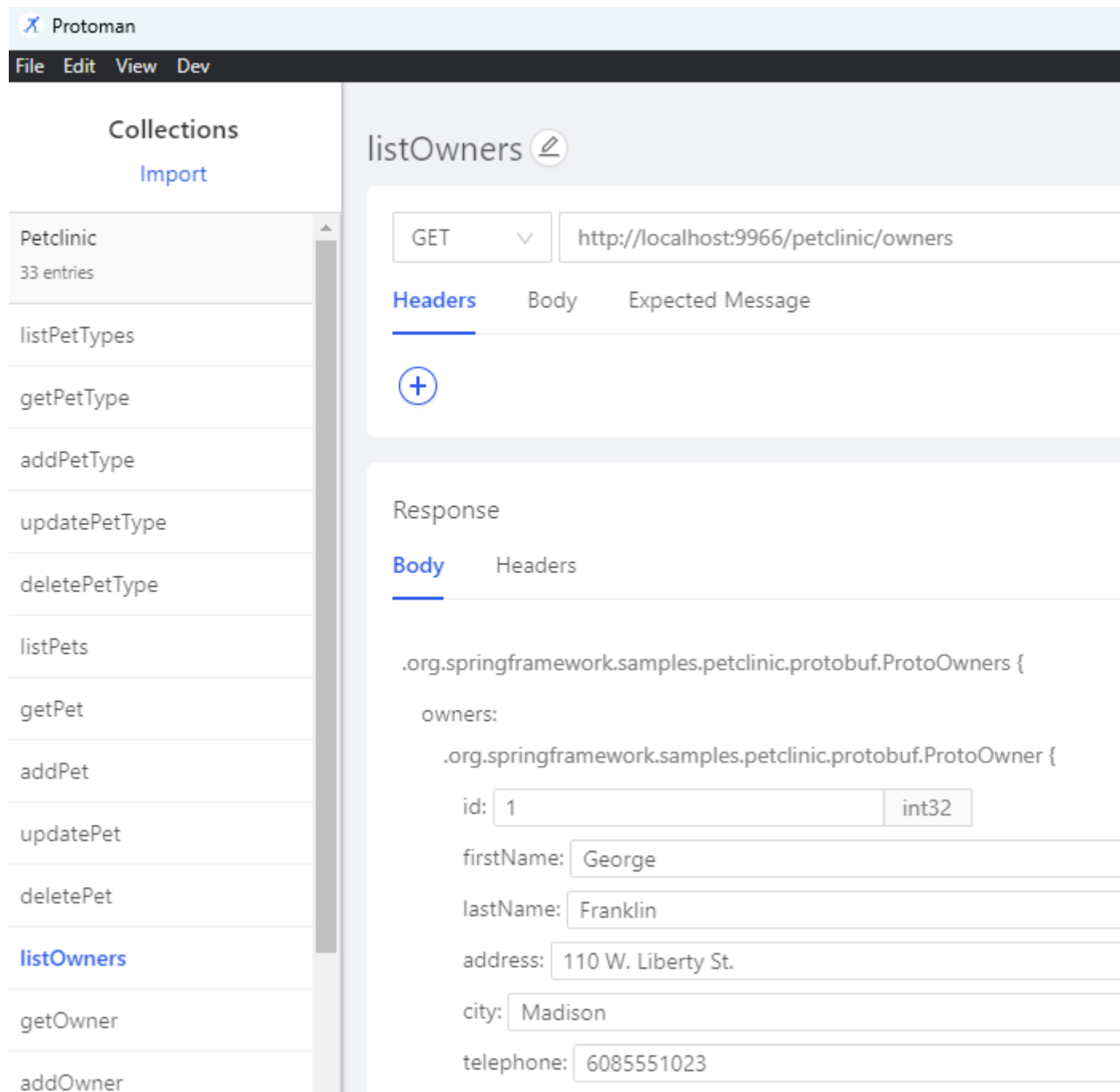


Figure 3.14: Example of an API call in Protoman and the request collection in the sidebar

3.2 Testing

This section will be dedicated to the testing of both the JSON and Protobuf projects, using JMeter as the main tool to obtain results to then analyze and take conclusions from.

3.2.1 Tests Configuration

Four different types of tests were designed in order to obtain the results necessary to make an analysis:

- **Baseline** - provides information about how the applications handle a minimal load.
- **Load** - provides information about how the applications handle what would be considered an average sized load.
- **Stress** - provides information about how the applications handle a huge load, representing what would be an application having peak traffic.
- **Soak** - provides information about how the applications handle a large load scaling over time.
- **Data Creation Tests** - provide information about how the applications handle POST requests, as well as creating necessary data to perform the other types of tests.

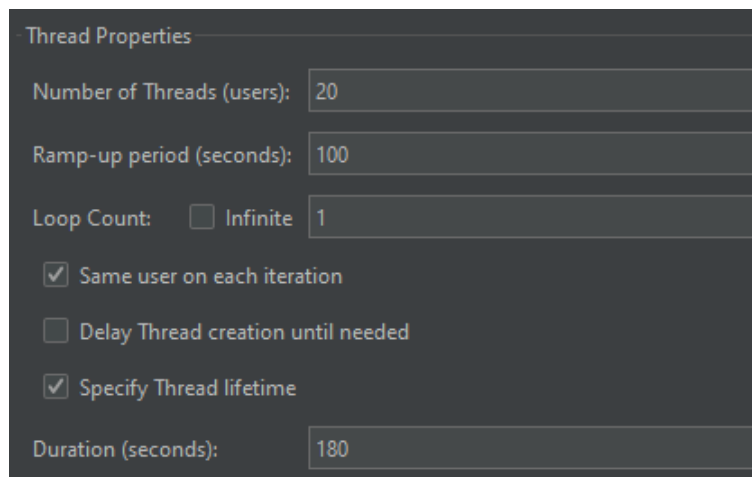
Four different API calls were chosen to be used in the tests:

- **addPetType** - adds a new pet type to the database
- **addOwner** - adds a new owner to the database
- **listPetTypes** - lists all pet types existent in the database
- **listOwners** - lists all owners existent in the database

The first two calls are used in the data creation tests and the other two will be used in the Baseline, Load, Stress and Soak tests.

As said before, JMeter was the application chosen to configure and perform these tests. Tests in JMeter are associated with what is called a Thread Group, that provide the following relevant configuration options to configure our tests:

- **Number of Threads** - represents the number of threads (or users) that will be connected to our applications and perform requests to them.
- **Ramp-up Period** - represents the number of seconds that will take to initialize all threads or users, using the ratio between the number of threads and the ramp-up period as the metric to separate the initialization of each thread (e.g. if there are 20 threads and the ramp-up period is 100 seconds, the time that will take to JMeter to initialize each thread is $100/20 = 5$ seconds).
- **Loop Count** - the number of times the process must be repeated (default is 1).
- **Duration** - represents the maximum number of seconds that the tests of a certain thread group can run.

The image shows a 'Thread Properties' dialog box with a dark background. It contains several input fields and checkboxes. The 'Number of Threads (users):' field is set to 20. The 'Ramp-up period (seconds):' field is set to 100. The 'Loop Count:' section has an unchecked 'Infinite' checkbox and a text field set to 1. Below this, there are three checkboxes: 'Same user on each iteration' (checked), 'Delay Thread creation until needed' (unchecked), and 'Specify Thread lifetime' (checked). At the bottom, the 'Duration (seconds):' field is set to 180.

Thread Properties	
Number of Threads (users):	20
Ramp-up period (seconds):	100
Loop Count:	<input type="checkbox"/> Infinite 1
<input checked="" type="checkbox"/> Same user on each iteration	
<input type="checkbox"/> Delay Thread creation until needed	
<input checked="" type="checkbox"/> Specify Thread lifetime	
Duration (seconds):	180

Figure 3.15: Example of a Thread Group Configuration

Each Thread Group contains tests associated with it, each one of them responsible for making an API call to either the original JSON project or the Protobuf project. Associated with each test are what JMeter calls "Listeners", responsible for gathering data of the tests performed. In the configuration made in JMeter, three types of Listeners ended up being used:

- **View Results Tree** - displays all the information of each API call made during the running of the tests.
- **Simple Data Writer** - writes test results to a certain predesignated file.
- **Summary Report** - displays relevant information relative to each different request present in the tests.

In the Data Creation Tests, two more elements are present in each test, a "Pre Processor" and a "HTTP Header Manager". The Pre Processor element is used to dynamically create a different request body for each request made, and the HTTP Header Manager to add to the request headers the type of content expected in the request body (application/json in the case of the original JSON project and application/x-protobuf in the case of the Protobuf Project).

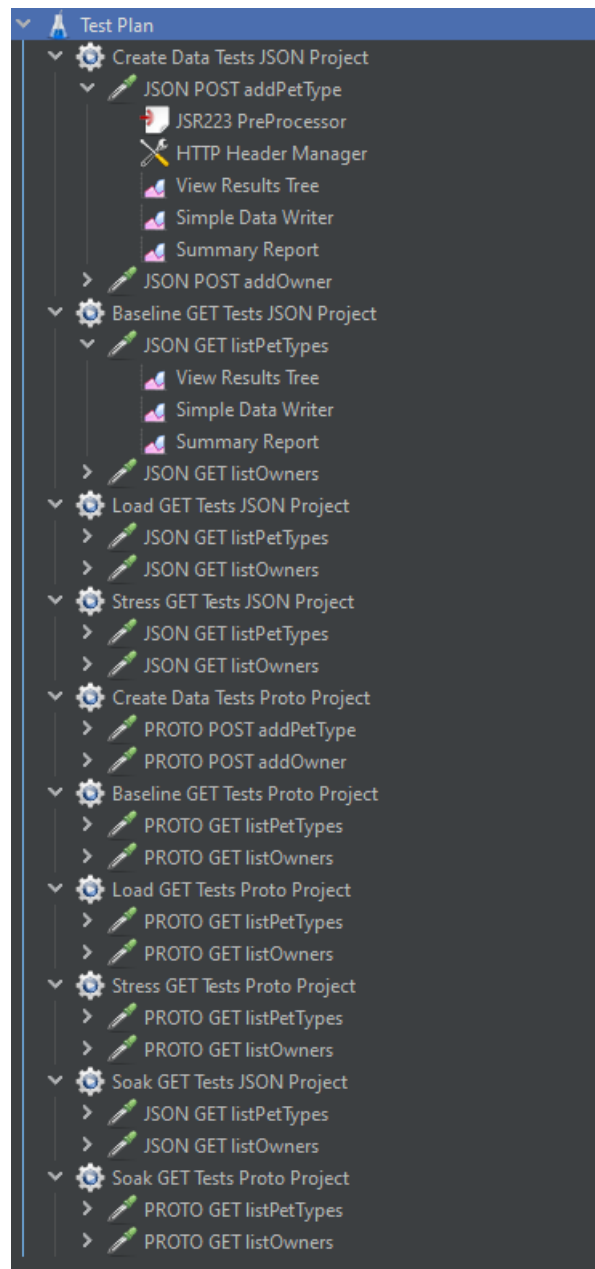
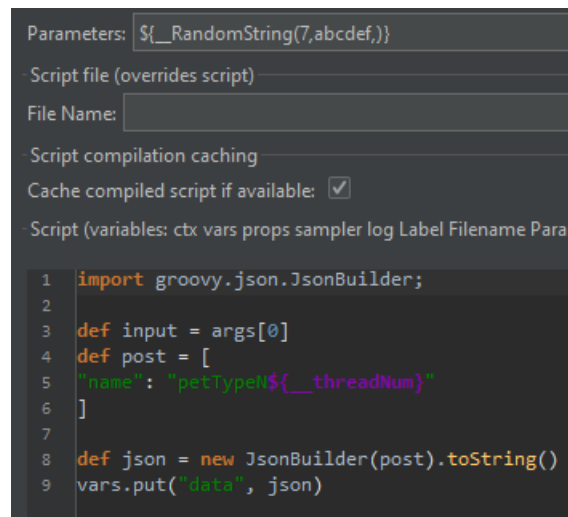


Figure 3.16: JMeter's Test Plan Structure

3.2.2 Data Creation Tests

The data creation tests are responsible to measure performance metrics of both projects when handling POST requests. As said before, the API calls invoked are `addPetType` and `addOwner`, that add new pet types and owners to the database, respectively. The created data will also be used in the other tests.

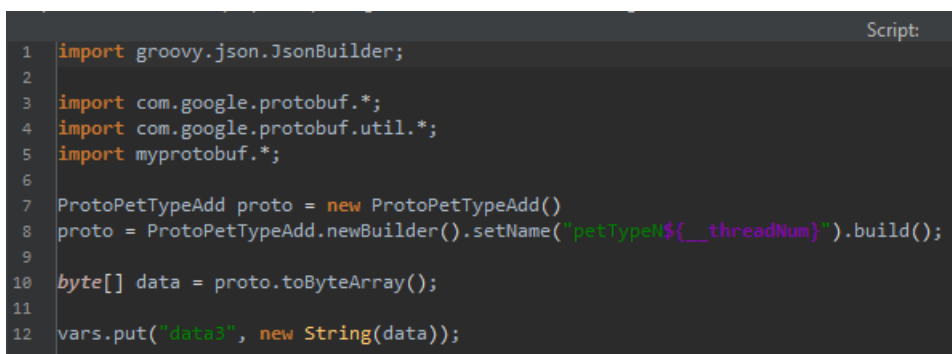
In order to construct a dynamic request body for each request, two scripts were developed in order to accomplish that, one for each project, using the JSR223 Pre Processor. The differences between the bodies of the different requests are accomplished by using variables that change dynamically for each request, such as a random generated string or using the thread number of that request as part of the request body. A few small Java archives (.jar) were added to JMeter, containing the necessary code to create Protobuf messages compatible with what the Protobuf project expects to receive in the request body.



The screenshot shows a JMeter JSR223 Pre Processor configuration. The 'Parameters' field contains `${__RandomString(7,abcdef,)}`. The 'Script file (overrides script)' field is empty. The 'File Name' field is empty. The 'Script compilation caching' section has 'Cache compiled script if available' checked. The script content is as follows:

```
1 import groovy.json.JsonBuilder;
2
3 def input = args[0]
4 def post = [
5   "name": "petType${__threadNum}"
6 ]
7
8 def json = new JsonBuilder(post).toString()
9 vars.put("data", json)
```

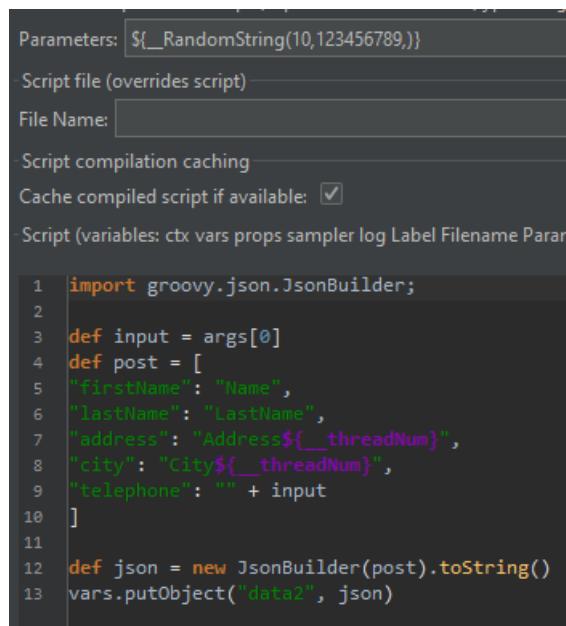
Figure 3.17: Script to create JSON body for `addPetType` call



The screenshot shows a JMeter JSR223 Pre Processor configuration. The 'Script' field contains the following code:

```
1 import groovy.json.JsonBuilder;
2
3 import com.google.protobuf.*;
4 import com.google.protobuf.util.*;
5 import myprotobuf.*;
6
7 ProtoPetTypeAdd proto = new ProtoPetTypeAdd()
8 proto = ProtoPetTypeAdd.newBuilder().setName("petType${__threadNum}").build();
9
10 byte[] data = proto.toByteArray();
11
12 vars.put("data", new String(data));
```

Figure 3.18: Script to create Protobuf body for `addPetType` call



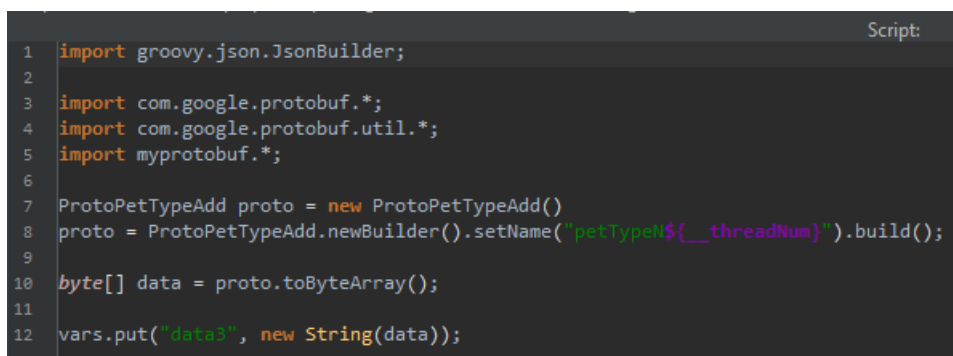
```

Parameters: ${__RandomString(10,123456789,)}
Script file (overrides script)
File Name:
Script compilation caching
Cache compiled script if available: ☒
Script (variables: ctx vars props sampler log Label Filename Param

1 import groovy.json.JsonBuilder;
2
3 def input = args[0]
4 def post = [
5   "firstName": "Name",
6   "lastName": "LastName",
7   "address": "Address:${__threadNum}",
8   "city": "City:${__threadNum}",
9   "telephone": "" + input
10 ]
11
12 def json = new JsonBuilder(post).toString()
13 vars.putObject("data2", json)

```

Figure 3.19: Script to create JSON body for addOwner call



```

Script:
1 import groovy.json.JsonBuilder;
2
3 import com.google.protobuf.*;
4 import com.google.protobuf.util.*;
5 import myprotobuf.*;
6
7 ProtoPetTypeAdd proto = new ProtoPetTypeAdd()
8 proto = ProtoPetTypeAdd.newBuilder().setName("petType${__threadNum}").build();
9
10 byte[] data = proto.toByteArray();
11
12 vars.put("data3", new String(data));

```

Figure 3.20: Script to create Protobuf body for addOwner call

The data creation tests were configured with the following values:

- **Number of Threads** - 200
- **Ramp-up period** - 15
- **Loop Count** - 5
- **Duration** - 180

The test results (next four figures) show that there isn't a relevant difference between the results of both projects, with the only discrepancies coming from a difference in elapsed time in the Max request response time relative to each project. Nevertheless, the mean elapsed time and throughput and the main relevant metrics, and they are extremely similar between both projects.

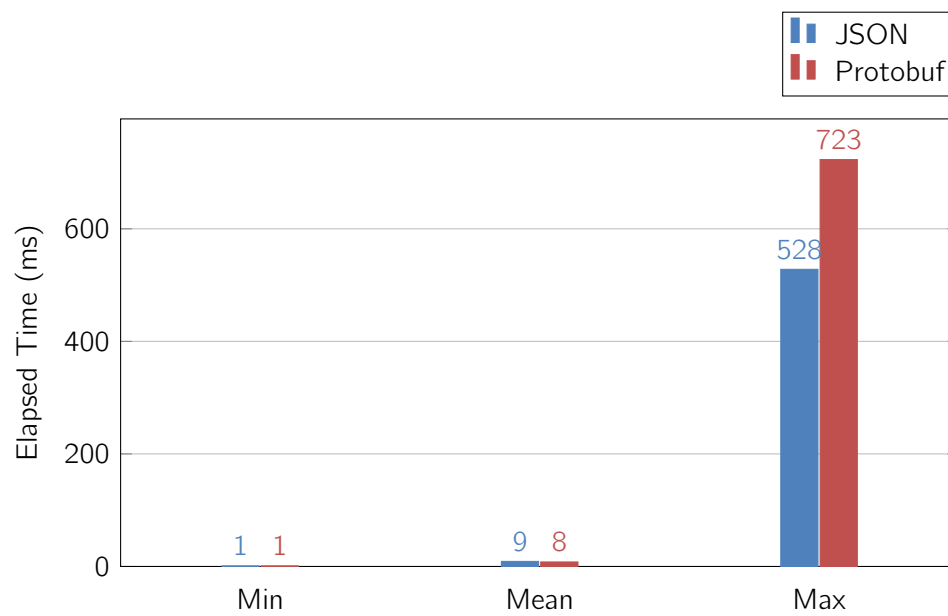


Figure 3.21: POST addPetType Elapsed Time

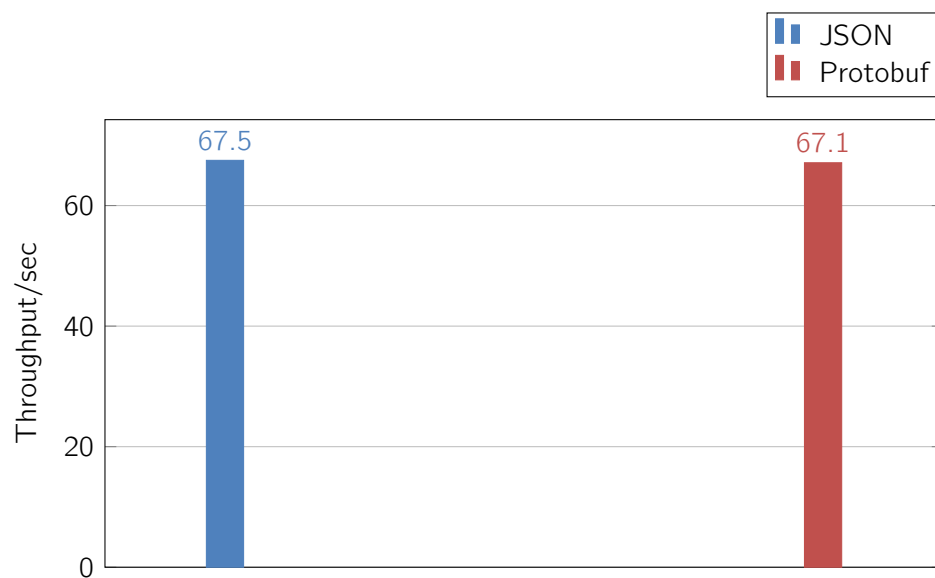


Figure 3.22: POST addPetType Throughput

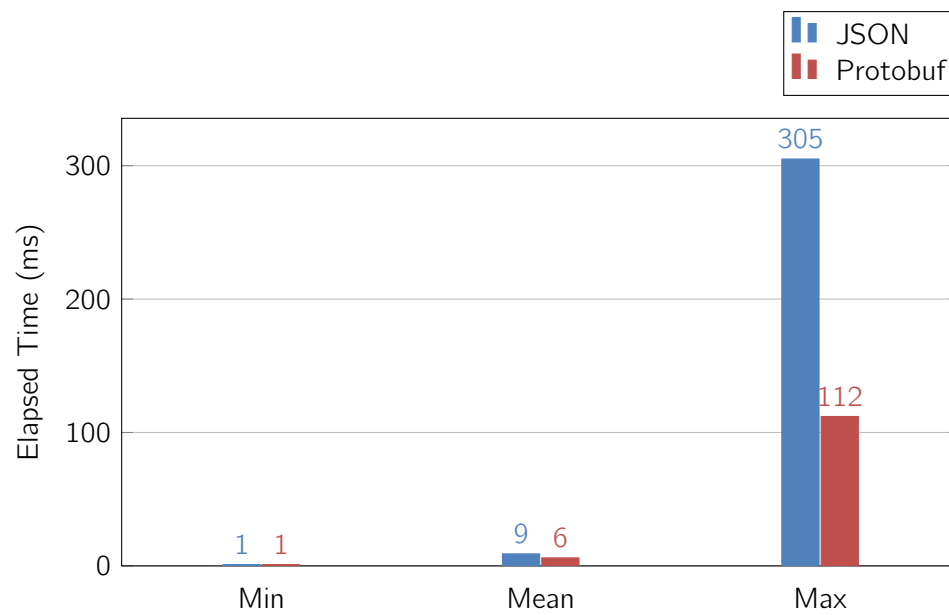


Figure 3.23: POST addOwner Elapsed Time

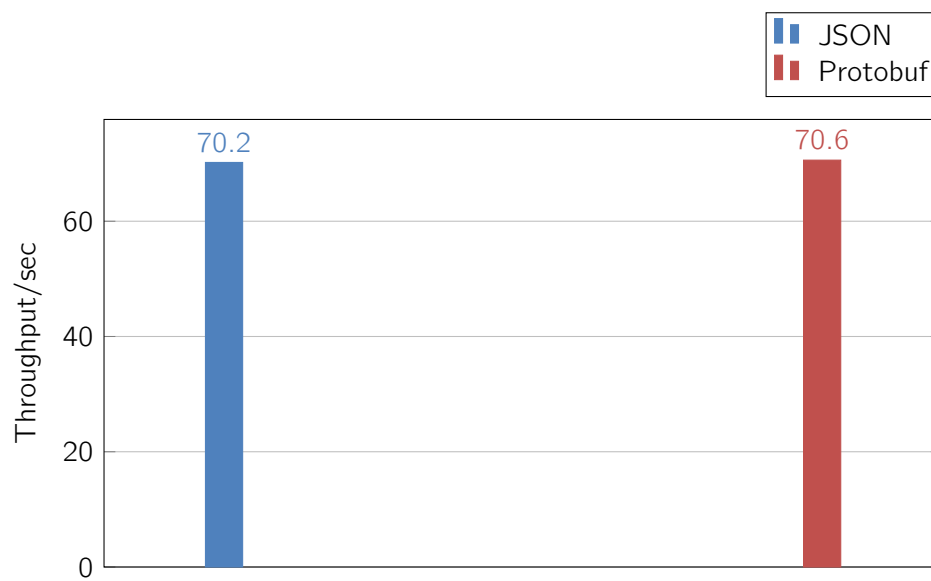


Figure 3.24: POST addOwner Throughput

3.2.3 Baseline Tests

The baseline tests were configured with the following values:

- **Number of Threads** - 20
- **Ramp-up period** - 10
- **Loop Count** - 1
- **Duration** - 180

The test results (next four figures) show that both projects have extremely similar performances when submitted to these tests, with the only small discrepancy being the Max request response time; the mean elapsed time and throughput metrics are very similar.

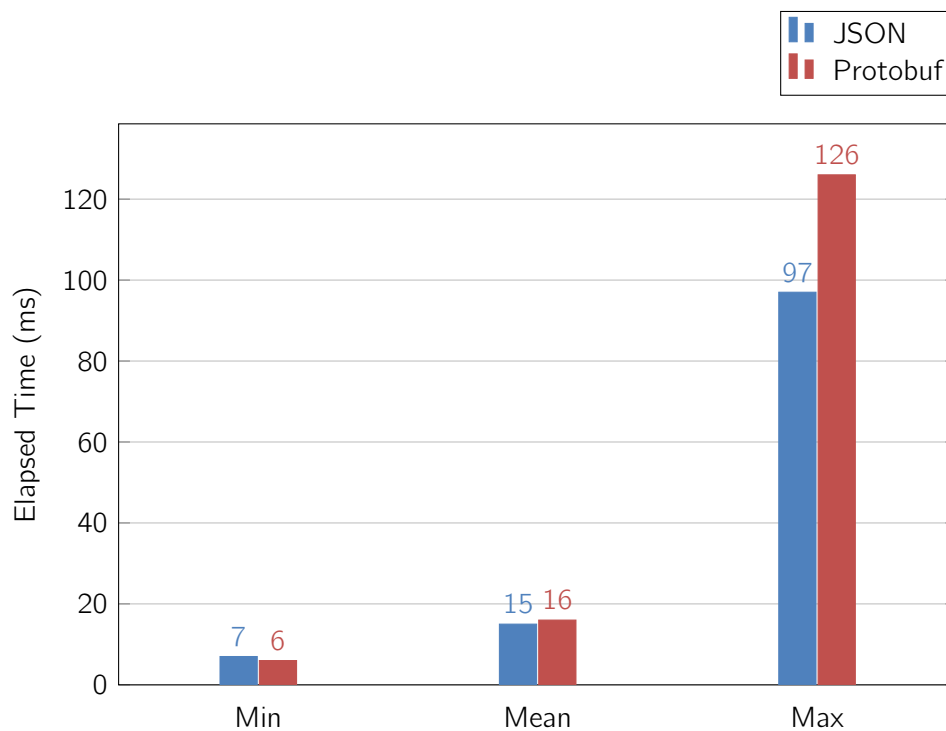


Figure 3.25: Baseline GET listPetTypes Elapsed Time

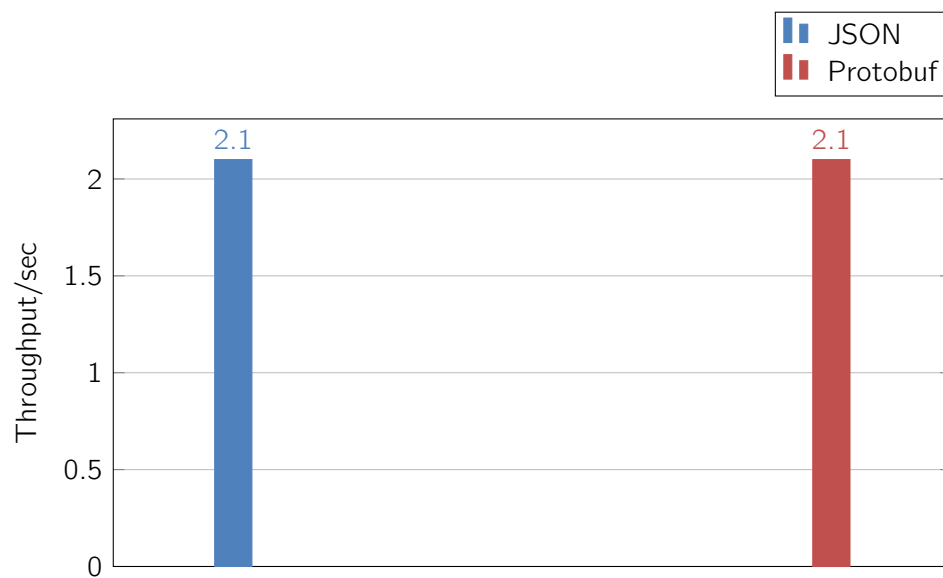


Figure 3.26: Baseline GET listPetTypes Throughput

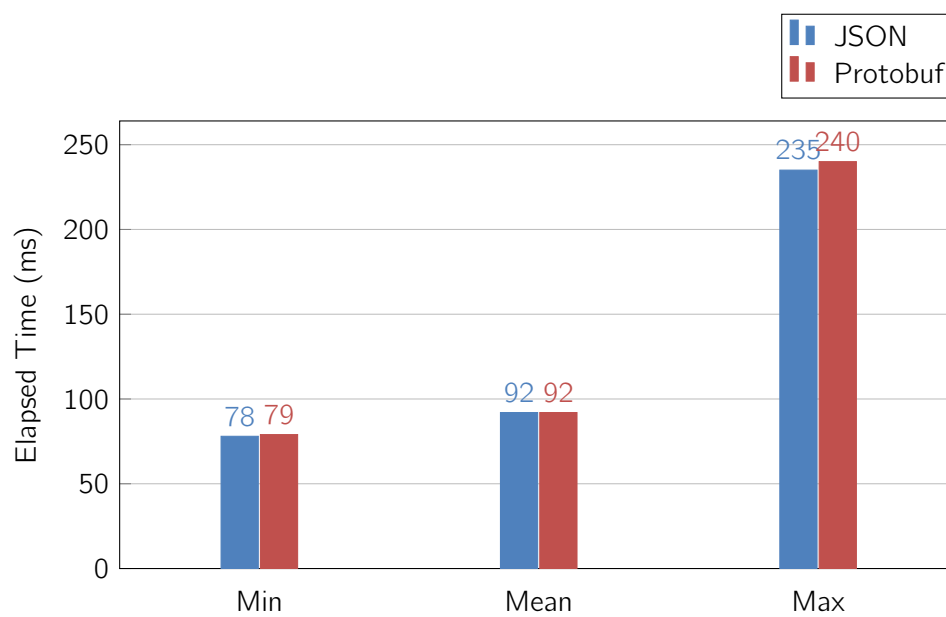


Figure 3.27: Baseline GET listOwners Elapsed Time

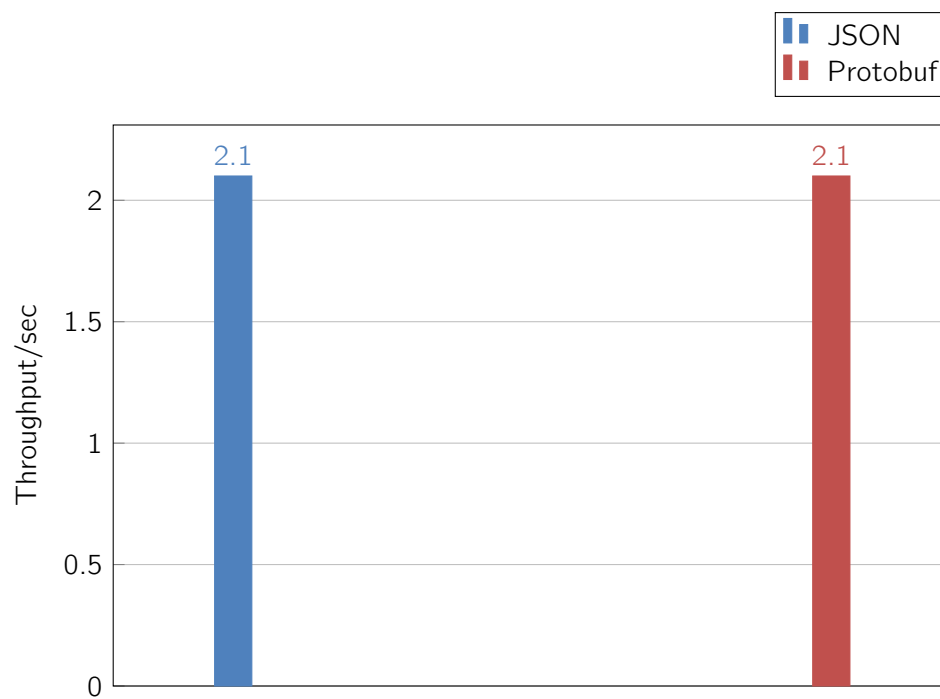


Figure 3.28: Baseline GET listOwners Throughput

3.2.4 Load Tests

The load tests were configured with the following values:

- **Number of Threads** - 500
- **Ramp-up period** - 30
- **Loop Count** - 1
- **Duration** - 300

The test results (next four figures) show, once more, that both projects have extremely similar performances when submitted to these tests, with again the only small discrepancy being the Max request response time; the mean elapsed time and throughput metrics are very similar, as in the baseline tests.

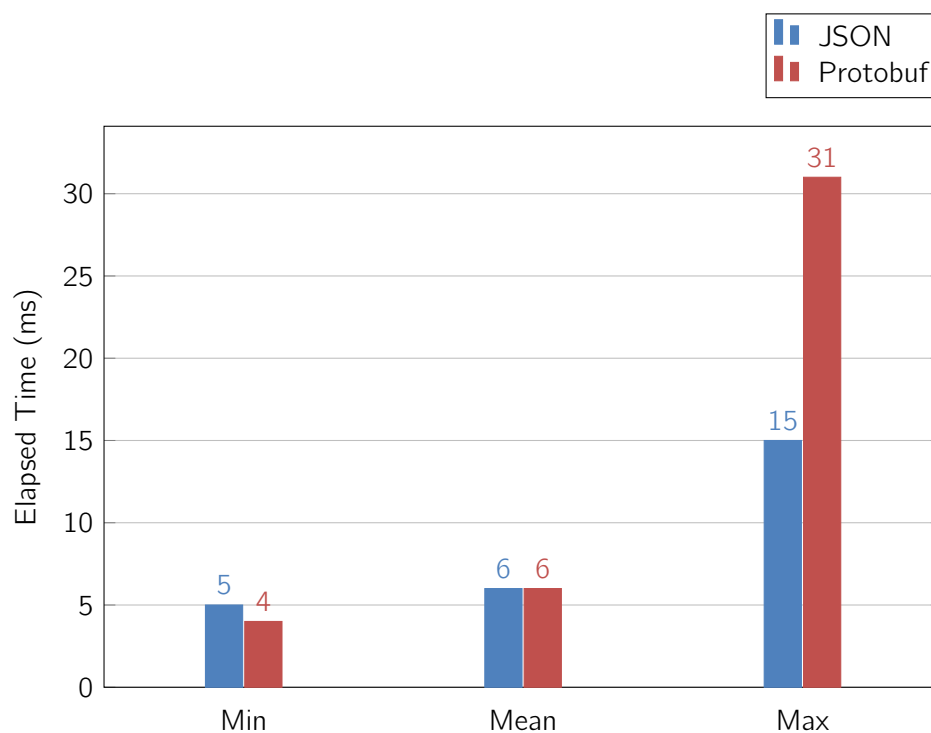


Figure 3.29: Load GET listPetTypes Elapsed Time

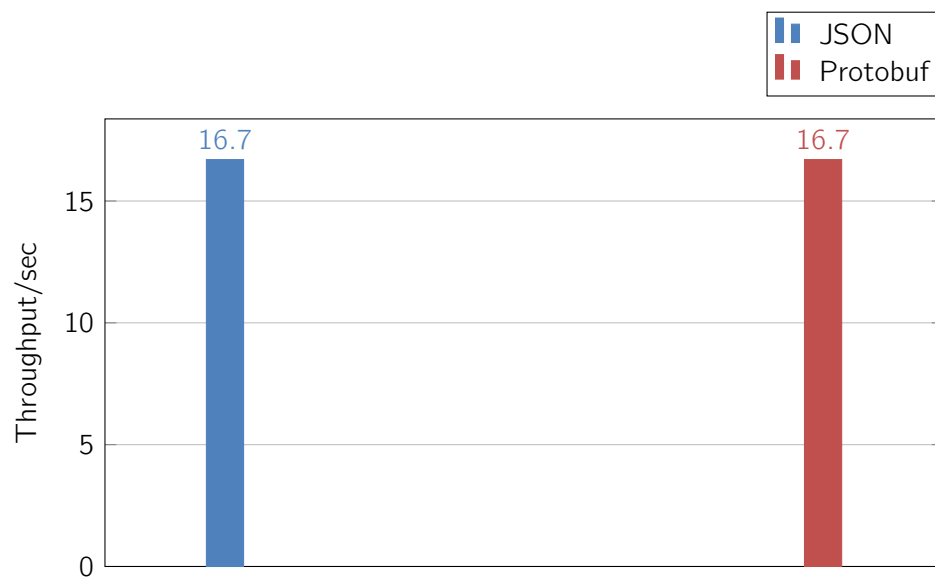


Figure 3.30: Load GET listPetTypes Throughput

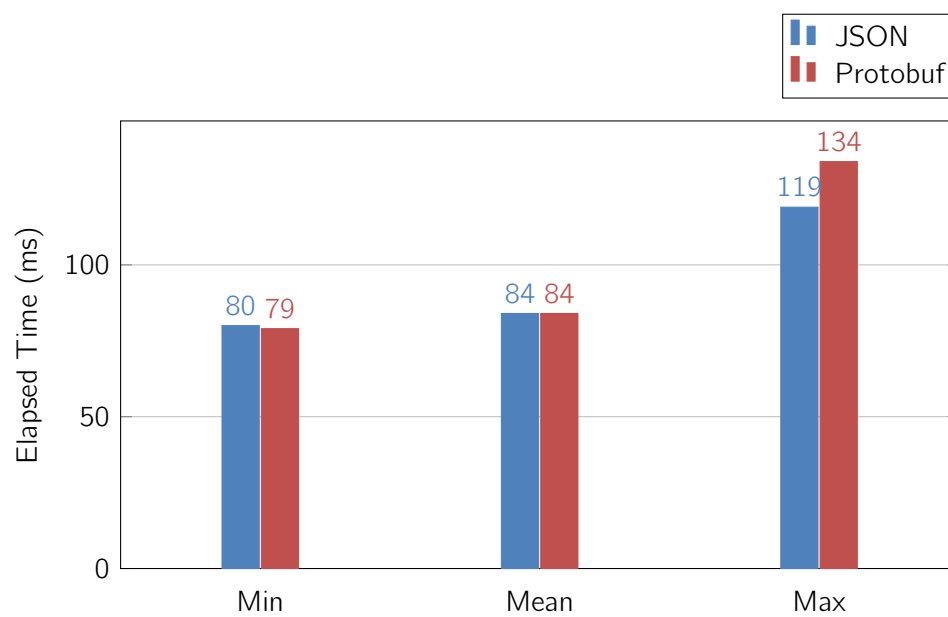


Figure 3.31: Load GET listOwners Elapsed Time

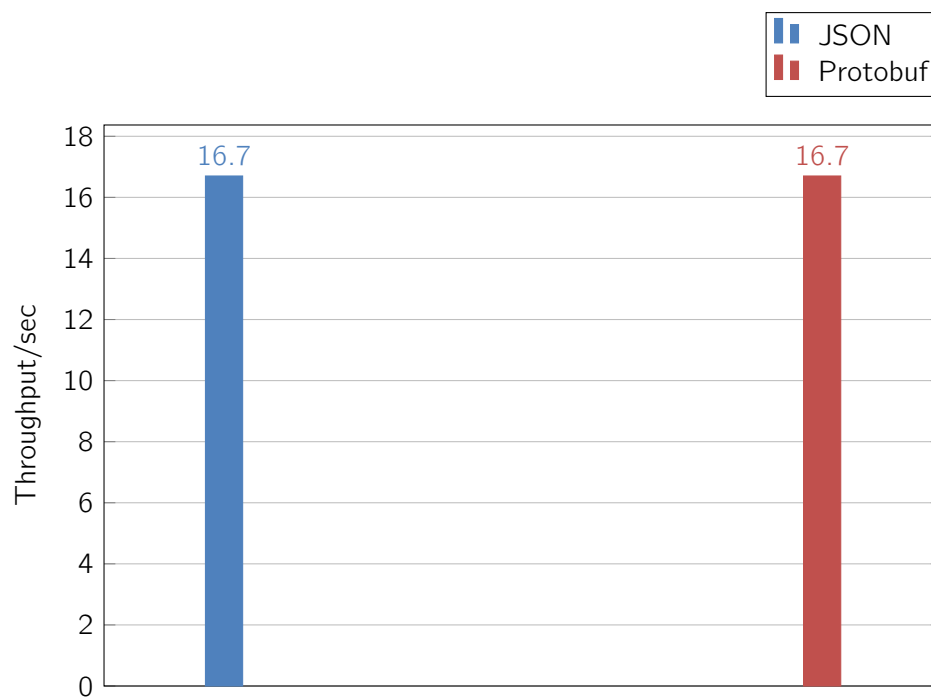


Figure 3.32: Load GET listOwners Throughput

3.2.5 Stress Tests

The stress tests were configured with the following values:

- **Number of Threads** - 3500
- **Ramp-up period** - 100
- **Loop Count** - 1
- **Duration** - 500

The test results (next four figures) show significant differences between the JSON and Protobuf projects. The Protobuf project clearly outperforms the JSON project, having a slightly larger throughput than the JSON project but, above all, having much smaller request response times. The fastest Protobuf request response is approximately twice as fast as the fastest JSON request response and the slowest request response was almost three times faster in Protobuf than in JSON. The difference in the mean request response time is even larger, results showing that GET listPetTypes is approximately eight times faster in Protobuf than in JSON and approximately six times faster in Protobuf for GET listOwners.

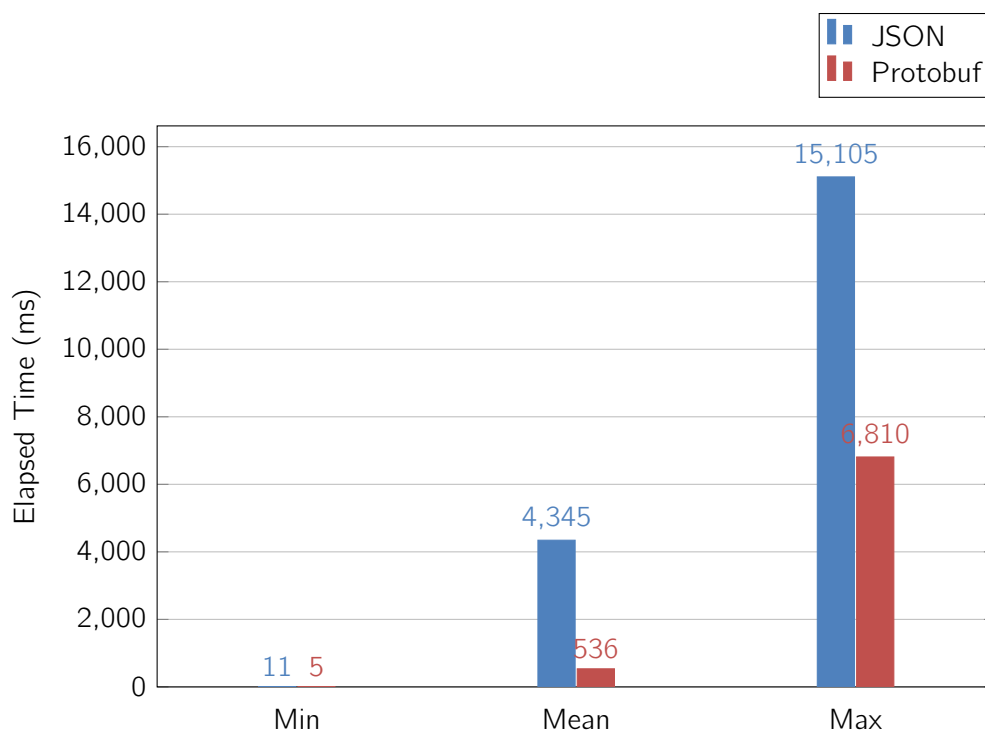


Figure 3.33: Stress GET listPetTypes Elapsed Time

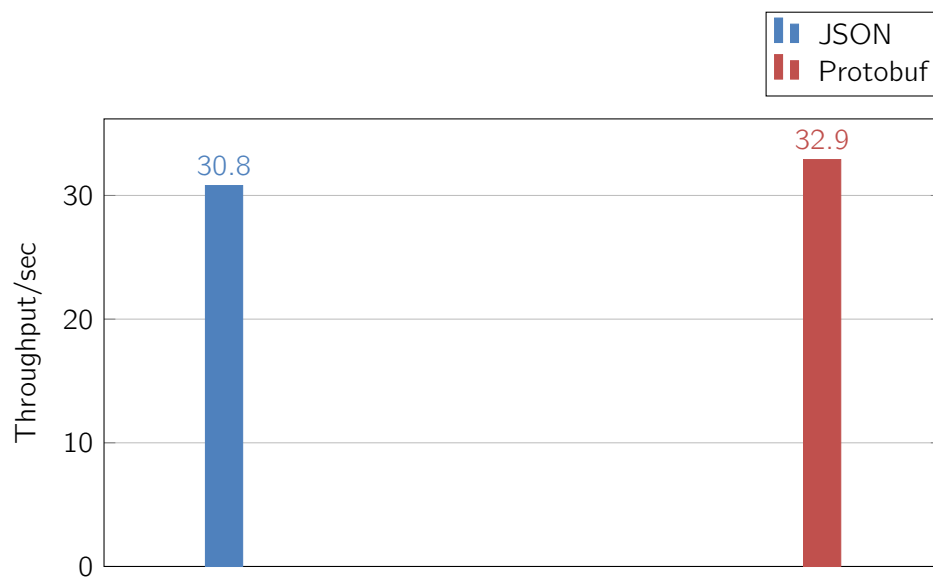


Figure 3.34: Stress GET listPetTypes Throughput

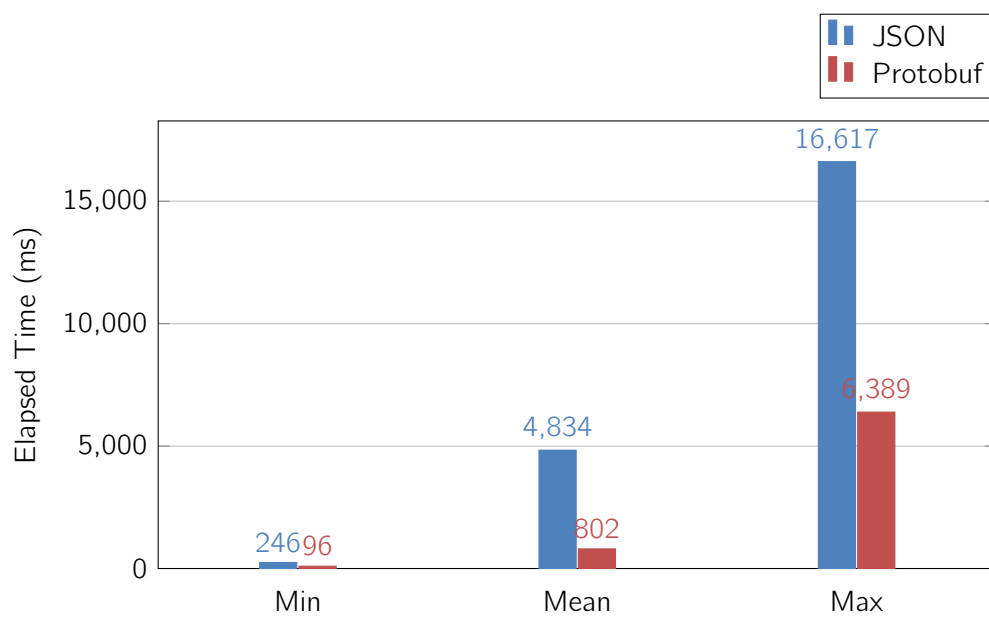


Figure 3.35: Stress GET listOwners Elapsed Time

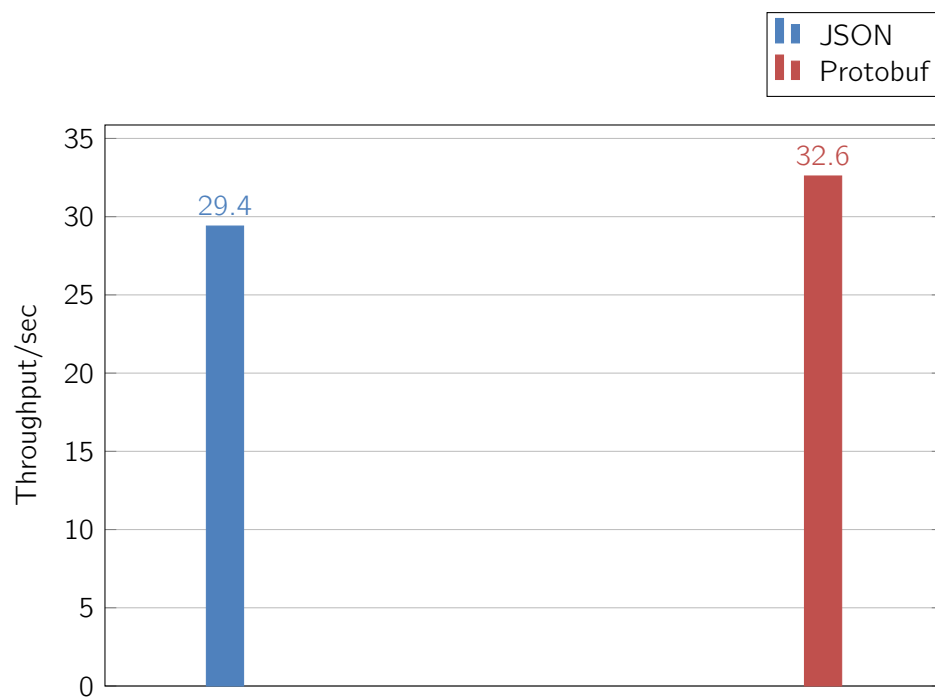
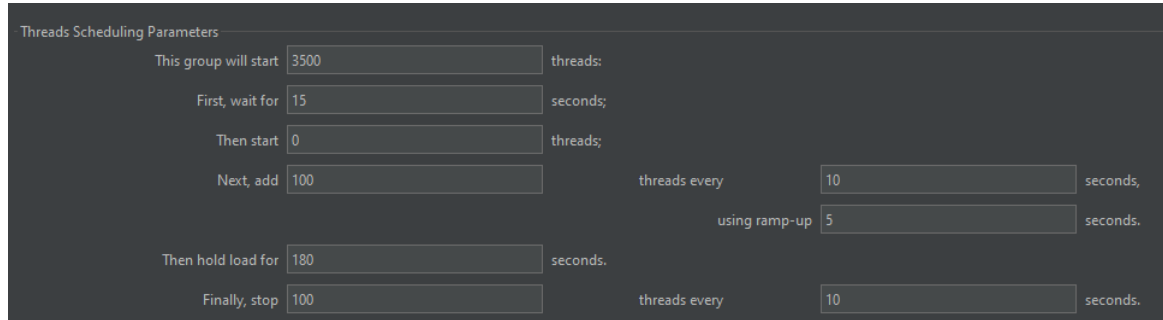


Figure 3.36: Stress GET listOwners Throughput

3.2.6 Soak Tests

Soak Tests are usually designed to work over a long period of time, but due to limitations in the development of this project, it will not be possible to perform them in that way. Instead, a JMeter plugin will be used to scale the number of threads over a certain period of time, hit a predesignated maximum number of threads and hold that number of threads for a certain period of time, then gradually decrease the number of threads until it reaches zero and the test is finished.



The screenshot shows the 'Threads Scheduling Parameters' dialog box in JMeter. It contains the following fields and values:

Parameter	Value	Unit
This group will start	3500	threads
First, wait for	15	seconds
Then start	0	threads
Next, add	100	threads every
Then hold load for	180	seconds
Finally, stop	100	threads every
using ramp-up	5	seconds

Figure 3.37: Parametrization of the Soak Tests

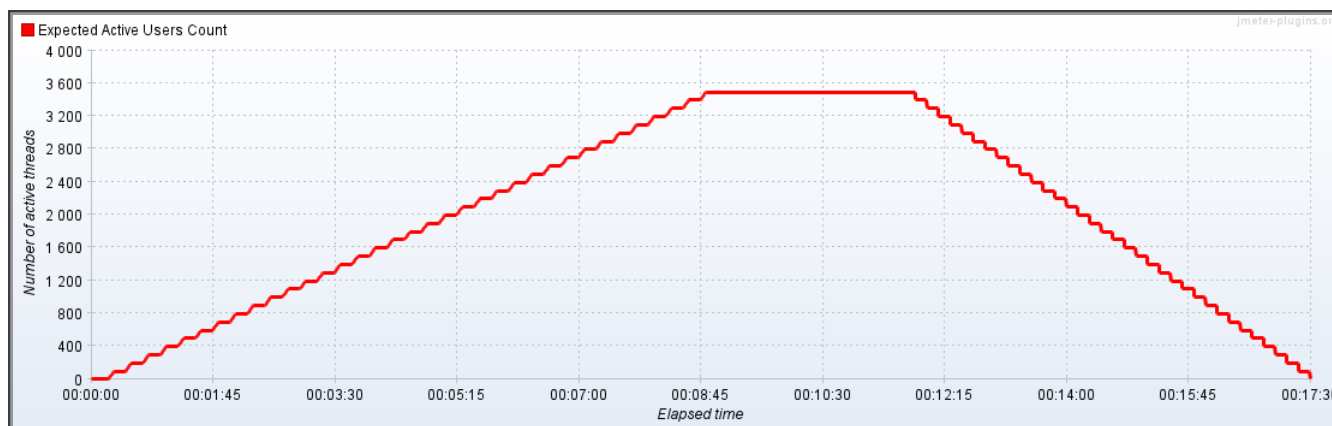


Figure 3.38: Graph representing the number of expected active users during the duration of the test

The test results (next four figures) show some differences in performance between the JSON and Protobuf projects. The Protobuf project shows slightly better performance than the JSON project, having slightly better throughput and mean and max response request times, with the JSON project having only a slightly better metric in the min response request time.

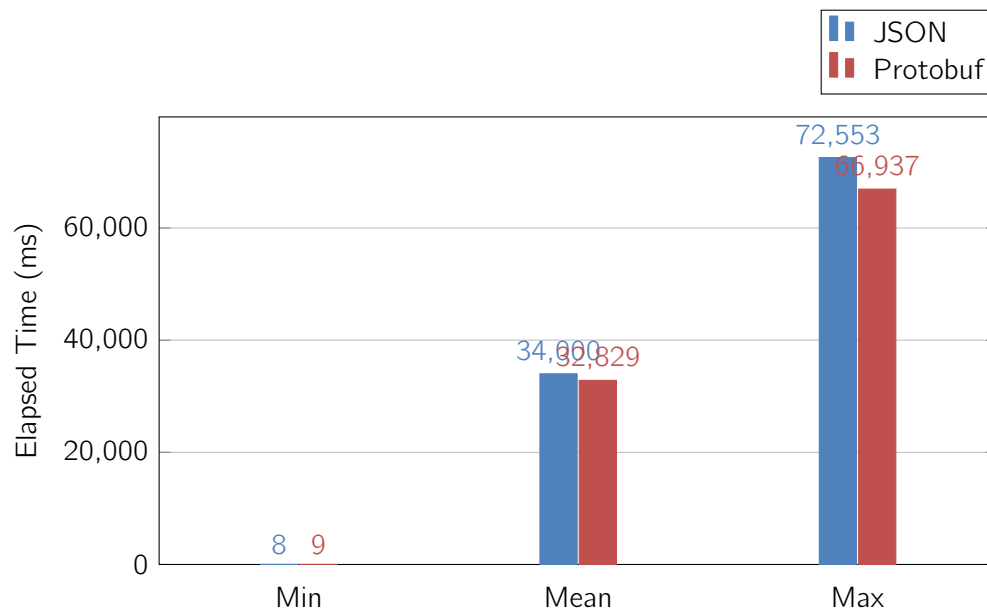


Figure 3.39: Soak GET listPetTypes Elapsed Time

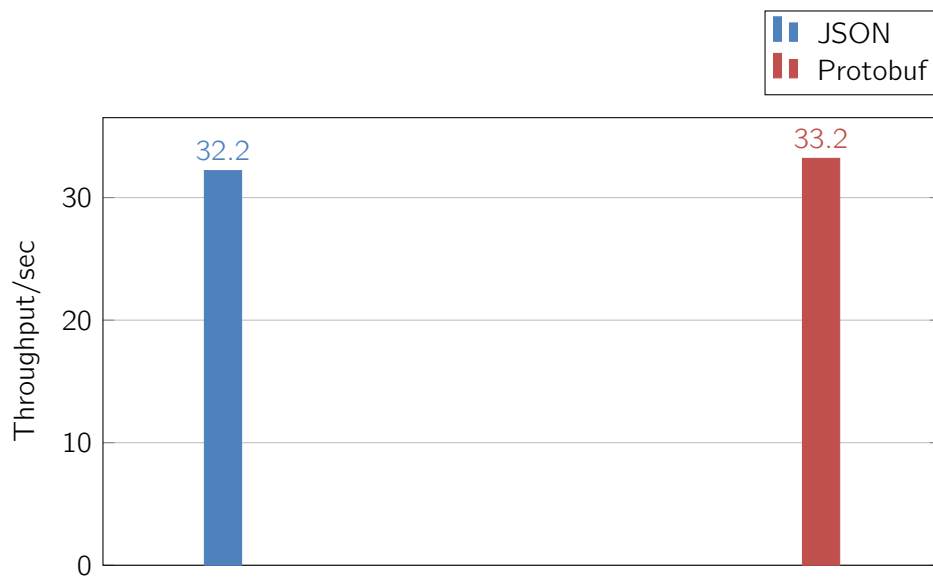


Figure 3.40: Soak GET listPetTypes Throughput

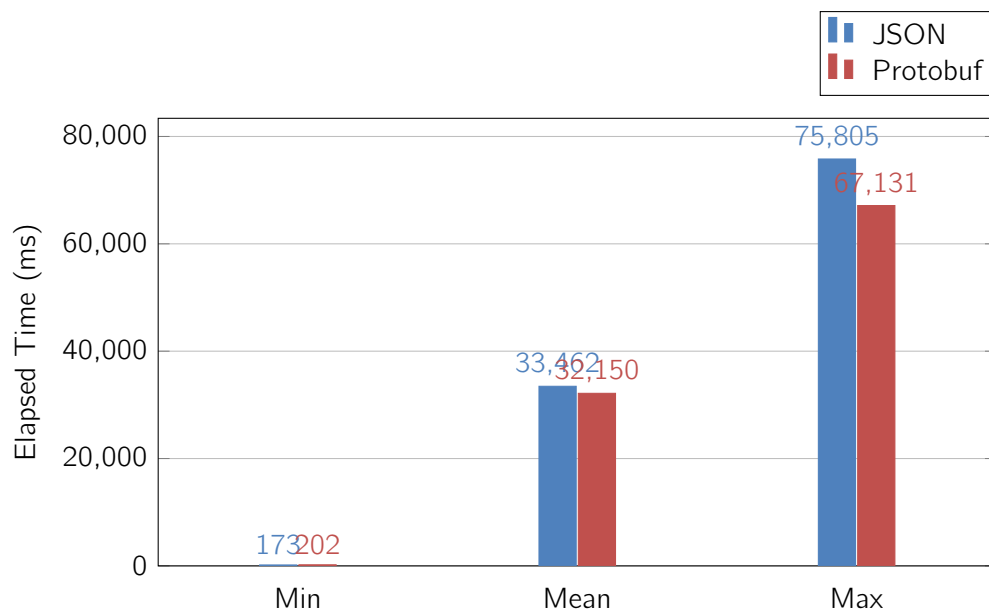


Figure 3.41: Soak GET listOwners Elapsed Time

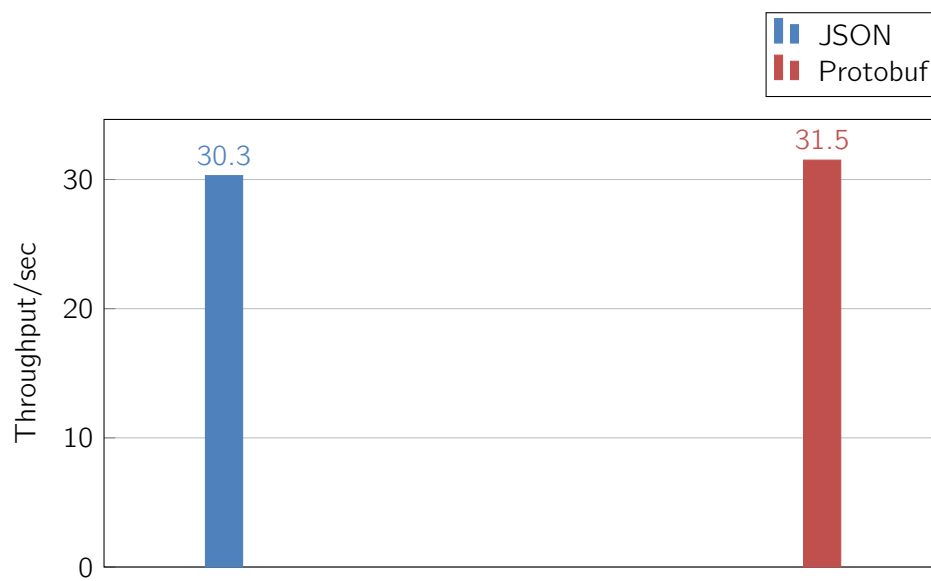


Figure 3.42: Soak GET listOwners Throughput

Chapter 4

Conclusions

4.1 Objectives Achieved

The main objective of this project was to find possible benefits in using Protobuf in REST instead of the more commonly used JSON. To accomplish that objective, a preexisting REST project using JSON was chosen and adapted to use Protobuf instead of JSON. The adaptation involved analyzing the original project to see its current structure and to know what changes were needed to adapt the project. Tools like Postman and Protoman were essential in the process of adaptation of the project, with Postman being mainly used in the original project and Protoman in the adapted project.

After the adaptation of the project was concluded, a series of different tests were developed in JMeter in order to gather data that could provide Information about potential differences in performance between the projects. An analysis of the tests' results show us that Protobuf potentially provides benefits over JSON when the applications are dealing with considerable sized loads, while presenting very similar performance metrics with JSON when the loads are small or averaged sized. This conclusion was made based in the fact that the Baseline and Load Tests presented very similar results for both projects, while the Stress and Soak tests presented better metrics when testing the Protobuf project.

4.2 Contributions

All the work developed is available in a GitHub public repository owned by the author of this report [22].

Bibliography

- [1] Khaled Sellami, Mohamed Saied, and Ali Ouni. *A Hierarchical DBSCAN Method for Extracting Microservices from Monolithic Applications | Proceedings of the 26th International Conference on Evaluation and Assessment in Software Engineering*. url: <https://dl.acm.org/doi/10.1145/3530019.3530040> (visited on 05/10/2024).
- [2] Janaina Ludwig. "Uma análise comparativa entre gRPC e REST para a integração de serviços Web". June 24, 2022. url: <http://repositorio.utfpr.edu.br:8080/jspui/handle/1/32270> (visited on 05/10/2024).
- [3] Jan Vanura and Pavel Kriz. "Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats". In: *Services Computing – SCC 2018*. International Conference on Services Computing. Springer, Cham, 2018. isbn: 978-3-319-94376-3. url: https://link.springer.com/chapter/10.1007/978-3-319-94376-3_11 (visited on 05/10/2024).
- [4] Richa Maurya. "Application of Restful APIs in IOT: A Review". In: *International Journal for Research in Applied Science and Engineering Technology* (Feb. 28, 2021), p. 2. url: <https://www.ijraset.com/files/serve.php?FID=33013> (visited on 05/09/2024).
- [5] Mark Masse. *REST API Design Rulebook*, pp. 2–5. url: https://books.google.com/books/about/REST_API_Design_Rulebook.html?hl=pt-PT&id=eABpzyTcJNIC (visited on 05/10/2024).
- [6] *apidiagram.jpeg (JPEG Image, 750 × 404 pixels)*. url: https://tethys-staging.byu.edu/static/hydroshare_python/images/apidiagram.jpeg (visited on 05/10/2024).
- [7] amazon. *What are Microservices? | AWS*. url: <https://aws.amazon.com/microservices/> (visited on 08/26/2024).
- [8] Chris Richardson. *Microservices Patterns*. Nov. 19, 2018. url: https://books.google.com/books/about/Microservices_Patterns.html?hl=pt-PT&id=QTgzEAAAQBAJ (visited on 08/26/2024).
- [9] Hemant Mishra K. *Data Serialization and Deserialization: What is it?* Nov. 30, 2023. url: <https://medium.com/@khemanta/data-serialization-and-deserialization-what-is-it-29b5ca7a756f> (visited on 08/26/2024).
- [10] arvindpdmn and anuradhac. *Data Serialization*. Feb. 27, 2019. url: <https://devopedia.org/data-serialization> (visited on 08/26/2024).
- [11] Sabine Henneberger, Matthias Schulz, and Susanne Dobratz. "Introduction to XML for ETDs". In: (2003), p. 2. url: <https://edoc.hu-berlin.de/bitstream/handle/18452/1823/index.pdf?sequence=1&isAllowed=y>.
- [12] Erik T. Ray. *Learning XML*, pp. 14–16. url: https://books.google.com/books/about/Learning_XML.html?hl=pt-PT&id=Zilck1_0c5QC (visited on 05/09/2024).
- [13] Ben Smith. *Beginning JSON*. Feb. 27, 2015, pp. 37–38. isbn: 978-1-4842-0202-9. url: https://books.google.pt/books/about/Beginning_JSON.html?id=ZYYnCGAAQBAJ&redir_esc=y.
- [14] Saurabh Zunke and Veronica D'Souza. "JSON vs XML: A Comparative Performance Analysis of Data Exchange Formats". In: (2014), p. 1. url: <https://ijcsn.org/>

- IJCSN-2014/3-4/JSON-vs-XML-A-Comparative-Performance-Analysis-of-Data-Exchange-Formats.pdf.
- [15] Chris Currier. "Protocol Buffers". In: *Mobile Forensics – The File Format Handbook*. 2022, p. 225. isbn: 978-3-030-98467-0. url: https://link.springer.com/chapter/10.1007/978-3-030-98467-0_9 (visited on 05/09/2024).
 - [16] Google. *Protocol Buffers*. url: <https://protobuf.dev/programming-guides/proto3/> (visited on 08/26/2024).
 - [17] Postman. *What is Postman? Postman API Platform*. url: <https://www.postman.com/product/what-is-postman/> (visited on 08/26/2024).
 - [18] Inchan Hwang. *GitHub spluxx/Protoman*. Aug. 4, 2024. url: <https://github.com/spluxx/Protoman> (visited on 08/26/2024).
 - [19] JMeter. *jMeter Apache JMeter - Apache JMeter™*. url: <https://jmeter.apache.org/> (visited on 08/26/2024).
 - [20] Petclinic Rest. *spring-petclinic/spring-petclinic-rest*. Aug. 26, 2024. url: <https://github.com/spring-petclinic/spring-petclinic-rest> (visited on 08/27/2024).
 - [21] *Spring Framework Petclinic sample application | PPT*. url: <https://www.slideshare.net/slideshow/spring-framework-petclinic-sample-application/71978076> (visited on 07/19/2024).
 - [22] 1150812. *1150812/spring-petclinic-rest-protobuf*. url: <https://github.com/1150812/spring-petclinic-rest-protobuf>.