Politécnico do Porto
Instituto Superior de Engenharia do Porto

Comunicação em gRPC

João Filipe Moreira Vieira

LETI
Licenciatura em Engenharia de Telecomunicações e Informática

Departamento de Engenharia Eletrotécnica
Instituto Superior de Engenharia do Porto

Julho, 2023

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Unidade Curricular de Projeto/Estágio, do 3º ano, da Licenciatura em Engenharia de Telecomunicações e Informática.

Candidato: João Filipe Moreira Vieira, Nº 1201005, 1201005@isep.ipp.pt

Orientação Técnica e Científica: Isabel Azevedo, ifp@isep.ipp.pt

Organização: GILT - Games, lnteraction and Learning Technologies

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

Julho, 2023

Agradecimentos

Agradeço aos meus pais por me proporcionarem todas as condições necessárias para que eu me pudesse dedicar inteiramente aos estudos e à professora Isabel Azevedo por todas as horas de atenção e por todas as correções durante o projeto de estágio.

Resumo

gRPC é uma *framework* de *Remote Procedure Call* (RPC) de alto desempenho desenvolvida pela Google e que permite a construção de aplicações distribuídas com o modelo cliente-servidor de forma rápida e eficiente.

Com a crescente importância da rapidez na resposta dos serviços num mundo cada vez mais conectado, nós os utilizadores procuramos respostas rápidas e precisas independentemente do momento ou dispositivo que estamos a utilizar. Se um serviço demora muito para responder, os utilizadores podem ficar frustrados e procurar outras opções mais rápidas determinando o sucesso ou insucesso de uma aplicação. Garantir a satisfação do cliente pode ser alcançado por meio de técnicas de otimização de desempenho como o uso de tecnologias de comunicação de última geração como o gRPC.

Este documento explora o gRPC bem como conceitos associados à *framework* e que estão relacionados com o seu funcionamento no contexto dos microsserviços, a utilização de *Protocol Buffers* como formato de serialização e uma comparação com a abordagem REST. Para além disso, são desenvolvidos dois protótipos utilizando gRPC e REST, com múltiplos serviços com o principal objetivo de estudar gRPC como meio de comunicação entre serviços. Ambos os protótipos são submetidos a uma série de testes para garantir o seu correto funcionamento.

No final é feita uma avaliação ao desempenho dos protótipos desenvolvidos e a respetiva comparação entre ambos de onde é retirada a conclusão de que o tempo médio de resposta em gRPC é significativamente mais baixo do que em REST.

Palavras-Chave: gRPC, Protocol Buffers, REST, Comunicação, Microsserviços

Abstract

gRPC is a high performance Remote Procedure Call (RPC) framework developed by Google that allows you to build distributed applications using the client-server model quickly and efficiently.

With the growing importance of fast service response in an increasingly connected world, we the users are looking for fast and accurate responses regardless of the time or device we are using. If a service takes too long to respond, users can become frustrated and look for other, faster options determining the success or failure of an application. Ensuring customer satisfaction can be achieved through performance optimization techniques such as using state-of-the-art communication technologies like gRPC.

This document explores gRPC, as well as associated concepts related to the framework's operation, such as microservices, the use of Protocol Buffers as a serialization format, and a comparison with the REST approach. Furthermore, two prototypes are developed using gRPC and REST, incorporating multiple services, with the primary objective of studying gRPC as a means of communication between services. Both prototypes are subjected to a series of tests to ensure their proper functioning.

In the end, an evaluation of the performance of the developed prototypes is conducted, along with a comparison between the two, leading to the conclusion that the average response time in gRPC is significantly lower than in REST.

Keywords: gRPC, Protocol Buffers, REST, Communication, Microservices

Índice

Lista de Figuras

Lista de Tabelas

Listagens

Lista de Abreviaturas

API	*Application Programming Interface*
DEE	Departamento de Engenharia Electrotécnica
gRPC	*gRPC Remote Procedure Call*
HTML	*HyperText Markup Language*
HTTP	*Hypertext Transfer Protocol*
IDL	Interface Definition Language
IoT	Internet of Things
ISEP	Instituto Superior de Engenharia do Porto
JSON	*JavaScript Object Notation*
LETI	Licenciatura em Engenharia de Telecomunicações e Informática
REST	*REpresentational State Transfer*
RPC	*Remote Procedure Call*
TCP	*Transmission Control Protocol*
URI	*Uniform Resource Identifier*
XML	*Extensible Markup Language*

Capítulo 1

Introdução

Neste capítulo, é apresentado o projeto realizado durante o semestre para a unidade curricular Projeto/Estágio (PESTA), do 3.º ano da Licenciatura em Engenharia de Telecomunicações e Informática (LETI), do Departamento de Engenharia Electrotécnica (DEE), do Instituto Superior de Engenharia do Porto (ISEP). O projeto tem o seu foco na exploração e estudo de gRPC Remote Procedure Call (gRPC) como estrutura de comunicação entre serviços de uma aplicação.

1.1 Contextualização

O gRPC, é uma "*framework de Remote Procedure Call* (RPC) de alto desempenho, open source, que funciona em qualquer ambiente" [1] desenvolvida pela Google e que permite a construção de aplicações distribuídas com o modelo cliente-servidor de forma rápida e eficiente.

Com a crescente importância da rapidez na resposta dos serviços num mundo cada vez mais conectado, nós os utilizadores, procuramos respostas rápidas e precisas independentemente do momento ou dispositivo que estamos a utilizar. Se um serviço demora muito para responder, os utilizadores podem ficar frustrados e procurar outras opções mais rápidas, determinando o sucesso ou insucesso de uma aplicação. Garantir a satisfação do cliente pode ser alcançado por meio de técnicas de otimização de desempenho como o uso de tecnologias de comunicação de última geração como o gRPC.

O gRPC é amplamente utilizado em sistemas distribuídos modernos em vários setores e, algumas das empresas que utilizam gRPC incluem Square, Netflix, CoreOS, Cockroach Labs, Cisco e Juniper Networks. Para além disso, o gRPC está rapidamente a ganhar popularidade entre os desenvolvedores devido às suas características nomeadamente a performance.

1.2 Descrição do Projeto

Apesar de ser amplamente utilizado em vários setores e de estar a ganhar popularidade entre os desenvolvedores, ainda há muitos que desconhecem o gRPC e os seus recursos e ainda estão mais familiarizados com o REST. Isso pode ser devido à falta de divulgação mesmo nos estabelecimentos de ensino como universidades ou ao fato de que o gRPC é uma tecnologia relativamente recente (publicada pela Google em 2015). É importante destacar que o gRPC e o REST são duas tecnologias diferentes e é fundamental entender as diferenças entre ambas as tecnologias já que ambas têm as suas vantagens e desvantagens para o desenvolvimento de uma aplicação.

1.2.1 Objetivos

O principal objetivo deste trabalho é explorar o gRPC como uma estrutura de comunicação entre serviços para o desenvolvimento de aplicações distribuídas modernas. Outro objetivo é compreender e conhecer as funcionalidades do gRPC, as suas vantagens e desvantagens, e comparar o gRPC a outras estruturas de comunicação nomeadamente REST para o desenvolvimento de aplicações distribuídas eficientes e escaláveis. Para esse efeito, será apresentada uma implementação recorrendo a REST e outra implementação recorrendo a gRPC para a comunicação entre serviços com a intenção de responder a um determinado pedido feito através de uma *Application Programming Interface* (API) RESTful externa e, para além disso, fazer a comparação do desempenho dos 2 protótipos utilizando JMeter.

Para a exploração de gRPC, será desenvolvida uma aplicação com múltiplos serviços para suprir as necessidades de uma loja de sanduíches com funcionalidades relacionadas com o negócio, como, por exemplo, criação de sanduíches, listagem dos ingredientes disponíveis, realizar uma reserva de sanduíche para levantamento em loja, entre outras, com algum suporte multilíngua. Será adotada uma arquitetura baseada em microsserviços recorrendo também a eventos para comunicação entre serviços. Todo o trabalho ficará disponível num repositório público como contributo para divulgação de gRPC e da sua utilização. Na implementação da solução existem algumas restrições: o gRPC deve ser utilizado como meio de comunicação entre os microsserviços. Numa outra implementação será utilizado REST em vez de gRPC para efetuar uma comparação.

1.3 Calendarização

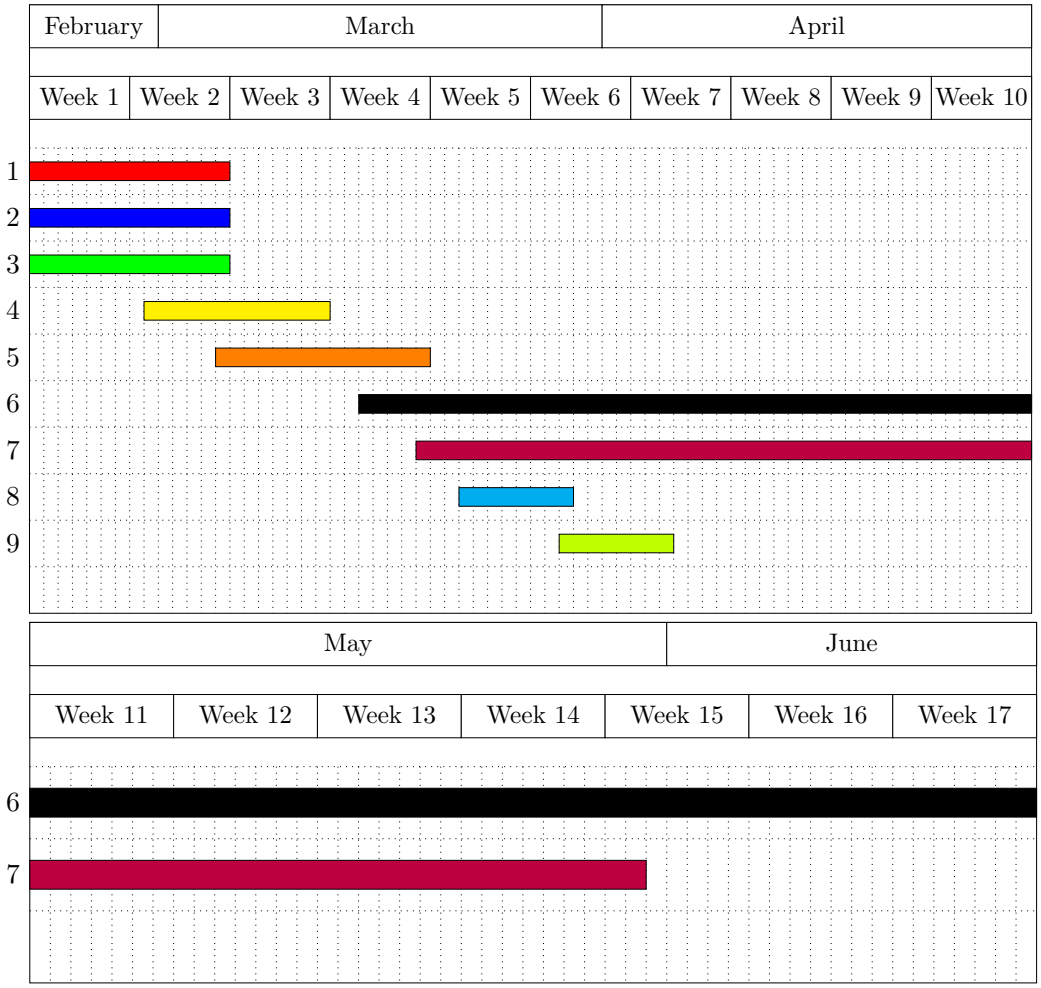


Figura 1.1: Calendarização

1. Análise da documentação de gRPC;
2. Desenvolvimento de aplicação base em REST;
3. Desenvolvimento de testes de integração;
4. Exploração de gRPC (desenvolvimento de aplicação-exemplo de Hello world);
5. Desenvolvimento de aplicação base em gRPC;
6. Escrita de relatório;
7. Adição de novas funcionalidades e melhoria das aplicações num desenvolvimento contínuo;
8. Exploração de jmeter;
9. Realização de testes jmeter;

1.4 Organização do Relatório

O documento é composto por 6 capítulos. No primeiro, é feita uma introdução e contextualização do projeto bem como os objetivos propostos e a calendarização do plano de trabalhos. No segundo, é feito um aprofundamento e exploração da tecnologia principal e outros conceitos relevantes e relacionados com a mesma. No terceiro, é apresentado um cenário em que o projeto é baseado e é feita uma análise do problema. No quarto é realizada a conceção da solução, apresentando vários diagramas referente à arquitetura da mesma. No quinto capítulo, é demonstrada a implementação da solução, vários extratos de código e os vários testes relativos à mesma. Finalmente, no Capítulo 6 são apresentadas as conclusões do trabalho, algumas das dificuldades durante o projeto, algumas sugestões para um trabalho futuro e apreciações finais.

Capítulo 2

Estado de Arte

O presente capítulo tem como finalidade explorar os principais conceitos associados ao problema apresentado previamente. Para isso, o capítulo explora em que consiste uma arquitetura baseada em microsserviços bem como os formatos de serialização de dados, um aspeto crítico para aplicações modernas. De seguida é analisado o gRPC, sendo o objeto de estudo em questão, assim como as suas vantagens e desvantagens, e alguns protocolos de comunicação relacionados e frequentemente comparados ao mesmo.

2.1 Serialização de Dados Estruturados

A serialização de dados estruturados é um processo muito importante para a troca de informações entre sistemas distribuídos. Diversos formatos são utilizados para representar, transportar e armazenar dados de forma organizada. Nesta secção, alguns desses formatos comuns: XML, JSON, Protocol Buffers, FlatBuffers e Thrift serão explorados.

2.1.1 *Extensible Markup Language* (XML)

XML é uma linguagem de anotação muito utilizada para representar dados estruturados. Quando se diz que é uma linguagem de marcação, significa que ele utiliza etiquetas ou *tags* para delimitar dados [2] e tanto as *tags* como a estrutura são definidas pelo autor, uma vez que não existem *tags* predefinidas [3]. O XML é facilmente lido por humanos e permite representar uma ampla variedade de estruturas de dados.

No entanto, o seu formato é baseado em texto e isso pode torná-lo mais extenso e menos eficiente em termos do tamanho de arquivo e da velocidade de processamento em comparação com outros formatos.

Nas linhas de código presentes na Listagem 2.1 está um exemplo simples de utilização de XML

```
<book>
  <title>A Christmas Carol</title>
  <author>Charles Dickens</author>
  <year>1843</year>
</book>
```

Listagem 2.1: Exemplo simples de dados estruturados em XML.

2.1.2 *JavaScript Object Notation* (JSON)

JSON é um formato leve e frequentemente utilizado para troca de dados. É baseado em Javascript e é fácil de ler e escrever para humanos e oferece vários tipos de dados como *string*, numérico, *boolean*, entre outros [4]. Devido à estrutura simplificada e mais concisa tornam-no, geralmente, mais eficiente tanto em tamanho de arquivo quanto em processamento do que XML. No entanto, devido ao facto de ser apenas texto também o pode tornar num formato mais extenso e menos eficiente comparativamente a outros formatos.

Nas linhas de código presentes na Listagem 2.2, está um exemplo simples de utilização de JSON.

```
{
  "title": "A Christmas Carol",
  "author": "Charles Dickens",
  "year": 1843
}
```

Listagem 2.2: Exemplo simples de dados estruturados em JSON.

2.1.3 *Protocol Buffers*

Ao longo deste documento, o *Protocol Buffers* já foi referido várias vezes, e sendo que ele é uma característica muito importante do gRPC é relevante explorar o tópico de *Protocol Buffers*.

O *Protocol Buffers*(*Protobuf*), também desenvolvido pela Google, constitui um mecanismo extensível, neutro em termos de linguagem e plataforma, para a serialização de dados estruturados. O *Protocol Buffers* é binário, o que significa que os dados são armazenados num formato compacto e eficiente em termos de espaço.

Para utilizar o *Protocol Buffers* é necessário definir a estrutura dos dados que pretendemos utilizar num ficheiro ".proto" e a partir daí, utilizando um compilador proto, vamos usar esse ficheiro para gerar código equivalente, mas na linguagem pretendida. O *Protocol Buffers* atualmente suporta diversas linguagens de programação como Java, Python, C++, etc. Para compreender melhor a definição de um serviço em *Protocol Buffers*, a Listagem 2.3 apresenta um exemplo simples de um "HelloService".

```
1  syntax = "proto3";
2  package com.hello;
3
4  message HelloRequest {
5    string greeting = 1;
6  }
7
8  message HelloResponse {
9    string reply = 1;
10 }
11
12 service HelloService {
13   rpc SayHello (HelloRequest) returns (HelloResponse);
14   rpc LotsOfReplies(HelloRequest) returns (stream HelloResponse);
15   rpc LotsOfGreetings(stream HelloRequest) returns (HelloResponse)
         ;
16   rpc LotsOfEverything(stream HelloRequest) returns (stream
         HelloResponse);
17 }
```

Listagem 2.3: Exemplo simples de definição de serviço.

Inicialmente no exemplo, é definida a versão de proto (proto3, outras versões são, até à data, fortemente desencorajadas) e o *package* onde os ficheiros com o código gerado serão encontrados.

As *message* são usadas para definir aquilo que é trocado entre o cliente e o servidor e geralmente são bem distinguidas entre um pedido(neste caso, *HelloRequest*) e uma resposta (neste caso, *HelloResponse*).
Já os *service* são usados para definir os métodos que um cliente pode chamar remotamente a um servidor. É possível reparar que no exemplo estão a ser usados os 4 tipos de gRPC (*Unary RPC, Server streaming RPC, Client Streaming RPC* e *Bidirectional streaming RPC*) que irão ser abordados mais adiante no documento.

Ao utilizar um *plugin* para gerar o código na linguagem desejada, estes métodos ficam prontos a ser utilizados e modificados pelo cliente e servidor.

2.1.4 *FlatBuffers*

FlatBuffers é uma biblioteca de serialização de dados de alto desempenho. Também desenvolvido pelo Google, foi originalmente criado para o desenvolvimento de jogos e outras aplicações em que o desempenho é crítico [5]. Os *Protocol Buffers* e os *FlatBuffers* são de facto bastante semelhantes e a principal diferença entre ambos é que nos *FlatBuffers* não é necessário desserializar todos os dados para uma representação secundária (onde só se quer parte dos mesmos) antes de se poder aceder aos mesmos. Isto faz com que os *FlatBuffers* sejam responsáveis por casos de utilização em que existem muitos dados e dados muito grandes [6]. Para além disso, os *FlatBuffers* foram concebidos para não serem modificados após a sua criação o que faz com que eles também sejam aconselhados para cenários de utilização com muitas operações de leitura e acessos apenas de leitura [7]. Ele também consome muito menos memória nas suas operações do que *Protocol Buffers*.

Nas linhas de código presentes na Listagem 2.4, está um exemplo simples de utilização de *FlatBuffers*.

```
1  namespace MyNamespace;
2
3  table HelloResponse {
4    message: string;
5  }
6
7  table HelloRequest {
8    name: string;
9  }
10
11 service HelloService {
12   rpc SayHello(HelloRequest) returns (HelloResponse);
13 }
```

Listagem 2.4: Exemplo simples de definição de serviço em FlatBuffers.

Para além das diferenças visiveis no código, enquanto que em *Protocol Buffers* a definição do serviço é feita num ficheiro ".proto", em *FlatBuffers* a definição do serviço é feita num ficheiro ".fbs".

2.1.5 *Thrift*

Thrift é uma *framework* desenvolvida pelo *Apache Software Foundation* que permite a definição de estruturas de dados e serviços para comunicação entre diferentes sistemas. Ele possui a sua própria Interface Definition Language (IDL) (linguagem de definição de interface) para descrever as estruturas de dados e as operações disponíveis. O *Thrift* oferece suporte a vários protocolos de serialização, como o formato binário compacto, JSON e XML. Tal como os *Protocol Buffers* e os *FlatBuffers*, Thrift também gera automaticamente o código-fonte necessário em várias linguagens de programação para facilitar a interoperabilidade entre diferentes sistemas [8].

Nas linhas de código presentes na Listagem 2.5, está um exemplo simples de utilização de *Thrift* [9].

```
1  namespace java com.example.hello
2
3  struct HelloResponse {
4    1: string message;
5  }
6
7  struct HelloRequest {
8    1: string name;
9  }
10
11 service HelloService {
12   HelloResponse sayHello(1: HelloRequest request);
13 }
```

Listagem 2.5: Exemplo simples de definição de serviço em Thrift.

O ficheiro utilizado possui a terminologia ".thrift".

2.2 Microsserviços e Protocolos

Nesta secção é apresentada uma análise sobre microsserviços e protocolos de comunicação. No âmbito de entender bem o gRPC, as suas vantagens, desvantagens e ter uma melhor perceção de quando se deve utilizar gRPC, é fulcral entender bem o que é uma arquitetura baseada em microsserviços, bem como alguns dos protocolos com que ele se relaciona.

2.2.1 Arquitetura Baseada em Microsserviços

A arquitetura baseada em microsserviços consiste num conjunto de serviços pequenos e autónomos e cada um responsável por apenas uma funcionalidade do negócio

e com a sua própria interface podendo cada serviço ser escrito em linguagens de programação diferentes e serem geridos por equipas diferentes [10]. A sua popularidade tem vindo a crescer ao longo dos anos devido às suas características e benefícios, nomeadamente:

- Escalabilidade: os serviços podem ser escalados independentemente, ou seja, se um serviço precisar de mais recursos este pode crescer e se adaptar sem que toda a aplicação tenha de crescer também
- Flexibilidade: os serviços podem ser implantados em diferentes dispositivos e em caso de falha de um dispositivo ou da sua rede, a aplicação pode continuar funcional
- Agilidade: caso ocorra um erro ou um bug é fácil de localizar e corrigir os bugs. Para além disso ao corrigir um bug ou ao ser precisa uma atualização só é necessário alterar o serviço e não toda a aplicação e é fácil aderir a novas tecnologias que seguem o mercado
- Elasticidade: com as variações do trabalho necessário um serviço pode aumentar ou diminuir dinamicamente a sua capacidade de atender aos pedidos
- Código base pequeno: cada serviço tem uma base de código pequena e privada o que facilita a sua gerência
- Disponibilidade e tolerância a falhas: caso um serviço falhe, a aplicação poderá continuar a funcionar já que cada serviço é independente o que fará com que a aplicação esteja disponível durante mais tempo
- Segurança e dados isolados: como o código não é partilhado e replicado entre os serviços, estes ficam menos vulneráveis a ataques e para além disso é mais fácil atualizar e alterar o "*schema*" ou os *models* já que apenas um serviço é afetado por essa alteração

Apesar de todos estes benefícios, a arquitetura baseada em microsserviços também tem dificuldades e desafios nomeadamente:

- Gestão e compatibilidade: se um serviço for alterado/atualizado, este pode perder a sua compatibilidade com outro serviço. Para além disso, como cada serviço pode estar em linguagens de programação diferentes, pode ser difícil fazer a sua gestão
- Complexidade: cada serviço é simples, no entanto, todos os serviços juntos, ou seja, a aplicação como um todo é um sistema muito complexo [11]
- Conectividade: os serviços precisam de comunicar uns com os outros enviando e recebendo dados, com o objetivo de responder a um determinado pedido do

cliente. Esta comunicação é um grande desafio na arquitetura de microsserviços, já que esta condiciona o tempo de resposta ao cliente e a pode tornar demasiado demorada.

2.2.2 HTTP

Hypertext Transfer Protocol (HTTP) é o protocolo padrão para as comunicações entre servidores web e clientes web, mas também pode ser utilizado noutros cenários. O HTTP segue o modelo clássico cliente-servidor em que o cliente abre uma conexão fazendo um pedido (HTTP Request) e espera até que o servidor responda (HTTP Response). O HTTP é um protocolo "*stateless*", o que significa que o servidor não guarda quaisquer dados entre dois pedidos seguidos [12].

Os termos "cliente" e "servidor" referem-se apenas aos papéis que estes programas desempenham para uma determinada ligação. O mesmo programa pode atuar como cliente em algumas ligações e como servidor em outras [13].

Um HTTP Request corresponde a um cabeçalho (*header*) e a um corpo (*body*) que é opcional. O *header* contém informações sobre o tipo de operação a realizar (por exemplo: *GET, POST, PUT, PATCH, DELETE*, etc.), o *Uniform Resource Identifier* (URI), as informações do cliente, *cookies* entre outros. O *body* possui os dados que o cliente quer enviar, geralmente em operações *POST, PUT* e *PATCH*.

Todas as operações têm funções diferentes [12] e dentro das ditas temos que:

- *GET*: obtém a representação de um recurso específico. Retorna informação de acordo com o nosso pedido
- *POST*: utilizado para criar um novo recurso
- *PUT*: faz uma modificação total de um recurso específico
- *PATCH*: semelhante ao PUT, mas faz apenas uma modificação parcial de um recurso específico
- *DELETE*: Elimina um recurso específico

Uma HTTP *Response* também corresponde a um cabeçalho e a um corpo. O cabeçalho contém informações sobre o status da operação, o tipo de conteúdo (JSON, HTML, etc.), *cookies*, etc. O corpo contém o conteúdo da resposta como, por exemplo, uma página HTML.

Dentro dos vários códigos de status de uma operação estes são os mais comuns:

- 200: *OK*
- 201: *Created*
- 400: *Bad Request*
- 403: *Forbidden*
- 404: *Not Found*
- 409: *Conflict*
- 500: *Internal Server Error*

Apesar de o HTTP ser *stateless*, as *cookies* permitem estabelecer sessões *stateful*, ou seja, permitem que haja informação mantida entre 2 pedidos HTTP [12].

HTTP/2

O HTTP é um protocolo extremamente bem-sucedido, no entanto, têm algumas características que afetam negativamente o desempenho das aplicações.

O HTTP/1.0 permitia que apenas um pedido fosse feito e houvesse uma resposta numa determinada conexão TCP. Em HTTP/1.1, foi adicionado o recurso "HTTP *request pipeline*" que, basicamente, permite que vários pedidos sejam feitos sem que nenhum tenha sido respondido e apesar de já ter parcialmente abordado o problema da concorrência de pedidos este ainda sofre de "*head-of-line blocking*", onde um pedido bloqueado impede o processamento de outros pedidos na fila. Por causa disso, os clientes que precisam de fazer muitos pedidos fazem várias conexões TCP para poderem enviar vários pedidos ao servidor ao mesmo tempo, e receber as respostas correspondentes de forma independente e assim reduzir a latência [14].

O HTTP/2 permite uma utilização mais eficiente dos recursos da rede. Esta versão introduz compressão de cabeçalhos (*header compression*), multiplexagem, codificação eficiente dos *headers* e também permite a priorização de pedidos, permitindo que pedidos mais importantes sejam concluídos mais rapidamente, melhorando ainda mais o desempenho [15]. Desde já, é importante destacar que o HTTP/2 suporta todos os principais conteúdos das versões anteriores, mas tem o objetivo de ser mais rápido e eficiente.

Antes de uma explicação das novas funcionalidades, é necessário conhecer os seguintes conceitos:

- *stream*: fluxo bidirecional de *bytes* dentro de uma ligação. Uma *stream* é capaz de transportar uma ou mais mensagens

- *message*: uma sequência de tramas que representam um pedido HTTP ou uma resposta HTTP

- *frame*: a menor unidade de comunicação em HTTP/2, cada uma contendo um cabeçalho, que no mínimo identifica a *stream* ao qual a trama pertence

Em HTTP/2 os dados são transmitidos num formato binário, o que permite que eles sejam mais facilmente processados por máquinas e otimiza a transmissão.

Assim sendo, uma das principais características e funcionalidades do HTTP/2 é o *Binary Framing Layer* (Figura 2.1) que define como as mensagens HTTP são encapsuladas e transferidas entre o cliente e o servidor [16].

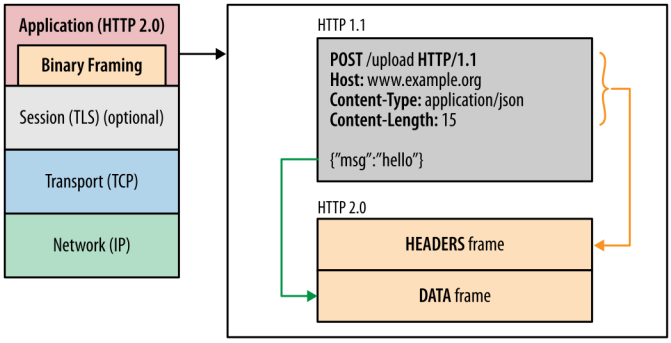


Figura 2.1: *Binary Framing Layer* [16]

Relacionando com os conceitos anteriores, há troca de *frames* em formato binário, que são depois mapeados para mensagens que pertencem a uma determinada *stream*, todas as quais são multiplexados dentro de uma única ligação TCP. Esta é a base que permite todas as outras funcionalidades fornecidas pelo HTTP/2. Outras funcionalidades entre as quais:

- *Multiplexing*: umas das principais diferenças entre o HTTP/2 e as versões anteriores. Várias solicitações e respostas podem ser enviadas e recebidas simultaneamente, em vez de esperar por uma única resposta de cada vez. Cada pedido ou resposta está associado à sua própria *stream*. Estas são independentes e caso ocorra a falha numa isto não impede outra de avançar

- Priorização de *streams*: assegura que os recursos podem ser utilizados primeiro para as *streams* mais importantes

- *Server push*: o servidor pode "empurrar" dados para um cliente se ele antecipar que o cliente irá necessitar deles trocando alguma utilização da rede contra um potencial ganho de latência

Na Figura 2.2, é visível a diferença da comunicação entre o cliente e o servidor:

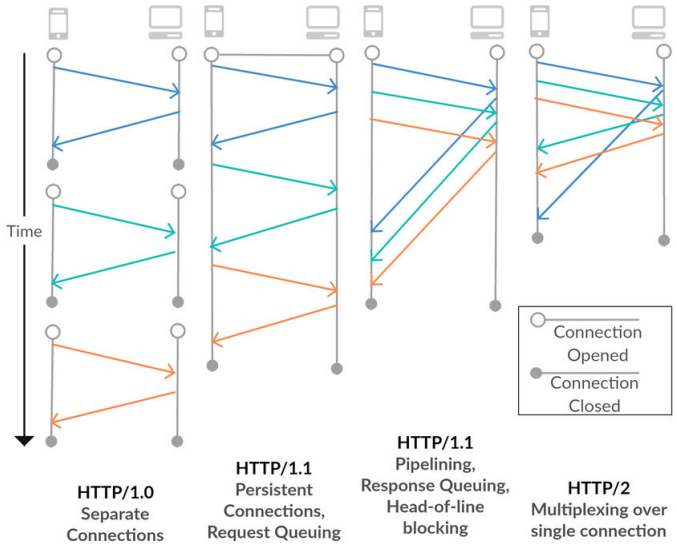


Figura 2.2: Comparação do envio de pedidos e respostas nas diferentes versões HTTP [17]

2.2.3 RPC

Remote Procedure Call (RPC) é um protocolo de comunicação que, como se pode observar na figura 2.3, segue o padrão cliente-servidor em que uma aplicação (cliente) faz um pedido a um serviço de outra aplicação (servidor) localizado noutro dispositivo sem que haja a necessidade de compreender os detalhes da rede, ou seja, o RPC permite que se chame a função de um processo remoto como se fosse uma função local [18].

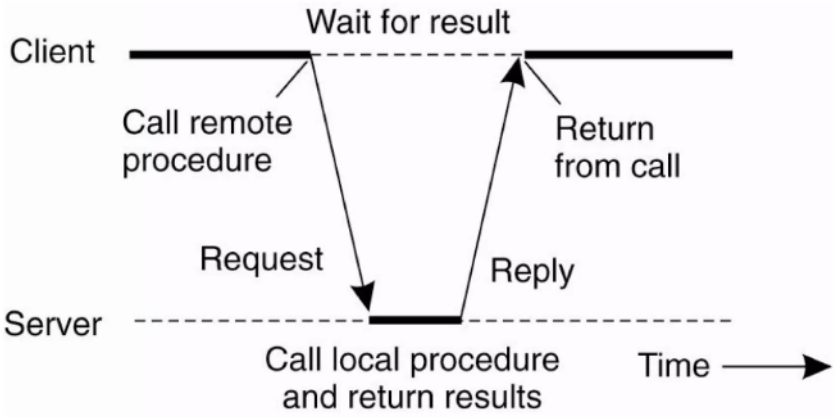


Figura 2.3: *Remote Procedure Call* [19]

2.3 Descrição de REST

REpresentational State Transfer (REST) é um estilo arquitetural de *software* que foi criado com o objetivo de criar um princípio orientador para a definição das interfaces dos vários componentes em sistemas distribuídos. REST é extremamente popular e isso deve-se em parte à sua base em HTTP.

O REST é baseado em alguns princípios:

- Adota o modelo cliente-servidor: reforça a separação de responsabilidades e preocupações o que ajuda tanto o cliente como o servidor a evoluir independentemente. O cliente utiliza URIs para obter recursos, sem qualquer preocupação de como é que o servidor processa o pedido. Por outro lado, o servidor processa o pedido e retorna um recurso, sem qualquer preocupação com a forma como o cliente vai utilizar esse recurso. Tanto o cliente como o servidor não precisam de se preocupar com as responsabilidades um do outro [20]

- Interface uniforme: as operações entre o cliente, servidor e possíveis intermediários devem ser definidas de forma clara e consistente. Para além disso, este principio é sub-dividido em 4 restrições: todos os recursos do sistema devem ser identificados por um identificador único (URI), manipulação de recursos através das suas representações como JSON, mensagens auto-descritivas e HATEOAS (*Hypermedia as the Engine of Application State*) [21]

- Comunicação *stateless*: uma API deve ser "*stateless*", ou seja, não deve armazenar qualquer informação sobre a sessão do utilizador. Por consequência, cada pedido deve fornecer dados completos para o seu processamento [22].

- *Cacheable*: cada resposta por parte do servidor deve ser definida como "*Cacheable*" ou "*Non-Cacheable*". se for "*Cacheable*" o cliente pode reutilizar os dados da resposta para pedidos semelhantes futuros e durante um período de tempo específico [21], caso contrário o cliente tem de mandar um novo pedido ao servidor

- Sistema em camadas: ocorre uma separação da API em múltiplas camadas, estas só devem utilizar as camadas abaixo e comunicar o resultado com as superiores. O cliente não deve saber a quantidade de camadas existentes logo podem-se adicionar, alterar ou remover com o objetivo de melhorar a escalabilidade [20]

- *Code on demand*(Opcional): o cliente pode fazer *download* do código em vez de receber os dados (em JSON, por exemplo) [20]

Para além disso, os recursos podem ser manipulados através das operações básicas de HTTP (*GET, POST, PUT, PATCH, DELETE*, entre outros). Na resposta

do servidor ao cliente, este também envia o resultado do processamento do pedido do cliente utilizado os vários códigos de status de HTTP (200, 400, 404, 500, entre outros).

2.4 gRPC

Como já foi dito anteriormente, o gRPC é uma *framework* de RPC de alto desempenho e *open source* que foi desenvolvida pela Google. O gRPC permite que microsserviços comuniquem entre si remotamente de forma rápida e eficiente.

A Google utilizava uma *framework* de uso geral chamada "*Stubby*" para conectar uma grande quantidade de microsserviços. Em 2015, a Google decidiu criar a próxima versão de "*Stubby*", tirar partido de novas tecnologias e alargar o seu uso a uso móvel, Internet of Things (IoT) e web [23].

2.4.1 Descrição e Características do gRPC

O gRPC é considerado uma evolução de RPC e é conhecido principalmente pela sua melhoria na performance, desempenho, rapidez e eficiência. As suas características são a base de tudo isso:

- HTTP/2: O gRPC utiliza o HTTP/2 como protocolo de transporte para o envio de mensagens. Sendo esta uma das razões para a alta performance dele
- *Protocol Buffers*: o gRPC utiliza *Protocol Buffers* como Interface Definition Language (IDL) para definir a interface de serviço e a estrutura das mensagens. o gRPC utiliza os *Protocol Buffers* como mecanismo para fazer a serialização dos dados o que aumenta a eficiência
- Suporte de várias linguagens de programação: o gRPC suporta várias linguagens de programação e não é necessário que o cliente e o servidor estejam na mesma linguagem

Padrões de Comunicação

Existem 4 tipos de comunicação baseada em gRPC, nomeadamente:

- *Unary* RPC (simples): O mais simples de todos, o cliente envia um pedido e recebe uma resposta a esse pedido [24], como se pode ver na figura 2.4. Um exemplo prático para este tipo pode ser um sistema de *login*

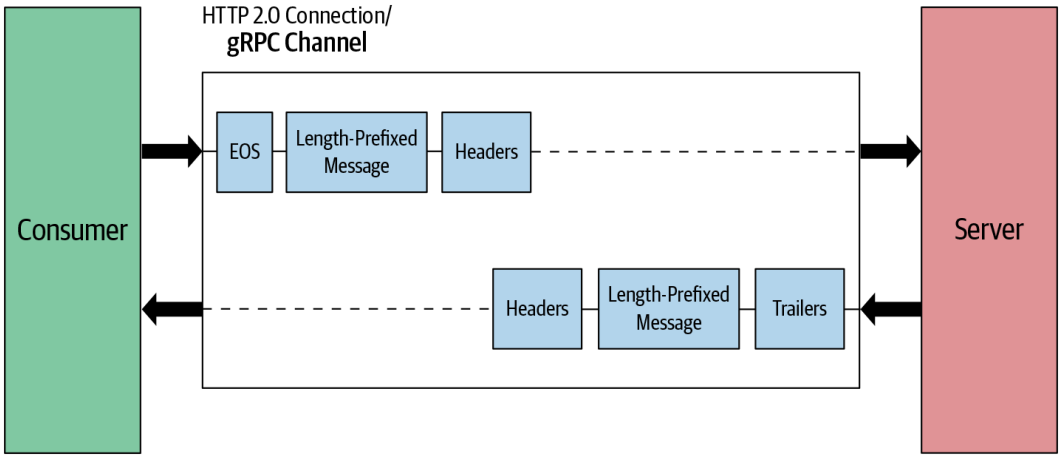


Figura 2.4: RPC Simples [25]

- *Server streaming* RPC: o cliente envia um único pedido ao servidor, mas recebe várias respostas a esse pedido [24], como se pode ver na figura 2.5. Neste caso, um exemplo pode ser a visualização de um episódio na Netflix, o cliente pede para ver um episódio e o servidor envia uma *stream* contínua de dados para o cliente para que ele possa ver o episódio

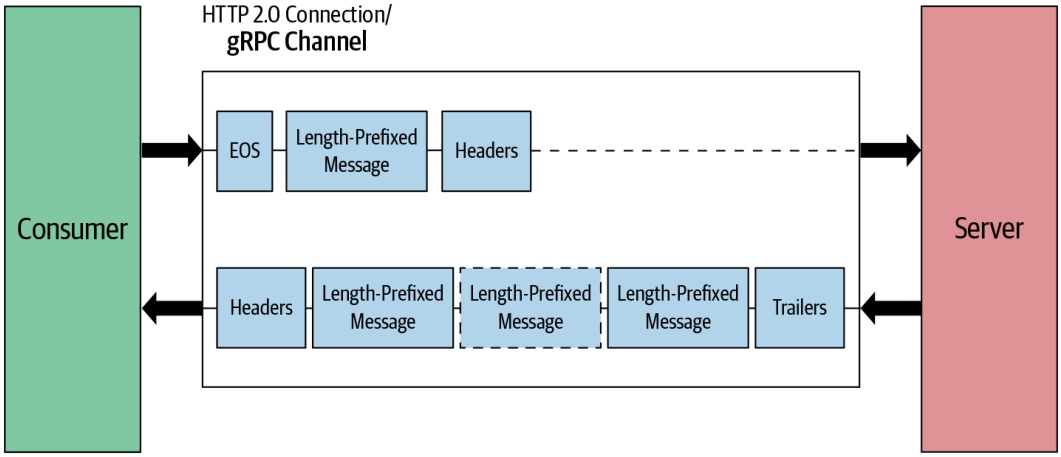


Figura 2.5: *Server streaming* RPC [25]

- *Client streaming* RPC: neste tipo é o contrário ao anterior. Neste caso, o cliente envia vários pedidos ao servidor e este apenas envia uma resposta ao cliente [24], como se pode ver na figura 2.6. Um exemplo prático pode ser um *upload* de um ficheiro em que o cliente envia uma *stream* de dados ao servidor e o servidor apenas envia uma resposta de sucesso ou insucesso

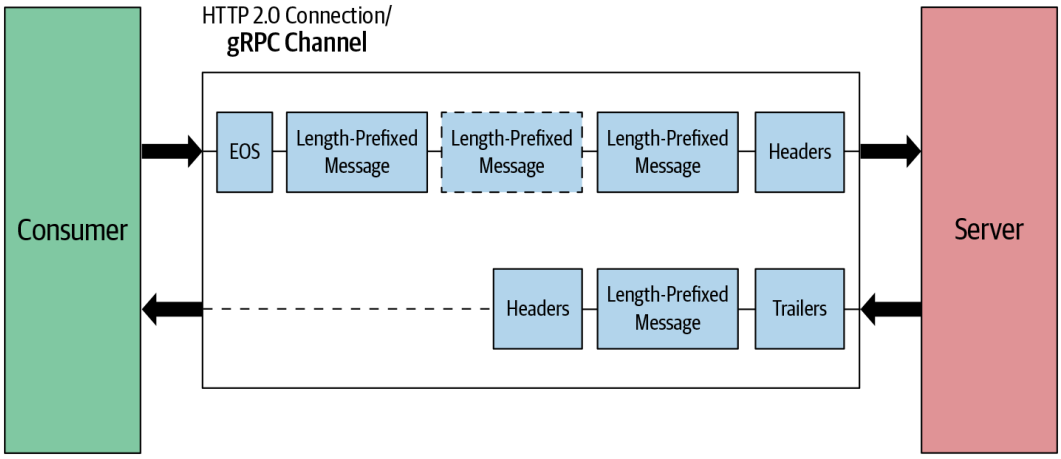


Figura 2.6: *Client streaming* RPC [25]

- *Bidirectional streaming* RPC: É a junção dos dois tipos anteriores. Neste, o cliente envia várias mensagens de pedidos ao servidor e este retorna várias mensagens de resposta ao cliente [24], como se pode ver na figura 2.7. Aqui um exemplo poderá ser um sistema de *chat* comum ou em jogos (sistemas que tenham a necessidade de comunicação em tempo real)

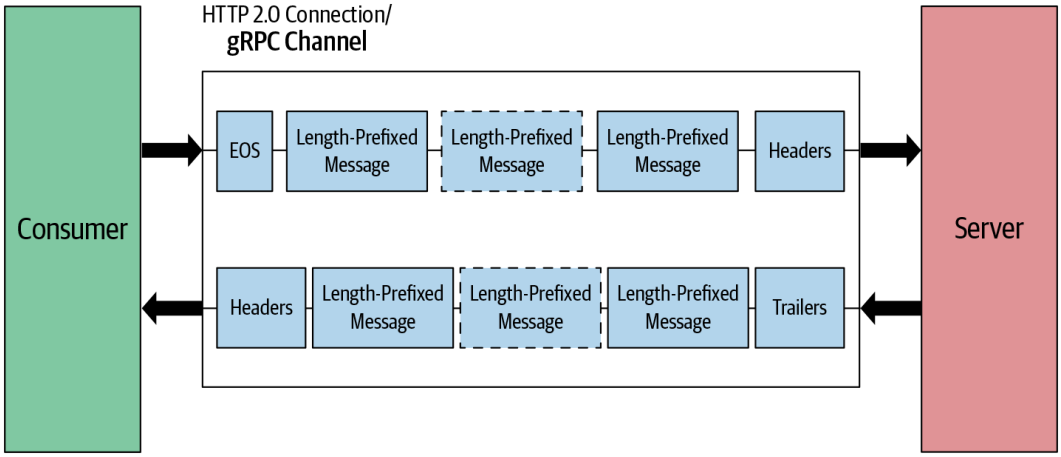


Figura 2.7: *Streaming* bidirecional RPC [25]

Na Figura 2.8, cada um destes padrões está representado graficamente de forma mais resumida:

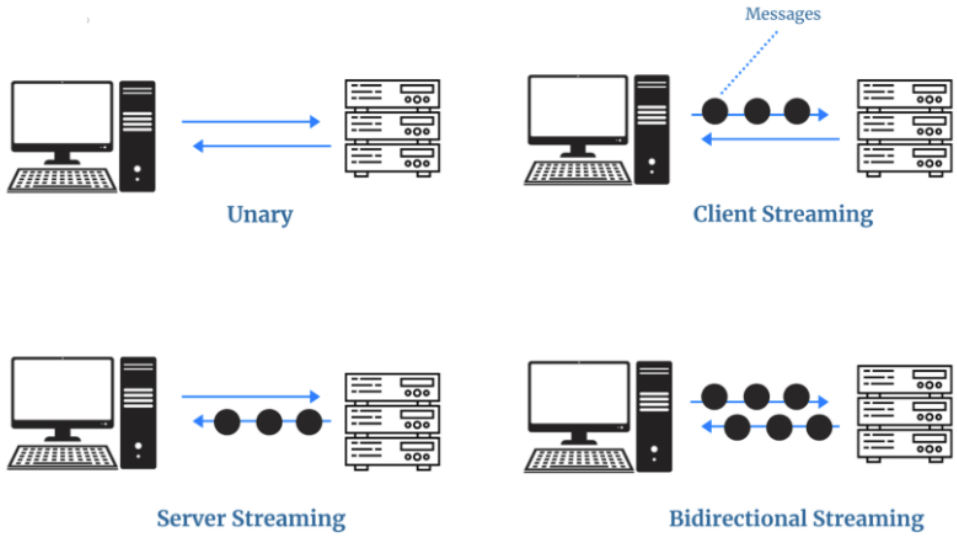


Figura 2.8: Padrões de comunicação com gRPC [26]

Pontos Fortes

Os pontos fortes da *framework* gRPC acabam por ser intuitivos após a análise das suas características. Entre outros, os pontos fortes são:

- Alto desempenho e eficiência: este é sem dúvida um dos maiores objetivos e vantagens do gRPC. A utilização de *Protocol Buffers* e HTTP/2 é uma mais-valia para gRPC
- Funciona em múltiplas plataformas e em múltiplas linguagens de programação
- Gera código automaticamente para cliente e servidor de acordo com a linguagem de programação utilizada
- Definição de serviço simples através de *Protocol Buffers*
- Vários tipos de comunicação, nomeadamente *streams* bidirecionais

Pontos Fracos

Como tudo, o gRPC também apresenta alguns pontos fracos:

- Pouca popularidade: o gRPC é considerado relativamente recente o que faz com que haja poucos desenvolvedores a utilizar gRPC. Isto faz com que haja pouco suporte fora da Google e que não haja muitas ferramentas relativas que melhorem a qualidade de vida ao programar. Para além disso, visto que não é muito explorado, a comunidade de desenvolvedores pode ter falta das melhores práticas, por exemplo. Apesar da pouca popularidade, esta está a crescer e esta desvantagem pode desaparecer em pouco tempo.

- Complexidade na configuração: É comum haver complexidade na configuração inicial dos sistemas de desenvolvimento para gRPC, nomeadamente em relação às suas dependências.
- Dificuldade na aprendizagem inicial: para compreender e implementar gRPC é necessário conhecer os *Protocol Buffers* e como eles funcionam. Caso não o façam a aprendizagem inicial, pode ser complicada.

Na tabela 2.1, é possível ver um resumo das vantagens e desvantagens em utilizar gRPC.

Tabela 2.1: Vantagens e desvantagens de gRPC

Vantagens	**Desvantagens**
Alto desempenho	Pouca popularidade
Alta eficiência	Complexidade na configuração
Multiplataforma	Dificuldade na aprendizagem inicial
Multilíngue	
Suporte de vários tipos de comunicação	
Definição de serviço simples	

2.4.2 gRPC vs REST

O gRPC e REST são duas abordagens para desenvolver APIs. Enquanto que REST é muito mais popular, não significa que seja a escolha mais acertada para todas as aplicações. Na tabela 2.2 podemos ver, lado a lado, as diferenças.

Tabela 2.2: gRPC vs REST

	gRPC	**REST**
Protocolo de transporte	HTTP/2	HTTP
Formato de mensagem	*Protocol Buffers*(geralmente)	JSON ou XML(geralmente)
Multiplataforma	Sim	Sim
Multilíngue	Sim	Sim
Complexidade de processamento	Mais baixa(binário)	Mais alta(texto)
Suporte dos navegadores	Limitado	Universal
Gerador de código	Sim(.protoc)	Não
Desempenho/performance	Alta	Baixa
Eficiência	Alta	Baixa

De notar que a escolha entre gRPC e REST depende das necessidades de cada projeto e de cada aplicação.
REST é uma boa escolha para aplicações que requerem uma comunicação mais simples e uma integração fácil com a *Web*, pois usa protocolos e formatos muito populares e de ampla aceitação.
Por outro lado, gRPC é uma *framework* moderna de alta performance e que oferece

mais eficiência do que REST e é uma boa escolha para microsserviços e situações em que a rapidez é essencial.

2.4.3 gRPC vs RPC

O gRPC é uma implementação moderna do mecanismo RPC. As principais diferenças estão nos protocolos utilizados para comunicação e nos formatos de serialização. O RPC utiliza protocolos de comunicação mais antigos, como o *Transmission Control Protocol* (TCP) e geralmente usa formatos de serialização mais simples, como o *JavaScript Object Notation* (JSON) ou o *Extensible Markup Language* (XML). Já o gRPC, geralmente utiliza *Protocol Buffers* como formato de serialização de dados e HTTP/2 (segunda versão de *Hypertext Transfer Protocol* (HTTP)) como protocolo de comunicação o que possibilita uma comunicação bidirecional (ver secção 2.2.2).

Capítulo 3

Análise do Problema

3.1 Contexto

Há uma loja de sanduíches que tem vindo a ganhar popularidade ao longo do tempo. Com o aumento da procura, o pessoal da loja percebeu que precisava de uma forma mais eficiente e prática para gerir os pedidos de reserva de sanduíches dos seus clientes. Por isso, criou-se uma aplicação que permite que o cliente visualize os ingredientes e as sanduíches existentes, efetue reservas de sanduíches com antecedência, bem como fazer avaliações, visualizar as avaliações de outros clientes e até votar de forma positiva ou negativa nas avaliações de outros clientes de forma a expressar a sua opinião relativamente às mesmas. As sanduíches são caraterizadas por um identificador privado, identificador público, designação, descrição e a sua respetiva lista de ingredientes. Cada ingrediente está inserido numa categoria.
Os utilizadores registados podem avaliar uma sanduíche fornecendo um texto e uma pontuação (0 a 5 estrelas, incluindo meias estrelas). Quando alguém vota numa avaliação, esta não pode ser apagada a menos que seja por um moderador. Para além disso, também podem fazer reservas de uma ou várias sanduíches fornecendo a lista de itens desejados e a quantidade de cada item. A reserva irá ficar num estado não ativo assim que a data ultrapassar o dia de entrega que posteriormente irá ser alterado para um estado de "entregue" ou "não entregue" por um funcionário da loja. O utilizador registado também pode cancelar a reserva.
O administrador é responsável pela criação e remoção de sanduíches, ingredientes e

as suas categorias, sendo capaz claro de utilizar todas as funcionalidades da aplicação.
O moderador é responsável por analisar as avaliações dos clientes e caso haja alguma infração pode remover as mesmas. Para além disso, também é capaz de ver as denúncias feitas às avaliações por outros utilizadores e decidir se estas efetivamente cometem alguma infração.

Uma vez que a aplicação desenvolvida (tanto em REST como gRPC) é, na prática, um prototipo, não vai ser possível o registo de novos utilizadores e os utilizadores a existir vão ser definidos e criados previamente por *bootstrap*. Para além disso, não foram definidos limites de reservas para uma determinada data nem limite de *stock* de ingredientes e sanduíches.
Devido à natureza do projeto e ao seu caráter público, em diagramas apresentados e na aplicação realizada, será utilizada a língua inglesa de forma a garantir uma melhor compreensão do projeto por parte de uma audiência global.

3.2 Modelo de Domínio

A figura 3.1 representa o modelo de domínio retirado do cenário apresentado anteriormente.

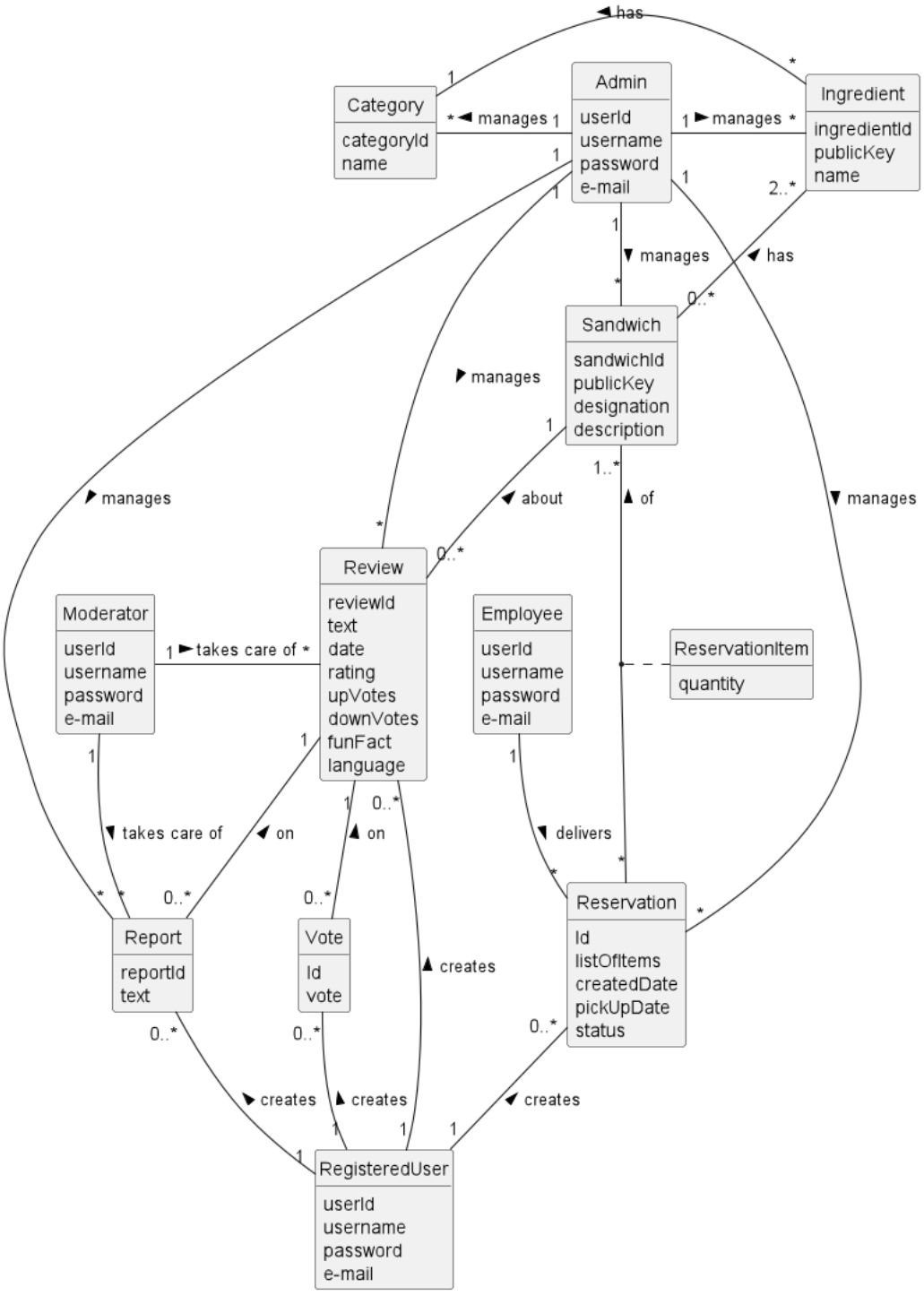


Figura 3.1: Modelo de Domínio

- ***Category***: representa uma categoria de ingredientes, com atributos como identificador e nome
- ***Ingredient***: representa um ingrediente utilizado nas sanduíches, com atributos como identificador, chave pública, nome e categoria

- ***Sandwich***: representa uma sanduíche, com atributos como identificador, chave pública, designação, descrição e uma lista de ingredientes
- ***Review***: representa uma avaliação feita por utilizadores registados sobre uma sanduíche, com atributos como identificador, texto, data, classificação, votos positivos e negativos, facto interessante e idioma
- ***Vote***: representa um voto dado pelos utilizadores sobre uma *review*, com atributos como identificador e voto (positivo ou negativo). Os "*Votes*" são semelhantes a "*likes*" e "*dislikes*" utilizados em várias plataformas
- ***Reservation***: representa uma reserva feita por utilizadores registados de uma ou mais sanduíche, com atributos como identificador, a referência ao utilizador que a fez, a lista de itens reservados, data de criação, data de recolha e estado
- ***Report***: representa uma denúncia feita por um utilizador registado a uma avaliação, com atributos como identificador e texto
- ***RegisteredUser***, ***Moderator***, ***Employee*** e ***Admin***: Representam os cargos com autoridades diferentes no sistema. Todas com atributos como identificador, nome de utilizador, palavra-passe e e-mail. O moderador faz o controlo das avaliações e das suas denúncias. O funcionário trata das reservas e de lhes mudar o estado para entregue ou não entregue. Já o utilizador registado pode visualizar as categorias, ingredientes e sanduíches existentes, fazer avaliações sobre sanduíches, votar ou denunciar as avaliações de outros utilizadores e fazer reservas de sanduíches.

3.3 Requisitos Funcionais

Tendo em conta o cenário anterior e dividindo-o pelas áreas identificadas, foram desenvolvidos diagramas relacionadas com as funcionalidades esperadas da aplicação. Nas Figuras 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 3.9, essas funcionalidades estão explícitas. No cenário existem cinco atores: anónimo, utilizador registado, moderador, funcionário e administrador e é importante destacar que o ator administrador ou "*Admin*" tem a capacidade de utilizar todas as funcionalidades da aplicação, no entanto, nos diagramas, apenas estão identificadas aquelas que diferem dos outros atores, com o objetivo de facilitar a leitura dos diagramas. Esta simplificação também acontece com os outros atores e é importante ter em conta a hierarquia neste protótipo. O ator com mais "poder" é o administrador, em segundo lugar é o funcionário, seguido do moderador que é seguido do utilizador registado que está acima de anónimos, ou seja, utilizadores não registados.

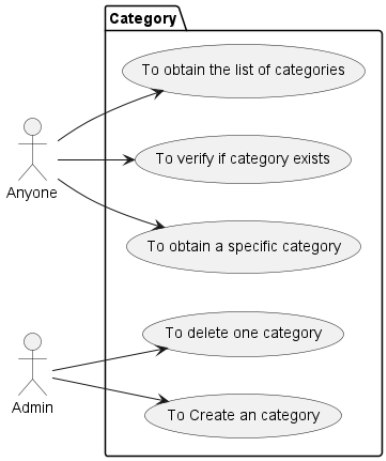


Figura 3.2: Diagrama de Casos de uso para *Category*

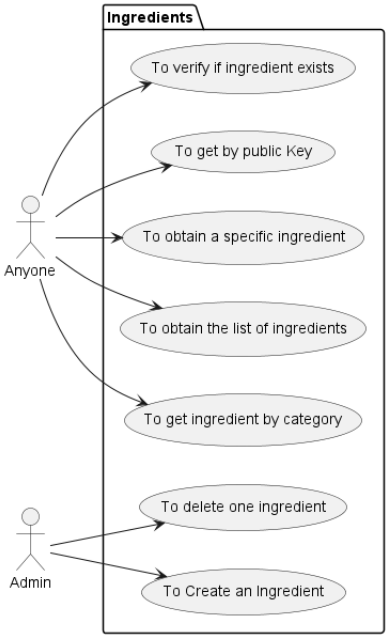


Figura 3.3: Diagrama de Casos de uso para *Ingredient*

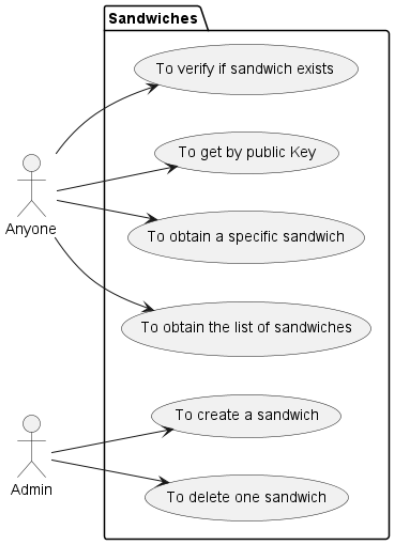


Figura 3.4: Diagrama de Casos de uso para *Sandwich*

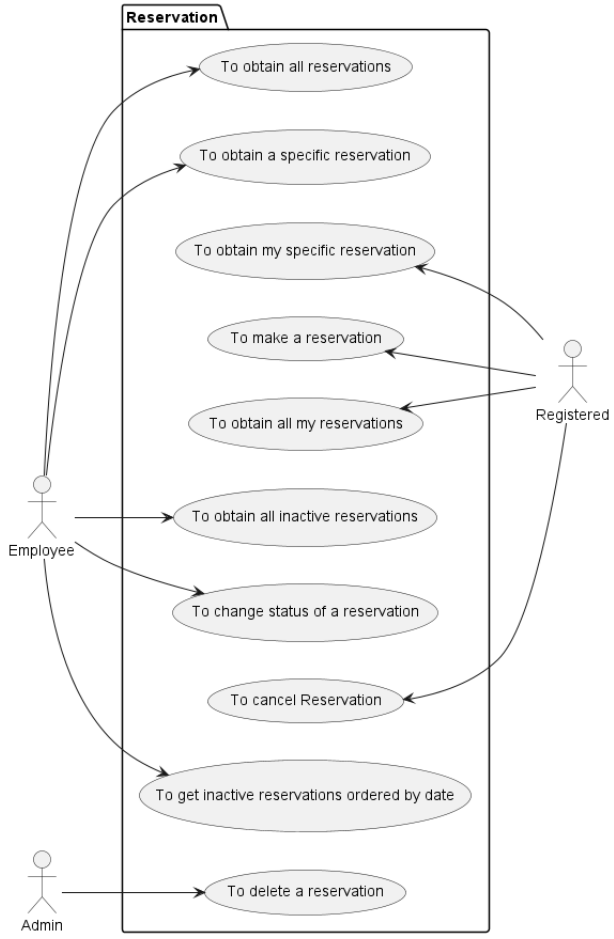


Figura 3.5: Diagrama de Casos de uso para *Reservation*

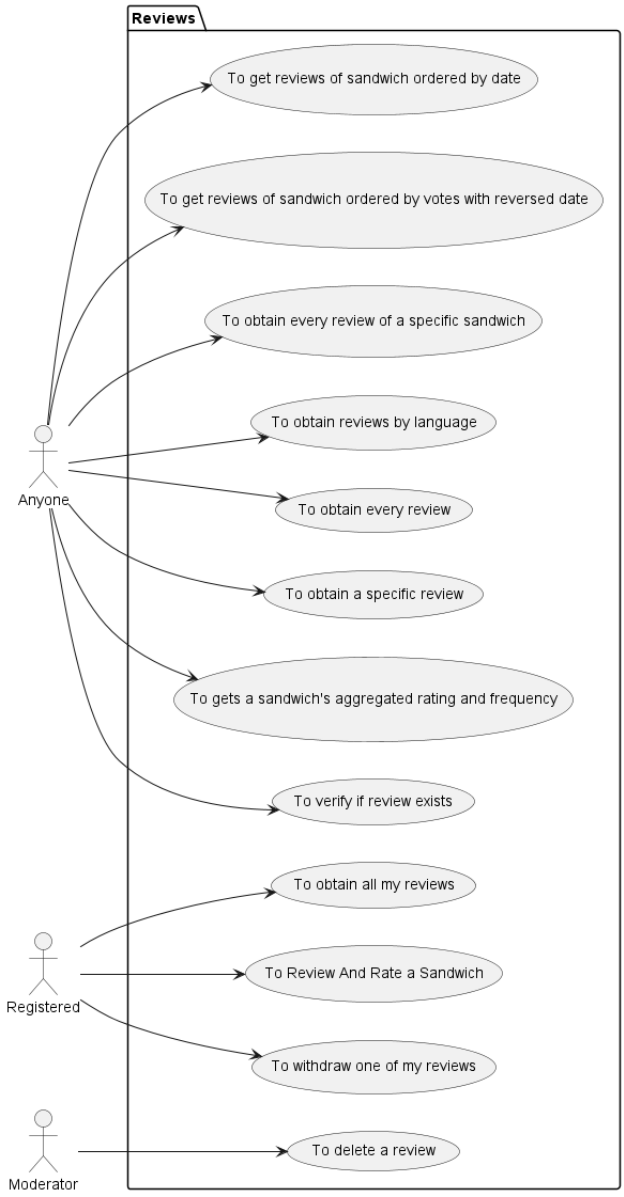


Figura 3.6: Diagrama de Casos de uso para *Review*

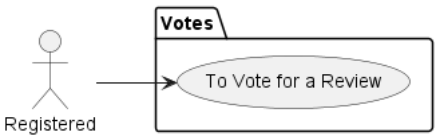


Figura 3.7: Diagrama de Casos de uso para *Vote*

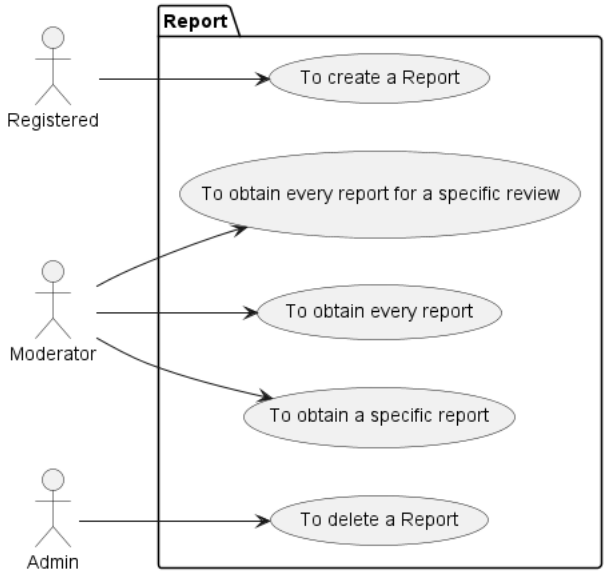


Figura 3.8: Diagrama de Casos de uso para *Report*

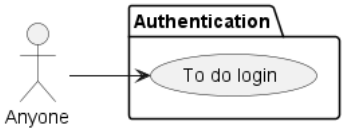


Figura 3.9: Diagrama de Casos de uso para *Authentication*

3.4 Restrições de implementação

- A arquitetura da aplicação deve ser baseada em microsserviços
- A Linguagem utilizada em toda a aplicação é Java. A utilização de linguagens diferentes em cada serviço não tem importância para os objetivos do projeto e como tal, Java surgiu naturalmente como a opção a ser utilizada
- O formato de serialização a ser utilizado na aplicação em gRPC é *Protocol Buffers*. Poderia ser outro, no entanto, como nesta aplicação o objetivo é o desempenho e é comum utilizar Protobuf em aplicações que utilizem gRPC então o formato utilizado foi *Protocol Buffers*
- A aplicação deve utilizar gRPC para comunicar entre serviços. Este é o foco principal do projeto

3.5 Atributos de Qualidade

A ISO/IEC 25010 é uma norma internacional que define um modelo de qualidade para a avaliação de um produto de software e determina quais as características de

qualidade que serão tidas em conta na avaliação. A qualidade de um sistema é o grau em que o sistema satisfaz as necessidades declaradas e implícitas das suas várias partes interessadas. O modelo de qualidade do produto definido na norma ISO/IEC 25010 inclui oito características de qualidade: Desempenho, funcionalidade, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade, portabilidade [27].
Neste prototipo os atributos de qualidade necessários são:

- Desempenho
  - A aplicação deve ser capaz de lidar com um grande volume de solicitações de criação de reserva rapidamente e deve responder em, no máximo, 3 segundos
- Segurança
  - A aplicação deve garantir a autenticação de utilizadores, evitando que pessoas não autorizadas tenham acesso recursos da aplicação
- Confiabilidade
  - A aplicação deve ter testes que comprovem e validem as suas funcionalidades
  - A aplicação deve ser capaz de lidar com erros e exceções de forma adequada e fornecer *feedback* aos utilizadores
- Manutenibilidade
  - As classes da aplicação devem assegurar uma cobertura de, pelo menos, 80% através de testes unitários

Capítulo 4

Conceção da Solução

Neste capítulo vai ser apresentada a representação arquitetural da aplicação e vão ser utilizados dois modelos para isso, C4 [28] e 4+1 [29].

O Modelo C4 apresenta o sistema com diferentes níveis de abstração e o modelo de vistas 4+1 apresenta o sistema de diferentes perspetivas. Ao combinar os dois modelos torna-se possível representar o sistema de diversas perspetivas, cada uma com vários níveis de detalhe. O modelo 4+1 é composto por vista lógica, vista física, vista de processos, vista de desenvolvimento e vista de cenários [29]. Já o modelo C4 faz a descrição do software com 4 níveis de abstração [28]:

- Nível 1: Contexto do sistema como um todo
- Nível 2: Contentores do sistema (aplicações, *databases*, microsserviços, etc.)
- Nível 3: Componentes dos contentores (*controllers*, *services*, etc.)
- Nível 4: Descrição do código

Nos diagramas realizados para este projeto, não são explorados todos os níveis de abstração uma vez que não são relevantes para a compreensão do projeto.

4.1 Protótipo que utiliza gRPC

Nesta secção irá ser apresentada a documentação realizada para o protótipo que utiliza gRPC para comunicar entre serviços.

4.1.1 Vista Lógica

A vista lógica, baseada em microsserviços e relativa aos aspetos do software, vem responder aos desafios do negócio e é possível identificar como as entidades de negócio se relacionam para fornecer as funcionalidades completas do sistema. Para além disso, também identifica as interfaces dos serviços. Nas Figuras 4.2 e 4.3 está representada a vista lógica de nível 1 e 2, respetivamente.

De forma a auxiliar a compreensão dos possíveis estados que uma reserva pode ter, foi criado um diagrama de estado apresentado na figura 4.1. Partindo do estado inicial e imaginando que uma reserva foi criada, ela começa e ficará num estado "ativo" até que se ultrapasse a data de entrega ou caso o utilizador cancele a mesma. Se o utilizador cancelar a reserva, esta ficará num estado "cancelado", por outro lado, se a data de entrega for ultrapassada, esta ficará num estado inativo até que um funcionário conclua este processo mudando o estado para "entregue" ou "não entregue".

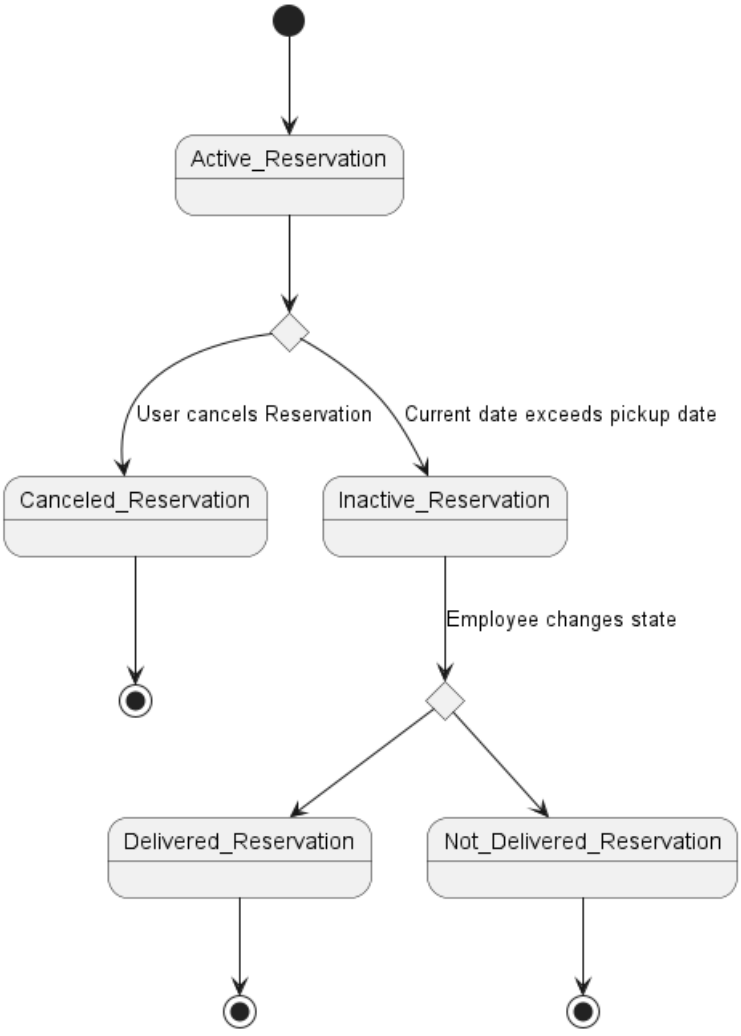


Figura 4.1: Diagrama de Estado

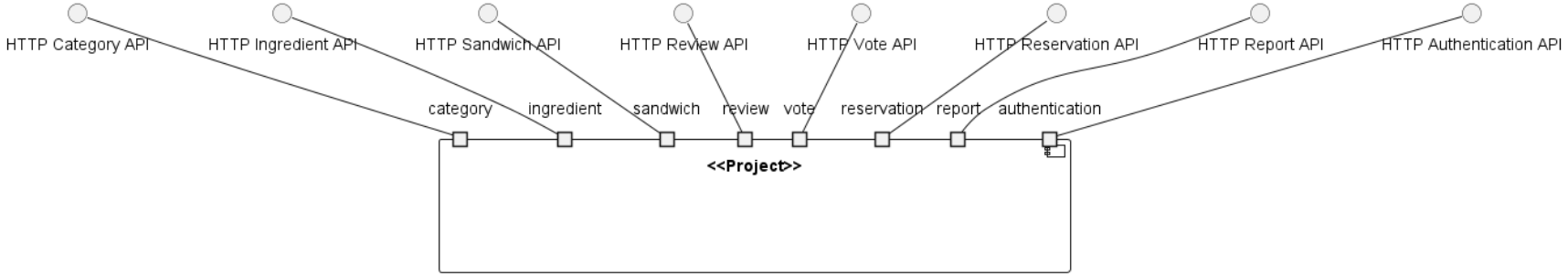


Figura 4.2: Vista Lógica Nível 1

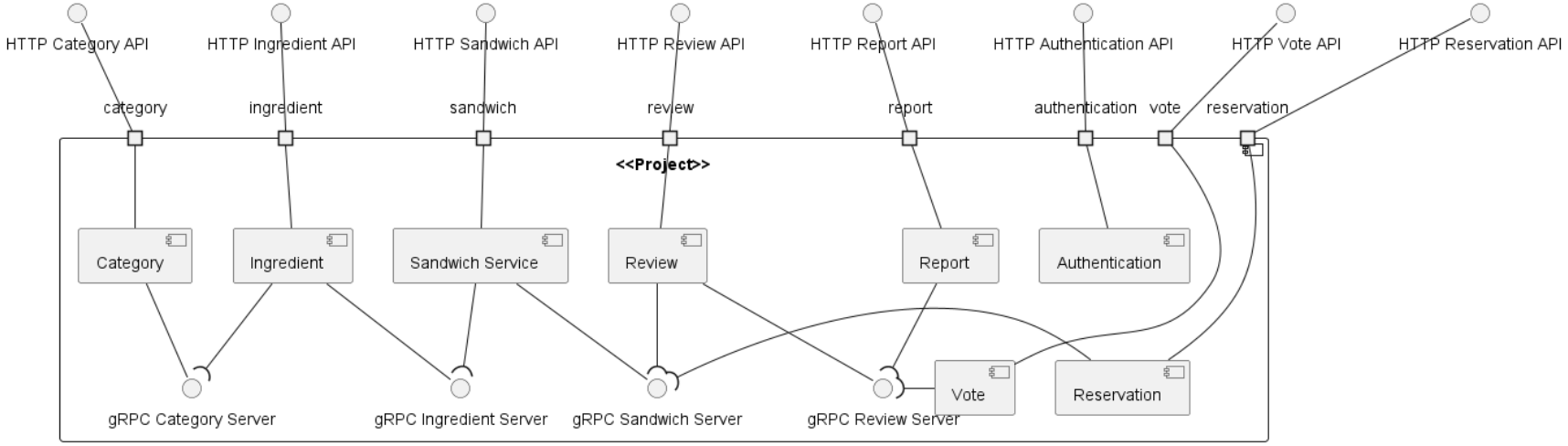


Figura 4.3: Vista Lógica Nível 2

4.1.2 Vista Física

A vista física do cenário (Figura 4.4) também é baseada em microsserviços, e envolve a interação entre várias aplicações que comunicam entre si, utilizando gRPC. A vista física também nos permite perceber e mapear os diferentes componentes de software.

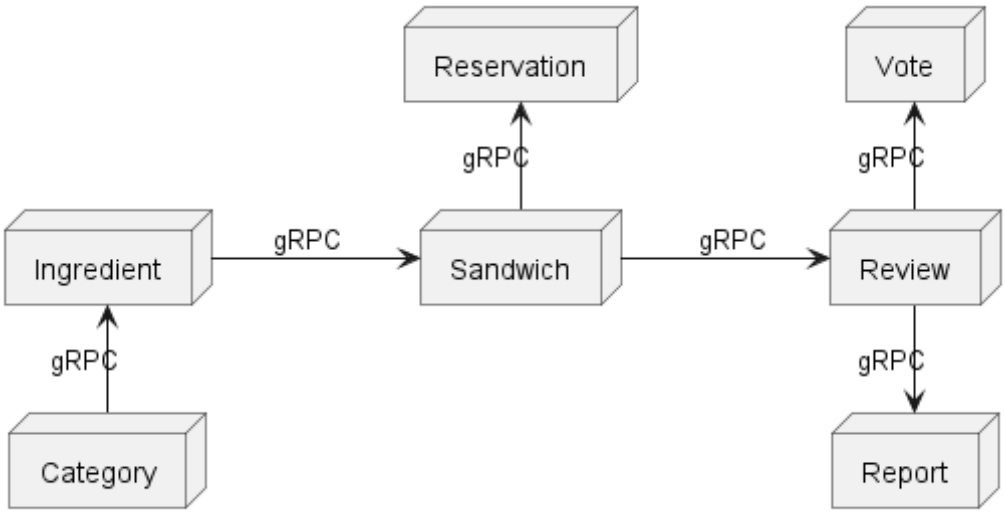


Figura 4.4: Vista física

4.1.3 Vista de Processo

A vista de processos é relativa ao fluxo de processos ou interações no sistema envolvidas na conclusão de uma determinada atividade. Neste projeto, são demonstrados 2 exemplos de ações que são possíveis realizar na aplicação (Requisitar o catálogo de sanduíches e criar uma reserva). No primeiro exemplo, é apresentado um diagrama de sequência de nível 3 e no segundo exemplo são apresentados 2 diagramas de sequência, um de nível 2 e outro de nível 3. O segundo exemplo foi acompanhado de um diagrama de nível 2 para facilitar a compreensão do leitor devido à alta complexidade do diagrama de nível 3.

Nesta secção, são dados dois exemplos de diagramas de sequência, no entanto, é importante destacar que todos os pedidos *GET* são semelhantes ao exemplo fornecido em termos lógica de fluxos de processo e o mesmo se aplica aos pedidos *POST*. Em todos os pedidos *POST* é necessário fazer um pedido em gRPC a outro serviço. Essa mesma lógica pode ser vista na figura 4.3.

O primeiro exemplo é ilustrado na Figura 4.5 e representa o processo de requisição do catalogo de sanduíches.

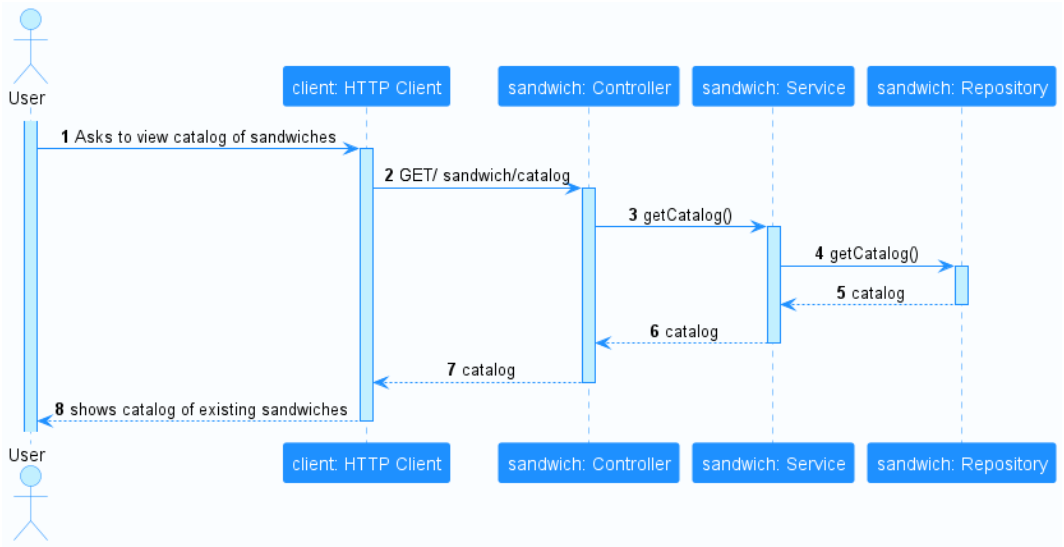


Figura 4.5: Diagrama de Sequência para requisitar o catalogo de sanduíches

O segundo exemplo é apresentado nos Diagramas de Sequência das Figuras 4.6 e 4.7 e representam o processo de criação de uma reserva.

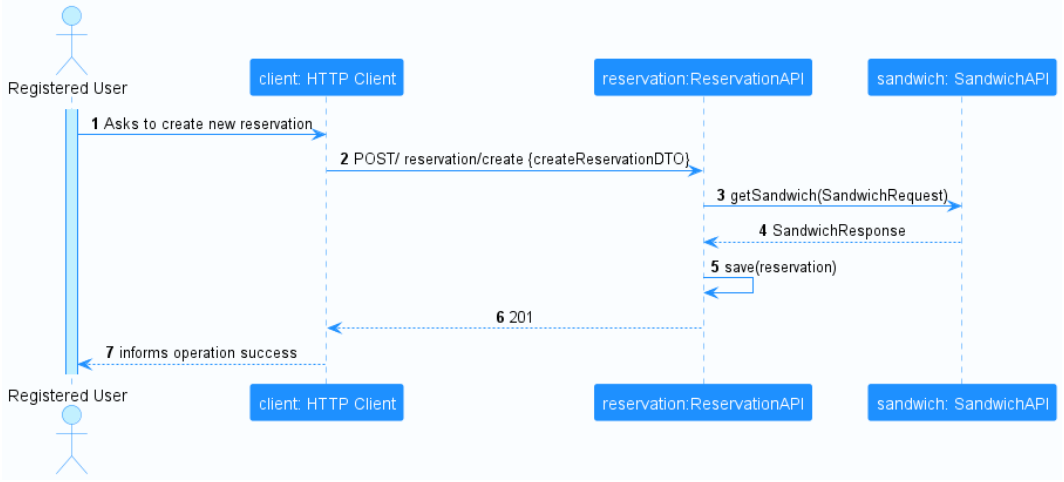


Figura 4.6: Diagrama de sequência de nível 2 para criar uma reserva

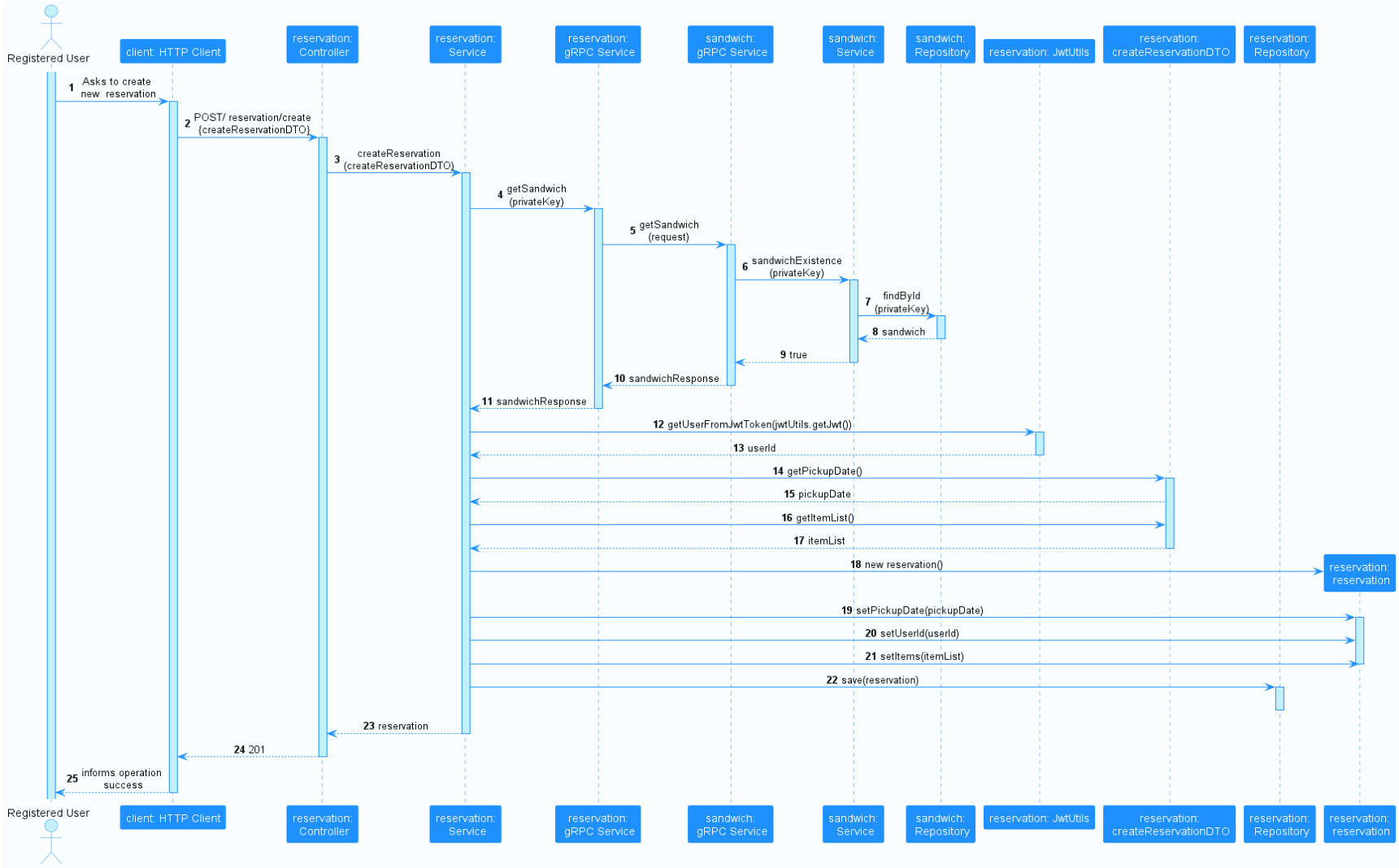


Figura 4.7: Diagrama de sequência de nível 3 para criar uma reserva

4.1.4 Vista de Implementação

Na figura 4.8, encontra-se representada a vista de implementação. Nestas consegue-se identificar de que forma os componentes se encontram organizados no ambiente de desenvolvimento.

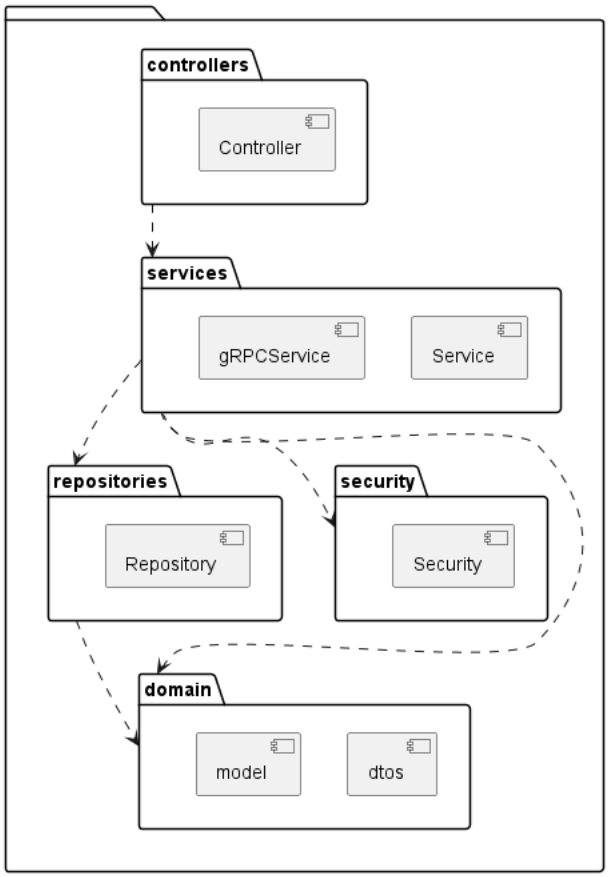


Figura 4.8: Vista de Implementação de nível 3

4.2 Protótipo que utiliza REST

Nesta secção irá ser apresentada a documentação realizada para o protótipo que utiliza REST para comunicar entre serviços. Quaisquer diagramas que seriam repetidos não serão apresentados.

4.2.1 Vista Lógica

Na vista lógica, o diagrama de estados e a vista lógica de nível 1 são comuns ao protótipo que utiliza gRPC. Na figura 4.9 está representada a vista lógica de nível 2.

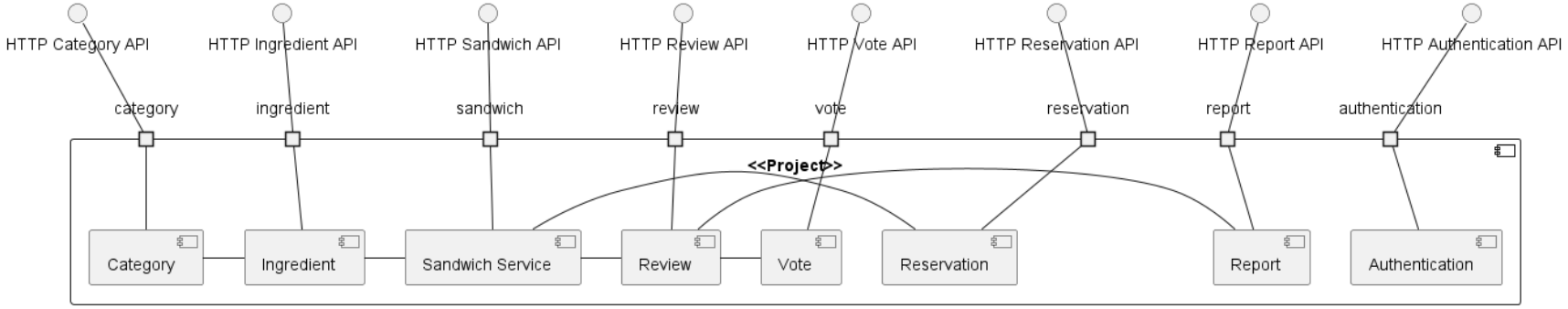


Figura 4.9: Vista Lógica Nível 2

4.2.2 Vista Física

A vista física (Figura 4.10) envolve a interação entre várias aplicações que comunicam entre si, utilizando HTTP.

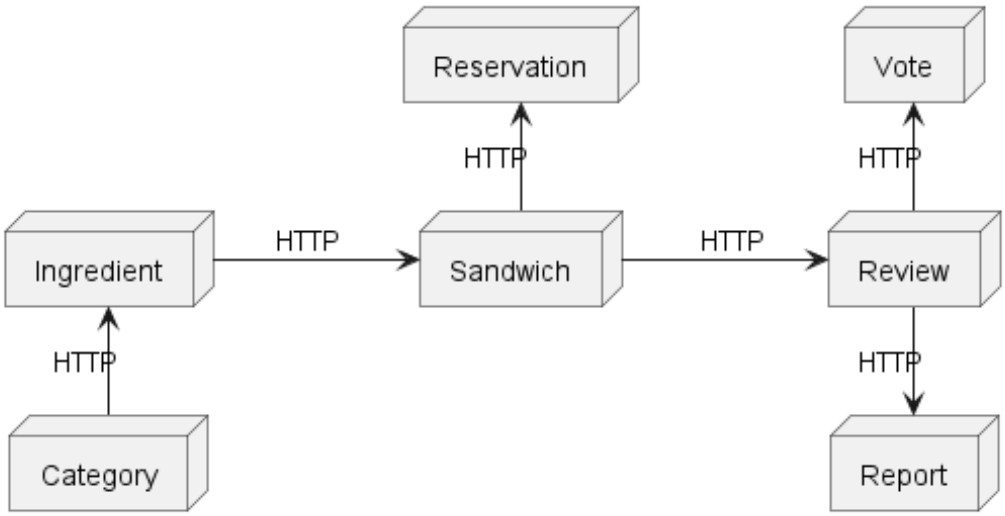


Figura 4.10: Vista física em REST

4.2.3 Vista de Processo

Na vista de processo, o diagrama de sequência para requisitar o catalogo de sanduíches é o mesmo. Nas figuras 4.11 e 4.12, estão representados os diagramas de sequência de nível 2 e 3, respetivamente, para criar uma reserva.

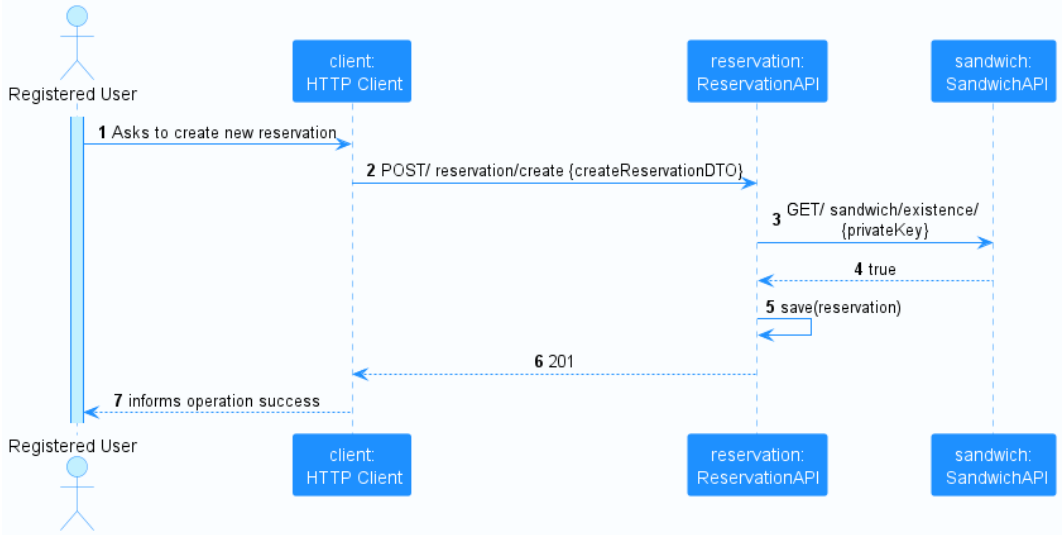


Figura 4.11: Diagrama de sequência de nível 2 para criar uma reserva

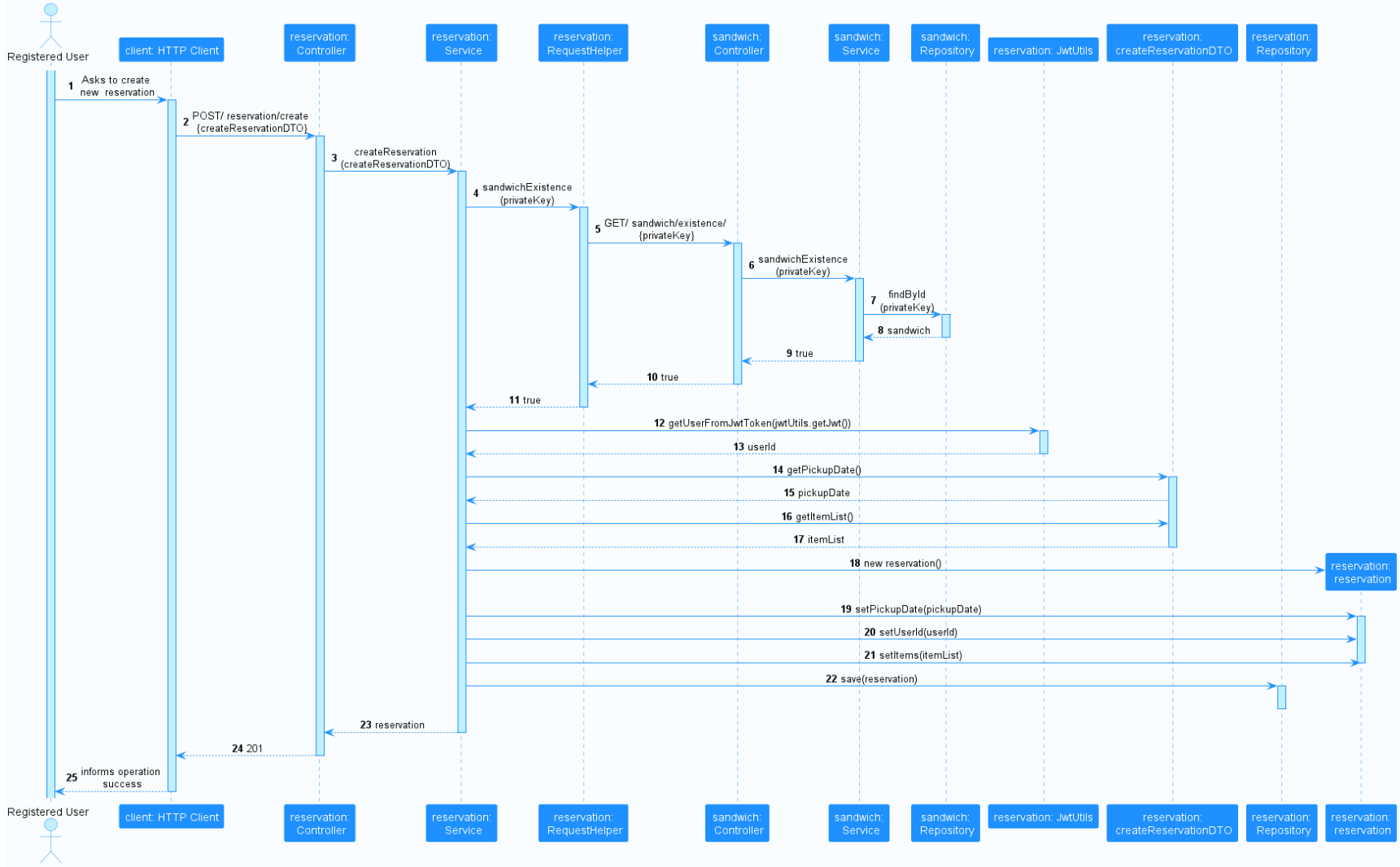


Figura 4.12: Diagrama de sequência de nível 3 para criar uma reserva

4.2.4 Vista de Implementação

Na figura 4.13 está representada a vista de implementação.

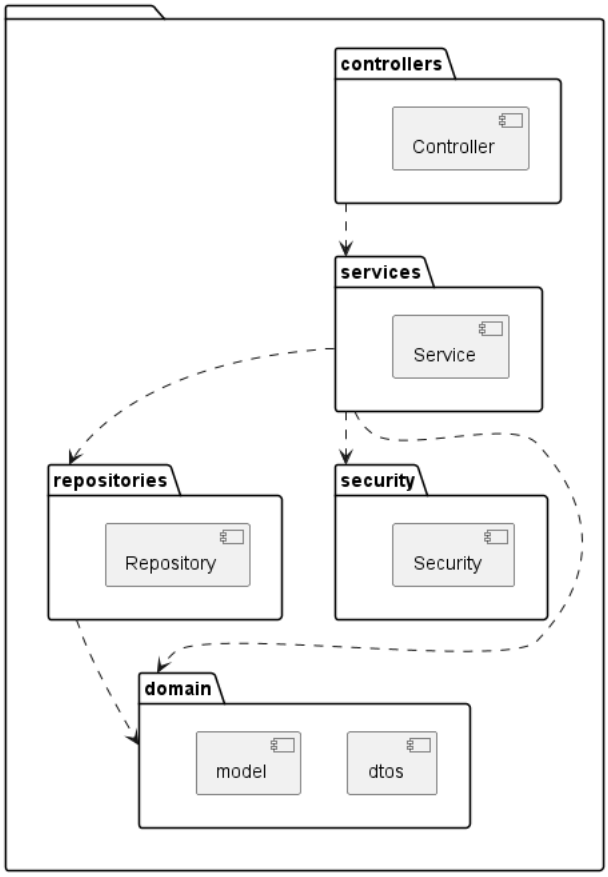


Figura 4.13: Vista de Implementação de nível 3

4.3 *Design* alternativo

Na figura 4.14, é possível compreender uma das caraterísticas da arquitetura proposta e adotada, conhecida como "*Database per service*", que se refere ao fato de cada microsserviço possui a sua própria base de dados [30]. Os serviços são expostos através de APIs e o diagrama também mostra as dependências entre os serviços. Por exemplo, o serviço das sandes precisa do serviço dos ingredientes para obter informações relativamente à existência de cada ingrediente da sandes. Já na figura 4.15, é possível observar uma possível alternativa, onde se pode observar que os serviços possuem uma base de dados centralizada. Esta alternativa é mais rápida, no entanto, peca em termos de segurança.

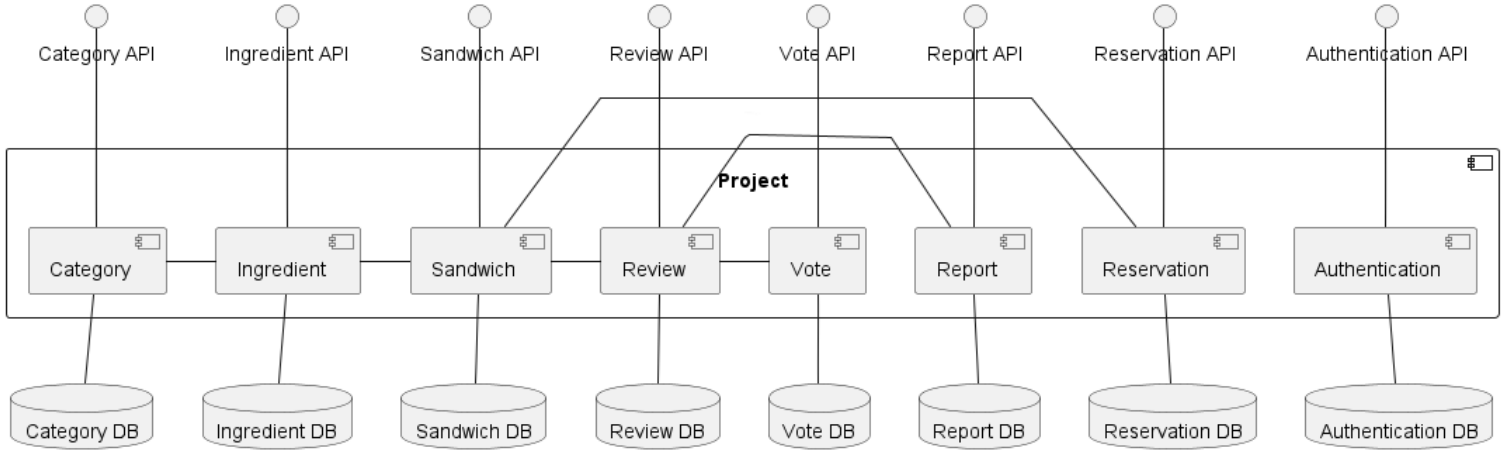


Figura 4.14: *Database per service*

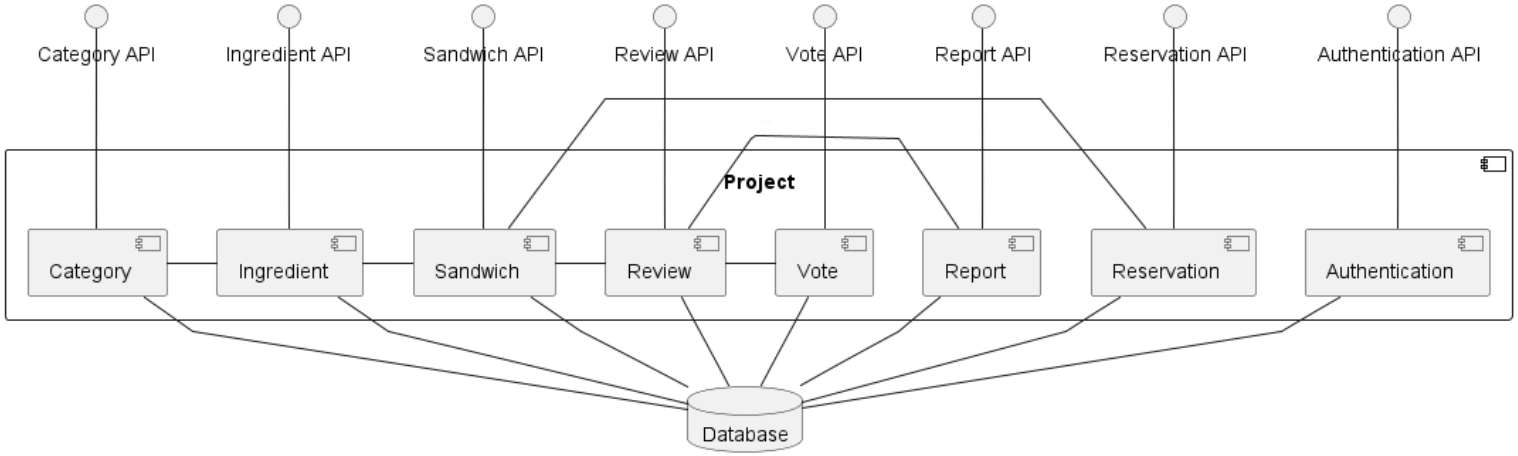


Figura 4.15: *Database* Centralizada

Capítulo 5

Implementação da Solução

Como foi referido anteriormente, o projeto é implementado utilizando REST e gRPC como meio de comunicação entre serviços tendo sido realizados o prototipo para cada um dos casos, dando origem a um total de 16 aplicações individuais desenvolvidas.

Ao longo deste capítulo, é apresentada a implementação da solução proposta ao longo dos capítulos 3 e 4 tanto em REST como em gRPC realçando sobretudo, a aplicação em gRPC, uma vez que a exploração desse meio de comunicação é o principal foco do projeto. Ambas as aplicações são desenvolvidas em *Java Spring Boot*.

No fim, são realizados testes de desempenho com o objetivo de responderem ao atributo de qualidade de desempenho definido e de comparar a implementação em gRPC com a implementação em REST e verificar se há uma diferença significativa entre ambos.

5.1 Planeamento e Configuração Inicial

Uma das coisas que é muito dispendiosa de tempo para novos utilizadores de gRPC é a configuração inicial e perceber que há uma compatibilidade entre as versões das diferentes ferramentas (gRPC e *Spring Boot*, por exemplo) a cumprir e, sendo assim, é necessário estar atento às versões a utilizar em cada um dos projetos criados. Em relação às versões de *Spring boot* e gRPC, é possível consultar [31] para mais detalhe.

A abordagem inicial para o projeto foi a exploração de gRPC como meio de comunicação com várias tentativas de utilização do mesmo para fazer um simples serviço

de "*Hello world*". Assim que já estavam acertadas as dependências necessárias e as versões das mesmas, na implementação da solução proposta para este trabalho decidiu-se começar pela conceção do protótipo que adota REST como estilo arquitetural. Esta decisão baseou-se, além do facto de existir alguma familiaridade para com a abordagem, o que por sua vez traz rapidez de desenvolvimento mas também pelo facto de uma pequena parte da solução já estar adiantada de um outro projeto e serviu como base para a restante implementação.

5.1.1 Dependências e Propriedades Necessárias para gRPC

Na listagem de código 5.1 é possível ver a propriedades presentes no ficheiro *pom.xml*. Inicialmente, é definida a versão do Java (propriedade predefinida) e após isso, é definida a versão de gRPC e a versão do Protobuf a utilizar nas dependências e *plugins*.

```
<properties>
    <java.version>17</java.version>
    <!--propriedades para gRPC-->
    <grpc.version>1.51.0</grpc.version>
    <protobuf.version>3.21.7</protobuf.version>
</properties>
```

Listagem 5.1: Propriedades para gRPC no ficheiro pom.xml.

Na listagem de código 5.2, é possível ver as dependências necessárias para utilizar gRPC em *Java Spring Boot*.

```
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-netty-shaded</artifactId>
    <version>${grpc.version}</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-protobuf</artifactId>
    <version>${grpc.version}</version>
</dependency>
<dependency>
    <groupId>io.grpc</groupId>
    <artifactId>grpc-stub</artifactId>
    <version>${grpc.version}</version>
</dependency>
<dependency>
```

```
18      <groupId>net.devh</groupId>
19      <artifactId>grpc-spring-boot-starter</artifactId>
20      <version>2.14.0.RELEASE</version>
21  </dependency>
```

Listagem 5.2: Dependências para gRPC no ficheiro pom.xml.

Na listagem de código 5.3, é possível ver o *plugin* que gera os ficheiros dos serviços definidos em *Protocol Buffers (Protobuf)*.

```
1  <plugin>
2      <groupId>org.xolstice.maven.plugins</groupId>
3      <artifactId>protobuf-maven-plugin</artifactId>
4      <version>0.6.1</version>
5      <configuration>
6          <protocArtifact>com.google.protobuf:protoc:${protobuf.
               version}:exe:${os.detected.classifier}</protocArtifact>
7          <pluginId>grpc-java</pluginId>
8          <pluginArtifact>io.grpc:protoc-gen-grpc-java:${grpc.
               version}:exe:${os.detected.classifier}</pluginArtifact>
9      </configuration>
10     <executions>
11         <execution>
12             <goals>
13                 <goal>compile</goal>
14                 <goal>compile-custom</goal>
15             </goals>
16         </execution>
17     </executions>
18 </plugin>
```

Listagem 5.3: Plugin do Protobuf no ficheiro pom.xml.

5.2 Detalhes da Implementação

Este tópico tem como objetivo dar a conhecer os detalhes mais relevantes de implementação dos casos de uso da aplicação. Uma vez que a aplicação é muito extensa e havendo 8 serviços para cada abordagem (REST e gRPC) e o comportamento do sistema é semelhante no que toca à comunicação entre serviços em todos os casos de uso, só se irá entrar em detalhes de implementação da estrutura de comunicação em um dos casos de usos cujo diagrama de sequência já foi exposto no capítulo anterior na figura 4.7.

A criação de uma reserva de sanduíches é iniciada no cliente HTTP uma vez que todo o sistema realizado é *back-end*, não havendo Interface de Utilizador. No

caso, a ferramenta utilizada como cliente HTTP foi o *Postman*. Como se pode ver na figura 5.1, o pedido é acompanhado de um *body* composto pela data de entrega e pela lista de sanduíches e, para além disso, deve ser acompanhado de um *Bearer token* que é gerado pelo serviço de autorização que identifica o utilizador que está a fazer a reserva.

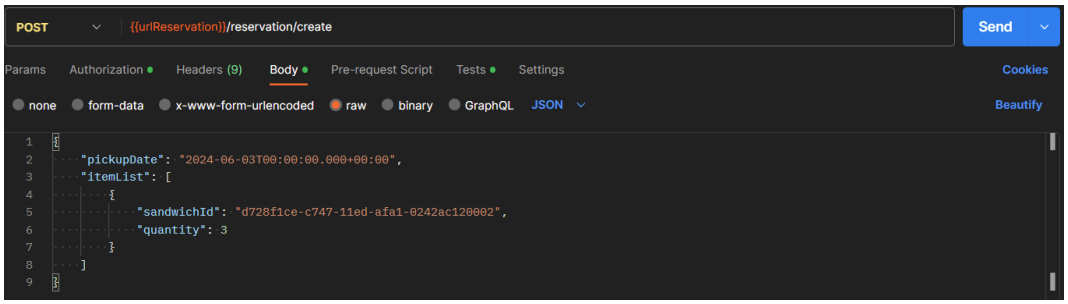


Figura 5.1: Pedido para fazer reserva de sanduíches

Os dados são recebidos pela camada *Controller* que está à espera de um objeto do tipo "CreateReservationDTO" como se pode ver na Listagem 5.4, que é composto exatamente pela data de entrega e pela lista de itens da reserva.

```
1 @PostMapping(value = "/create")
2 @ResponseStatus(HttpStatus.CREATED)
3 public ResponseEntity<Reservation> createReservation(@Valid
      @RequestBody final CreateReservationDTO createReservationDTO){
4     final Reservation reservation = service.createReservation(
          createReservationDTO);
5     return ResponseEntity.status(HttpStatus.CREATED).body(
          reservation);
6 }
```

Listagem 5.4: Método createReservation do Controller

O método no *Controller* encaminha o pedido para a camada *Service*. Em primeiro lugar, ocorre a deteção da data atual do sistema seguida da deteção do utilizador a partir do *JWT Token* que acompanhou o pedido feito no cliente HTTP. De seguida, foi feita uma verificação se a data de entrega definida pelo utilizador não é anterior ao dia em que ele está a fazer a reserva, caso isso aconteça é lançada uma exceção com a informação de que não é possível fazer isso. Após isso, num ciclo "*for*" é feita a verificação se cada sanduíche selecionada existe, havendo comunicação com o serviço das sanduíches uma vez que este é o responsável pelas mesmas e este é um ponto-chave em que a implementação em REST difere da implementação em gRPC. De seguida, é criado o objeto do tipo "*Reservation*" e os seus atributos são atualizados utilizando os "*setters*" da classe que irão garantir que o objeto está de

acordo com as regras do negócio. Finalmente o objeto é guardado no repositório e a operação de criação de uma reserva está concluída.

Na camada *Service* é onde começam as diferenças entre a implementação utilizando gRPC e a que utiliza REST.

```
1  public Reservation createReservation(CreateReservationDTO
       createReservationDTO)  {
2      long millis = System.currentTimeMillis();
3      Date atual = new Date(millis);
4      Long userId = Long.valueOf(jwtUtils.getUserFromJwtToken(
           jwtUtils.getJwt()));
5
6      if (createReservationDTO.getPickupDate() != null &&
           createReservationDTO.getPickupDate().before(atual)) {
7          throw new ResponseStatusException(HttpStatus.BAD_REQUEST,"
               Pickup date cannot be before creation date");
8      }
9
10     List<ReservationItem> reservationItemList =
           createReservationDTO.getItemList();
11     Utils.verifyDuplicatedItems(reservationItemList);
12
13     for(int i=0; i<reservationItemList.size();i++){
14         SandwichResponse response = sandwichGrpcService.
               getSandwich(createReservationDTO.getItemList().get(i).
               getSandwichId());
15         if(response.getCode()==404){
16             throw new ResponseStatusException(HttpStatus.NOT_FOUND
                   ,"That sandwich does not exist: "+
                   createReservationDTO.getItemList().get(i).
                   getSandwichId());
17         }
18     }
19
20     Reservation reservation = new Reservation();
21
22     reservation.setPickupDate(createReservationDTO.getPickupDate()
           );
23     reservation.setItems(createReservationDTO.getItemList());
24     reservation.setUserId(userId);
25
26
27     reservation.setCreationDate(atual);
28     reservation.setStatus(Reservation.ReservationStatus.ACTIVE);
29     repository.save(reservation);
30     return reservation;
31 }
```

Listagem 5.5: Método createReservation do ReservationService

5.2.1 gRPC

Analisando agora com mais detalhe o serviço gRPC que faz a comunicação entre microsserviços, utilizado na linha 13 da listagem de código 5.5.

Inicialmente, foi necessário definir o serviço *SandwichService.proto* visível na listagem 5.6, tanto no serviço das reservas como no serviço das sanduíches. Com o auxílio do *plugin* da listagem de código 5.3, foram gerados os ficheiros necessários para que o método "*getSandwich*" pudesse ser chamado noutros locais.

```
1  syntax = "proto3";
2  option java_multiple_files = true;
3  package com.joao.sandwich;
4
5  message SandwichRequest {
6    string sandwichId = 1;
7  }
8
9  message SandwichResponse {
10   int32 code = 1;
11 }
12
13 service SandwichService {
14   rpc getSandwich(SandwichRequest) returns (SandwichResponse);
15 }
```

Listagem 5.6: Serviço *SandwichService.proto*

Na figura 5.2, é possível observar os ficheiros gerados pelo plugin para o serviço definido em cima.

Figura 5.2: Ficheiros gerados pelo *plugin*

gRPC *Client*

Tendo os ficheiros gerados, o próximo passo foi a implementação do serviço *Grpc-Client* para efetivamente poder fazer pedidos gRPC ao serviço das sanduíches. É possível observar a classe na listagem de código 5.7.

```
1  @Service
2  public class SandwichGrpcServiceImpl implements
       SandwichGrpcService {
3
4      @GrpcClient("SandwichClient")
5      private SandwichServiceGrpc.SandwichServiceBlockingStub
           sandwichServiceStub;
6      private ManagedChannel channel;
7
8      public SandwichGrpcServiceImpl() {
9          channel = ManagedChannelBuilder.forAddress("localhost",
               9091)
10                 .usePlaintext()
11                 .build();
12     }
13
14     @Override
15     public SandwichResponse getSandwich(UUID sandwichId) {
16         sandwichServiceStub=SandwichServiceGrpc.newBlockingStub(
               channel);
17         SandwichResponse sandwichResponse = sandwichServiceStub.
               getSandwich(
18                 SandwichRequest.newBuilder()
19                         .setSandwichId(sandwichId.toString())
20                         .build()
```

```
21          );
22          return sandwichResponse;
23      }
24  }
```

Listagem 5.7: Classe SandwichGrpcServiceImpl

Na classe "*SandwichGrpcServiceImpl*", o tipo *SandwichServiceBlockingStub* da linha 5 é uma classe gerada automaticamente pelo protocolo gRPC com base no arquivo de definição do serviço. Utilizou-se a anotação "*@GrpcClient*"para indicar que o objeto "*sandwichServiceStub*" é um cliente gRPC (Pode-se dar o nome que se queira desde que não se repita).

Um "*stub*" é uma representação do servidor do serviço gRPC e fornece métodos que correspondem às operações definidas no serviço (ficheiro .proto). Ele age como uma interface para o cliente interagir com o servidor gRPC. O *stub* é responsável por encapsular os detalhes de comunicação de rede, serialização e desserialização dos dados, permitindo que o cliente chame os métodos do serviço (que na prática são remotos) como se estivesse a chamar métodos locais. No caso utilizou-se um *Blocking Stub* que é o equivalente a dizer *stub* síncrono. Caso se quisesse fazer um pedido assíncrono em gRPC deve-se usar um "*Future Stub*".

De seguida, foi declarado o canal gRPC usado para a comunicação remota. O tipo *ManagedChannel* provém das dependências da listagem 5.2. Após isso, configurou-se o canal definindo o nome e a porta do servidor. Finalmente, é feita a chamada ao método *getSandwich* para o qual é necessário fornecer uma mensagem do tipo "*SandwichRequest*" e o resultado fica armazenado num objeto do tipo "*SandwichResponse*" tal como foi definido no serviço da listagem 5.6.

gRPC Server

Do lado do servidor, foi necessária a implementação do serviço *GrpcService (server)* para poder responder aos pedidos gRPC dos clientes. É possível observar a classe na listagem de código 5.8.

```
1  @GrpcService
2  public class SandwichGrpcServiceImpl extends SandwichServiceGrpc.
       SandwichServiceImplBase {
3
4      @Autowired
5      private SandwichService service;
6
7      @Override
8      public void getSandwich(SandwichRequest request,
           StreamObserver<SandwichResponse> responseObserver) {
```

```
9           boolean sandwichExistence= service.sandwichExistence(UUID.
                fromString(request.getSandwichId()));
10
11          SandwichResponse response;
12          if(sandwichExistence){
13              response = SandwichResponse.newBuilder()
14                      .setCode(200)
15                      .build();
16          }else{
17              response = SandwichResponse.newBuilder()
18                      .setCode(404)
19                      .build();
20          }
21          responseObserver.onNext(response);
22          responseObserver.onCompleted();
23      }
24  }
```

Listagem 5.8: Classe SandwichGrpcServiceImpl

Mais uma vez a classe estende uma outra classe que foi gerada pelo *plugin* a partir do serviço definido no ficheiro ".proto". A anotação "*@GrpcService*" marca a classe como um serviço gRPC e é usada apenas no lado do servidor. Ao receber o pedido do tipo "*SandwichRequest*" do cliente, começou-se por verificar se a sanduíche existia a partir do método "*sandwichExistence*" da camada "*Service*". Após isso, construiu-se um objeto do tipo "*SandwichResponse*" em que caso a sanduíche existisse, o código enviado seria o 200, caso contrário o código enviado seria o 404. Esta abordagem surgiu naturalmente devido à familiarização com os códigos HTTP, no entanto, é importante destacar que o gRPC suporta *booleans* sem qualquer problema e poderia ser uma abordagem alternativa, neste caso. De seguida, é importante destacar que o *responseObserver* é o objeto responsável por enviar as repostas para o cliente. Utilizou-se o método "*onNext(response)*" para indicar ao *responseObserver* que o objeto "*response*" é a resposta do serviço gRPC, ou seja, é a resposta a enviar ao cliente. De seguida, o método "*onCompleted()*" foi usado para indicar ao cliente que não haverá mais repostas, concluindo assim a chamada ao serviço gRPC.

5.2.2 REST

Passando agora para a implementação em REST, a lógica é toda semelhante em termos de negócio, no entanto, o serviço das reservas faz um pedido HTTP diretamente ao serviço das sanduíches através de um *endpoint* no lado do serviço das sanduíches.

Para começar, na camada *Service* há uma diferença na linha 14 (o interior do ciclo *for*) alterando-se para a listagem de código 5.9

```
for(int i=0; i<reservationItemList.size();i++){
    boolean sandwichExistence = helper.doesSandwichExist(
        createReservationDTO.getItemList().get(i).getSandwichId());
    if(!sandwichExistence){
        throw new ResponseStatusException(HttpStatus.NOT_FOUND,"
            That sandwich does not exist: "+createReservationDTO.
            getItemList().get(i).getSandwichId());
    }
}
```

Listagem 5.9: Diferença no ciclo *for* para REST

Como se pode ver foi chamado um método *doesSandwichExist* do objeto *helper* que é do tipo *HttpRequestHelper* e o seu conteúdo pode ser observado na listagem 5.10. O resultado do *endpoint* é o resultado do método (que é um pedido HTTP ao serviço das sanduíches).

```
public boolean doesSandwichExist(UUID sandwichId) throws
    IOException, InterruptedException {
    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
            .GET()
            .uri(URI.create("http://localhost:8081/sandwich/
                existence/"+ sandwichId ))
            .build();
    HttpResponse response = client.send(request, HttpResponse.
        BodyHandlers.ofString());

    String body = response.body().toString();
    boolean result = false;
    if (body.equals("true")){
        result=true;
    }
    else if(body.equals("false")){
        result=false;
    }
    return result;
}
```

Listagem 5.10: HTTP *Request Helper* em REST

5.2.3 Serviço de Autorização e Autenticação

Para fazer grande parte dos pedidos é preciso ter determinado cargo no negócio, seja utilizador registado ou administrador. Para identificar o utilizador que está a fazer

determinado pedido é necessário enviar um "*Bearer token*" junto com o mesmo num *header* de *Authorization*. Esse *token* é gerado pelo serviço de autorização ao fazer *login* com as credenciais do utilizador. Como já foi dito anteriormente no contexto da aplicação, para o protótipo não é possível fazer registo de novos utilizador pelo que são criados a partir de *bootstrap* para desenvolvimento. Nesta subsecção irão ser analisadas as partes fundamentais para perceber a implementação do serviço de autorização.

Em primeiro lugar são necessárias 3 dependências no ficheiro *pom.xml* e estas podem ser consultadas na listagem 5.11.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</
        artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.1</version>
</dependency>
```

Listagem 5.11: Dependências para autorização

As informações do utilizador são encriptadas através de um JWT (*JSON Web Token*) *token*. Todas as informações que o *token* tem estão na listagem 5.12.

```
public String generateJwtToken(Authentication authentication) {

    JwtUserDetails userPrincipal = (JwtUserDetails) authentication
        .getPrincipal();

    return Jwts.builder()
            .setId(String.valueOf((userPrincipal.getId())))
            .setSubject(userPrincipal.getAuthorities().toString())
            .setIssuedAt(new Date())
            .setExpiration(new Date((new Date()).getTime() +
                jwtExpirationMs))
            .signWith(SignatureAlgorithm.HS512, jwtSecret)
            .compact();
}
```

Listagem 5.12: *Builder* do *token* JWT

De forma a limitar os recursos que cada cargo pode aceder, foi feita a configuração da listagem 5.13 para versões do *Java Spring Boot* inferiores a 3.0. Esta última parte também é necessária noutros serviços para que efetivamente os *endpoints* possam ser limitados a quem é suposto.

```
@Override // from WebSecurityConfigurerAdapter
protected void configure(HttpSecurity httpSecurity) throws
    Exception {
    httpSecurity.cors().and().csrf().disable()
            .exceptionHandling().authenticationEntryPoint(
                unauthorizedHandler).and()
            .sessionManagement().sessionCreationPolicy(
                SessionCreationPolicy.STATELESS).and()
            .authorizeRequests().antMatchers("/authenticate","/
                swagger-ui/**","/v3/api-docs/**","/h2/**").
                permitAll()
            .anyRequest().permitAll();

    httpSecurity.headers().frameOptions().disable();
    httpSecurity.addFilterBefore(authenticationJwtTokenFilter(),
        UsernamePasswordAuthenticationFilter.class);
}
```

Listagem 5.13: Configuração de *endpoints* em *Spring Boot* 2+

Os serviços que estão implementados com gRPC como meio de comunicação entre eles utilizam a versão de configuração acima. Já os serviços em REST, cuja versão do *Spring Boot* é superior a 3.0 utilizam a versão de configuração da listagem

5.14. A lógica é semelhante, no entanto, a classe "*WebSecurityConfigurerAdapter*" está *deprecated* na versão 3.0+ e a sintaxe está diferente.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http)
    throws Exception {
        http
            .csrf().disable()
            .exceptionHandling().authenticationEntryPoint(
                unauthorizedHandler).and()
            .sessionManagement().sessionCreationPolicy(
                SessionCreationPolicy.STATELESS).and()
            .authorizeHttpRequests().requestMatchers("/
                authenticate").permitAll()
            .anyRequest().permitAll()
            .and().headers().frameOptions().disable()
            .and().addFilterBefore(authenticationJwtTokenFilter(),
                UsernamePasswordAuthenticationFilter.class);

        return http.build();
}
```

Listagem 5.14: Configuração de *endpoints* em *Spring Boot* 3+

5.2.4 Serviço de Categorias, Ingredientes e Sanduíches

Esta secção tem como objetivo mostrar alguns detalhes da implementação dos serviços de categorias, ingredientes e sanduíches. Estes serviços estão agrupados porque tem semelhanças no que toca à lógica de criação e apenas se irá expor um caso, criação de sanduíches, realçando as diferenças com os outros.

Na criação de sanduíches (feita por um administrador), começou-se por eliminar ingredientes repetidos. De seguida, verificou-se que todos os ingredientes da lista existiam e que o nome da sanduíche não era repetido. Após isso criou-se um objeto do tipo "*Sandwich*" através do construtor da classe de forma a impor as regras do negócio (através dos *setters*) como, por exemplo a lista de ingredientes ser composta por pelo menos 2 ingredientes.

```
public Sandwich createSandwich(CreateSandwichDTO sandwichDTO) {
    List<IngredientKeyDTO> listOfIngredientsDTO1= sandwichDTO.
        getListOfIngredients();
    Set<IngredientKeyDTO> setSemDuplicados = new LinkedHashSet<>(
        listOfIngredientsDTO1);
    List<IngredientKeyDTO> listOfIngredientsDTO = new ArrayList<>(
        setSemDuplicados);
```

```
5
6       List<Ingredient> listOfIngredients = new ArrayList<>();
7
8       for(int i=0; i<listOfIngredientsDTO.size();i++){
9           IngredientResponse response = ingredientGrpcService.
                getIngredient(listOfIngredientsDTO.get(i).getPrivateKey
                ().toString());
10          if(response.getCode() == 404){
11              throw new ResponseStatusException(HttpStatus.NOT_FOUND
                    ,"That ingredient does not exist: "+
                    listOfIngredients.get(i).getName());
12          }
13
14          Ingredient ingredient = new Ingredient(response.getName(),
                 response.getCategory());
15          listOfIngredients.add(ingredient);
16      }
17
18      if(repository.getSandwichByDesignation(sandwichDTO.
            getDesignation())!=null){
19          throw new ResponseStatusException(HttpStatus.CONFLICT, "
                That name is already used");
20      }
21      Sandwich sandwich = new Sandwich(sandwichDTO.getSandwichId(),
            sandwichDTO.getPublicKey(),sandwichDTO.getDesignation(),
            sandwichDTO.getDescription(),listOfIngredients);
22      return repository.save(sandwich);
23  }
```

Listagem 5.15: Método para a criação de sanduíche na camada *Service*

Nos ingredientes é em tudo similar, no entanto, nestes foi utilizado um *Normalizer*, visível na listagem 5.16, para retirar a acentuação ao verificar o nome dos mesmos, isto acontece porque o nome dos ingredientes não tem uma componente criativa, devendo ser objetivos.

```
1 String name1= Normalizer.normalize(createIngredientDTO.getName(),
      Normalizer.Form.NFD).replaceAll("\\p{Mn}", "");
```

Listagem 5.16: Normalizer para nome de ingredientes

5.2.5 Serviço das Avaliações, Denúncias e Votos

Esta secção tem como objetivo mostrar alguns detalhes da implementação dos serviços de avaliações, denúncias e votos. Estes serviços estão agrupados porque as

denúncias e os votos são estritamente sobre as avaliações. A criação de uma denúncia é semelhante aos outros em termos de lógica (verifica se a avaliação existe para possibilitar a criação da denúncia), não tendo muita complexidade no que toca a restrições já que é composta apenas do texto que justifica a denúncia logo não vai ser exposta neste documento. Para a criação de avaliações a lógica também é semelhante, no entanto, em avaliações ocorre a deteção de língua do texto da mesma, a obtenção de um facto curioso e o controlo de vocabulário (havendo determinadas palavras que são proibidas). Já em votos, até à criação é tudo similar, no entanto, depois da criação, o serviço dos votos envia um pedido ao serviço das avaliações para este aumentar o *upVote* ou *downVote* da avaliação correspondente.

Deteção de Língua do Serviço das Avaliações

No serviço das avaliações, o atributo "*language*" é obtido automaticamente através da deteção da língua em que o texto da avaliação está. A língua é "atribuída" na criação da avaliação, como foi dito anteriormente.

Com auxílio da dependência da listagem 5.17, é possível aceder aos métodos necessários para efetivamente fazer a deteção de língua.

```
1 <dependency>
2     <groupId>com.github.pemistahl</groupId>
3     <artifactId>lingua</artifactId>
4     <version>1.2.2</version>
5 </dependency>
```

Listagem 5.17: Dependência para deteção de língua

Na primeira linha da listagem 5.18, é criado um detetor de línguas (capaz de detetar as línguas que lhe são passadas: inglês, espanhol, português, francês e alemão) a partir de classes e métodos provenientes da dependência acima. O detetor é usado na linha três para identificar a língua em que o texto da avaliação está.

```
1 final LanguageDetector detector = LanguageDetectorBuilder.
      fromLanguages(ENGLISH, FRENCH, GERMAN, SPANISH, PORTUGUESE).
      build();
2
3 review.language=detector.detectLanguageOf(rev.text).toString();
```

Listagem 5.18: Deteção de língua

Posteriormente, foi feito um *endpoint* para ser possível pedir todas as avaliações com uma determinada língua. Na listagem 5.19, é possível ver um método da camada

Repository que faz a seleção de avaliações com uma determinada língua. Neste *endpoint*, não há nada complexo entre as camadas *Controller* e *Service*. A camada *Repository* é responsável por estabelecer a comunicação efetiva com a base de dados e os seus métodos possuem a anotação *@Query* uma vez que utilizam explicitamente a *query* SQL (*Structured Query Language*) na anotação.

```
1 @Query("SELECT r FROM Review r WHERE r.language = UPPER(:language)
      ")
2 List<Review> getReviewByLanguage(@Param("language") String
      language);
```

Listagem 5.19: Método da camada *Repository*

5.3 Testes e Experiências de avaliação

Foram realizados três tipos de teste: testes unitários às classes de *domain*, testes de integração utilizando *Postman* e testes de desempenho utilizando a ferramenta *Jmeter*. Esta secção tem como objetivo mostrar detalhes dos mesmos. Começou-se por realizar os testes unitários (113 para cada protótipo) e os testes de integração (641). Cada protótipo foi submetido aos mesmos testes para averiguar que estava tudo em funcionamento e que estavam em pé de igualdade. Após isso, foram submetidos aos testes de desempenho.

No fim, foi feito um teste de hipóteses para averiguar se existe uma diferença significativa estatisticamente. Para isto, foi utilizada a linguagem R e a ferramenta *RStudio*.

5.3.1 Testes Unitários

Os testes unitários consistem na verificação individual e isolada de fragmentos de código, como métodos. Nas listagens 5.20 e 5.21, é possível ver um exemplo de teste unitário de insucesso e sucesso, respetivamente.

```
1  @Test
2  void ensurePickupDateIsNotBeforeCreationDate() throws
       ParseException {
3      Reservation reservation = new Reservation();
4      long millis = System.currentTimeMillis();
5      Date date = new Date(millis);
6      reservation.setCreationDate(date);
7
8      String pickupDateString = "2022-06-03T00:00:00.000+00:00";
9      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
           'T'HH:mm:ss.SSSXXX");
10     Date pickupDate = dateFormat.parse(pickupDateString);
11
12     Throwable exception = assertThrows(IllegalArgumentException.
           class, () ->reservation.setPickupDate(pickupDate));
13     assertEquals("Pickup date cannot be before creation date",
           exception.getMessage());
14 }
```

Listagem 5.20: Exemplo de teste unitário de insucesso

```
1  @Test
2  void ensurePickupDateIsAccepted() throws ParseException {
3      Reservation reservation = new Reservation();
4      long millis = System.currentTimeMillis();
5      Date date = new Date(millis);
6      reservation.setCreationDate(date);
7
8      String pickupDateString = "2024-06-03T00:00:00.000+00:00";
9      SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd
           'T'HH:mm:ss.SSSXXX");
10     Date pickupDate = dateFormat.parse(pickupDateString);
11     reservation.setPickupDate(pickupDate);
12     assertEquals("Mon Jun 03 01:00:00 WEST 2024", reservation.
           getPickupDate().toString());
13 }
```

Listagem 5.21: Exemplo de teste unitário de sucesso

No total foram realizados 113 testes unitários às classes *domain* (em cada protótipo) assegurando uma cobertura de métodos superior a 80% (Figura 5.3).

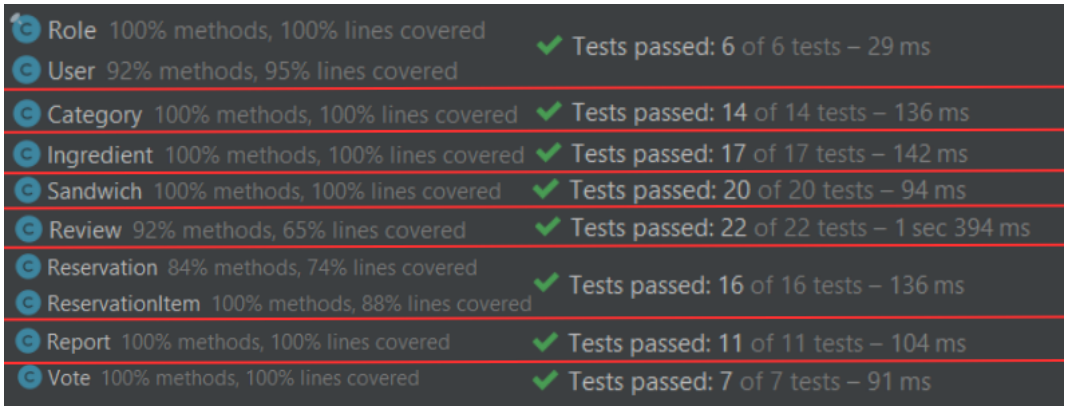


Figura 5.3: Cobertura e quantidade de testes unitários

5.3.2 Testes de Integração

Os testes de integração são usados para verificar se os diferentes componentes de um sistema comunicam e funcionam corretamente entre si. Para a realização dos testes de integração, foi usada a ferramenta *Postman*. Com esta ferramenta também é possível validar a funcionalidade da API, podendo incluir a verificação de autenticação. Por exemplo, ao criar uma reserva foi necessário confirmar que a mesma foi guardada e ficou com os devidos atributos. Foi possível fazer essa confirmação fazendo um novo pedido, desta vez um "*GET*" que possuísse testes para validar a criação da reserva. Na listagem 5.22, é possível ver exemplos para a validação do pedido.

```
1  pm.test("Reservation Found", function () {
2      pm.response.to.have.status(200);
3  });
4  pm.test("Checking Reservation Id", function () {
5      var jsonData = pm.response.json();
6      pm.expect(jsonData.reservationId).to.eql(pm.globals.get("
           reservationId"));
7  });
8  pm.test("Checking Reservation status", function () {
9      var jsonData = pm.response.json();
10     pm.expect(jsonData.status).to.eql("ACTIVE");
11 });
12 pm.test("Checking List Of Items", function () {
13     var jsonData = pm.response.json();
14     pm.expect(jsonData.items.length).to.eql(1);
15 });
```

Listagem 5.22: Exemplo de testes para validar a API

Os testes de integração envolvem a verificação do comportamento e do funcionamento dos componentes quando eles são integrados. Portanto, ao validar a criação da reserva e fazendo a confirmação dos dados num outro pedido usando os testes fornecidos, estamos a realizar testes de integração. De forma a garantir o bom funcionamento de todos os componentes do projeto, foram feitos um total de 641 testes de integração, como se pode ver na figura 5.4.

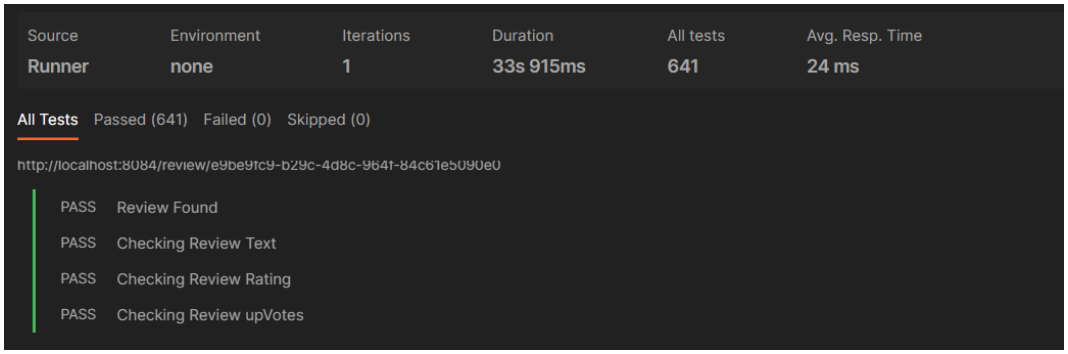


Figura 5.4: Quantidade de testes de integração

5.3.3 Testes de Desempenho

A avaliação do desempenho da criação de reservas em cada prototipo foi realizado recorrendo à ferramenta *JMeter*. *JMeter* é uma aplicação concebida para testar e examinar o desempenho e comportamento funcional de aplicações cliente/servidor. *JMeter* atua como cliente e mede, por exemplo, os tempos de resposta.

Neste projeto, o *JMeter* recolheu os seguintes tipos de dados [32] :

- ***Elapsed Time:*** Tempo decorrido desde o momento imediatamente anterior ao envio do pedido até o momento imediatamente após a última resposta ser recebida
- ***Latency:*** Tempo decorrido desde o momento imediatamente anterior ao envio do pedido até o momento imediatamente após a primeira resposta ser recebida. O tempo inclui todo o processamento necessário para a construção do pedido, bem como a construção da primeira parte da resposta
- ***Average:*** média de tempo decorrido para a conclusão de cada amostra
- ***Min:*** valor mais rápido para a conclusão de uma amostra
- ***Max:*** valor mais lento para a conclusão de uma amostra
- ***Standard Deviation:*** medida estatística padrão que indica o quão longe a população avaliada se encontra da mediana

- ***Throughput:*** medida calculada através da divisão do número de pedidos pelo tempo total. O tempo considerado vai desde o início da primeira amostra até ao fim da última amostra
- ***Received/Sent KB/Sec:*** taxa de transferência em *Kilobytes*/segundo (recebido e enviado)

Configuração da Carga

Em *JMeter*, a configuração da carga de testes envolveu:

- **Número de *threads* (utilizadores):** este número representa a quantidade de utilizadores virtuais que se espera que se conectem ao servidor em simultâneo - o número de *threads* que realiza pedidos ao servidor em paralelo
- **Período de *ramp-up*:** tempo (em segundos) que indica o período que o *JMeter* deve demorar a inicializar o número de threads definido. Por exemplo, com um número de threads igual a 100, e um período de *ramp-up* igual a 10, o *JMeter* inicializará 10 *threads* a cada segundo
- ***Loop Count***: número de vezes que o caso de teste é iterado

Foi estabelecida a configuração apresentada na tabela 5.1, resultando num total de 3000 execuções.

Tabela 5.1: Configuração do *JMeter*

Configuração	**Valores**
Número de *threads*	1000
Período de *ramp-up*	10
Loop Count	3

Teste Executado

Em primeiro lugar, são definidos os valores da tabela acima na secção de "*Thread Group*". De seguida, é definido o *IP*, porta e *path*, assim como o *body* para realizar o pedido na parte de "*Create Reservation*". Em seguida, na secção "*HTTP Header Manager*" são definidos alguns *headers* para fazer o pedido com sucesso, por exemplo o campo de *Authorization* é preenchido com o *token* de autorização e o tipo de conteúdo do *body* inserido na secção de "*Create Reservation*".

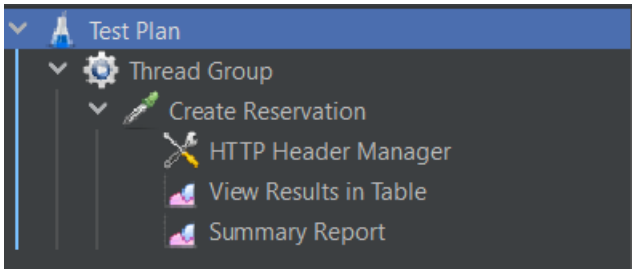


Figura 5.5: Teste para a criação de uma reserva em *JMeter*

Análise dos Resultados

As tabelas 5.2 e 5.3 apresentam os resultados obtidos.

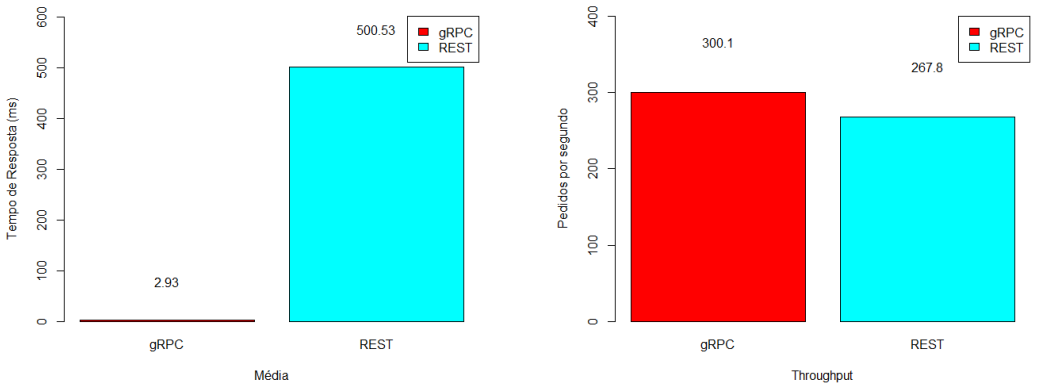
Tabela 5.2: Resultados da implementação em gRPC

Pedidos	**Execuções**	**Mediana**	**Max(ms)**	**Média(ms)**	**Throughput**
Criação de uma reserva	3000	3	13	2	300.1/s
Requisitar catálogo de sanduíches	3000	1	9	1.47	299.7/s

Tabela 5.3: Resultados da implementação em REST

Pedidos	**Execuções**	**Mediana**	**Max(ms)**	**Média(ms)**	**Throughput**
Criação de uma reserva	3000	226	9581	500	267.8/s
Requisitar catálogo de sanduíches	3000	1	28	1.56	299.7/s

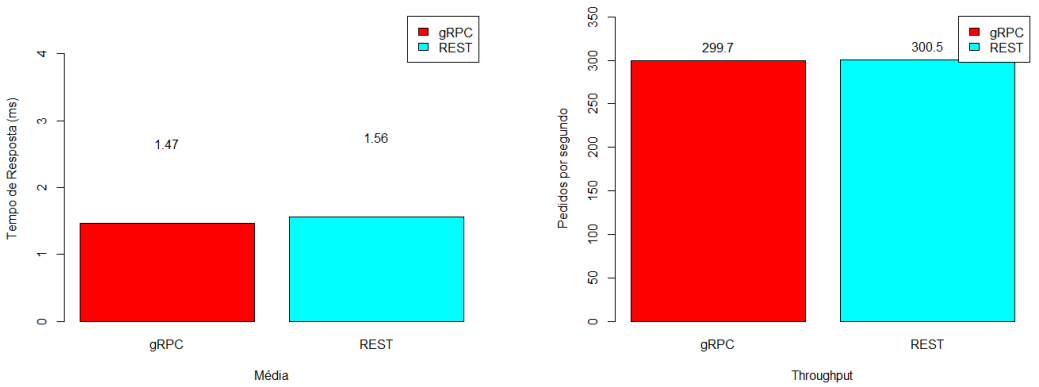
Nas figuras 5.6a e 5.6b, é possível ver comparações diretas entre as duas implementações. Olhando para a comparação entre a média de tempos de resposta (Figura 5.6a), é possível observar que o protótipo que adota gRPC apresenta um tempo de resposta inferior ao protótipo que adota REST o que faz com que, naturalmente, em gRPC acabe por apresentar um *throughput* superior.



(a) Comparação do tempo médio de resposta (b) Comparação do *Throughput*

Figura 5.6: Comparação de tempo médio de resposta e *throughput*

Na figuras 5.7a e 5.7b, não é possível ver diferenças significativas graficamente. Isto acontece porque nestes pedidos não existe comunicação com outros serviços, ou seja, não há qualquer influência do gRPC.



(a) Comparação do tempo médio de resposta

(b) Comparação do *Throughput*

Figura 5.7: Comparação de tempo médio de resposta e *throughput* do pedido GET

Análise Estatística

Primeiramente, foi necessário fazer o tratamento dos dados para que estes pudessem ser submetidos aos devidos testes. De seguida, foi necessário averiguar se a amostra seguia uma distribuição normal. Para isso, foi realizado o teste de *Lilliefors* para as quatro amostras, uma vez que estas possuíam 3000 observações cada (quando número de observações é menor que 30 utiliza-se o teste de *shapiro*). O passo seguinte foi verificar a simetria da distribuição. Finalmente, foi realizado um teste de hipóteses para verificar se a diferença dos tempos médios entre os dois protótipos é estatisticamente significativa. Durante todo o processo, foi definido o nível de significância como 0.05.

```
lillie.test(ConjuntoaverageTimeGetGrpc)
lillie.test(ConjuntoaverageTimePostGrpc)
lillie.test(ConjuntoaverageTimeGetRest)
lillie.test(ConjuntoaverageTimePostRest)

skewness(ConjuntoaverageTimeGetGrpc)
skewness(ConjuntoaverageTimePostGrpc)
skewness(ConjuntoaverageTimeGetRest)
skewness(ConjuntoaverageTimePostRest)

valor_p <- wilcox.test(ConjuntoaverageTimePostGrpc,
    ConjuntoaverageTimePostRest, paired=FALSE)$p.value
valor_p

valor_p <- wilcox.test(ConjuntoaverageTimeGetGrpc,
    ConjuntoaverageTimeGetRest, paired=FALSE)$p.value
valor_p
```

Listagem 5.23: Código do teste de normalidade, simetria e hipóteses

Ao realizar os testes de normalidade, o p-value foi menor que nível de significância em todos os conjuntos, logo há evidências estatísticas de que a distribuição não segue uma distribuição normal. Os resultados dos testes de assimetria foram todos maiores que 1 logo, a distribuição é assimétrica. Posto isto, optou-se pelo teste de *Wilcoxon* para realizar o seguinte teste de hipóteses para comparação entre os protótipos:

- Hipótese nula (H0): Não há diferença significativa no desempenho dos dois protótipos
- Hipótese alternativa (H1): Há diferença significativa no desempenho dos dois protótipos

Na comparação dos pedidos de criação de uma reserva (há comunicação entre serviços), o p-value foi 0 (menor que 0.05), ou seja, temos evidências estatísticas de que há diferença significativa no desempenho dos dois protótipos (Rejeitando-se a Hipótese nula).

Na comparação dos pedidos para requisitar o catálogo de sanduíches (não há comunicação entre serviços), o p-value é maior que 0.05 então, temos evidências estatísticas de que não há diferença significativa no desempenho dos dois protótipos (Não se rejeita a Hipótese nula).

Capítulo 6

Conclusões

Este capítulo tem como objetivo expor os objetivos concretizados, algumas das dificuldades durante a realização do projeto bem como em algumas sugestões para possíveis pontos num desenvolvimento futuro. No final são dadas algumas apreciações pessoais.

6.1 Objetivos Concretizados

Durante todo o projeto, os objetivos propostos como, a exploração de gRPC como meio de comunicação entre serviços, a sua divulgação e a comparação com REST foram alcançados e com isso, foi adquirido um vasto conhecimento relativo ao protocolo gRPC e outras ferramentas subjacentes como, por exemplo, o método de serialização *Protocol Buffers*. Para isso, foram realizados com sucesso dois protótipos, um recorrendo ao famoso REST, e outro recorrendo ao, em crescimento, gRPC.

Foram feitos uma série de testes unitários e de integração, para garantir que ambos os protótipos estavam com todas as funcionalidades pretendidas e em pé de igualdade para realizar os testes de desempenho.

Relativamente aos resultados obtidos, ficou provado estatisticamente que o protótipo que utiliza gRPC em pedidos que necessitem de comunicar com outros serviços, apresenta desempenho superior ao protótipo que adota REST como meio de comunicação entre serviços.

Todo o trabalho ficou disponível num repositório público, em GitHub, com a minha autoria, como contributo para divulgação de gRPC e da sua utilização. Este repositório está disponível em [33].

6.2 Dificuldades

As grandes dificuldades em desenvolver serviços utilizando gRPC devem-se em grande parte à configuração inicial como as dependências, as versões, a compreensão do *plugin* para a geração de código e, claro, há a necessidade de compreender e utilizar a linguagem de *protocol buffers* de modo a estruturar as mensagens, os serviços e os métodos RPC que se pretende disponibilizar para a aplicação. Após estes primeiros passos e após desenvolver uma metodologia para o desenvolvimento de serviços utilizando gRPC, estes ficam tão ou mais fáceis de implementar comparando com REST.

6.3 Trabalho Futuro

Na sequência do trabalho desenvolvido, é importante relembrar que o objetivo do trabalho era explorar gRPC e as suas potenciais vantagens e desvantagens, para isso foi desenvolvido um protótipo. O objetivo nunca foi ter uma aplicação totalmente funcional pelo que esta ainda pode ser mais desenvolvida.

Relativamente a gRPC, sugere-se que o próximo explore a capacidade de o gRPC ser assíncrono para que um pedido possa ficar à espera um certo tempo por um determinado serviço antes de lançar um erro que simboliza a falta de conexão com o mesmo. Para além disso, neste trabalho as mensagens enviadas via gRPC eram simples pelo que o *streaming* de dados também é um campo merecedor de devida análise e testagem. Por fim, gRPC tem a capacidade de funcionar em várias línguas e plataformas pelo que, também é um ramo que pode ser explorado.

Relativamente ao prótotipo, pode ser feito um desenvolvimento mais aprofundado do serviço dos utilizadores, permitindo que estes se possam registar, alterar o perfil, adicionar mais atributos como, por exemplo, uma foto de perfil. Seria possível também, adicionar fotos de ingredientes e sanduíches, tal como foto dos ingredientes na lista de ingredientes das sanduíches. Para além disso, pode-se implementar um melhor controlo de avaliações e votos, para que, por exemplo, quando um moderador apaga um avaliação que não tem nada a ver com a sanduíche, caso esta tenha *upVotes* ou *downVotes* estes sejam apagados do serviço dos votos automaticamente. Em adição, pode ser desenvolvido uma API *gateway* que atue como ponto de entrada único para as funcionalidades de todos os serviços e que encaminhe o pedido ao serviço responsável.

6.4 Apreciações Finais

No geral, sinto-me satisfeito com o trabalho realizado, no qual explorei gRPC como meio de comunicação entre serviços e sinto-me orgulhoso por ter aplicado muitos dos conhecimentos adquiridos ao longo do curso neste projeto e por ter contribuído para a divulgação desta inovadora tecnologia.

Esta experiência proporcionou-me uma compreensão teórica e prática sobre as vantagens de gRPC mas também sobre as tecnologias subjacentes como, *Protocol Buffers* e HTTP/2 e como estas tecnologias podem impactar o desempenho e a eficiência das comunicações entre serviços. Além disso, a comparação de desempenho entre gRPC e REST permitiu-me avaliar mais concretamente a diferença entre os dois protocolos.

Referências

[1] Google, "About grpc." Available at https://grpc.io/about/, Apr. 2005. (Último acesso em 12/05/2023). [Citado na página 1]

[2] Microsoft, "Xml para principiantes." Available at https://support.microsoft.com/pt-pt/office/xml-para-principiantes-a87d234d-4c2e-4409-9cbc-45e4eb857d44. (Último acesso em 23/04/2023). [Citado na página 5]

[3] W3Schools, "Introduction to xml." Available at https://www.w3schools.com/xml/xml_whatis.asp. (Último acesso em 23/04/2023). [Citado na página 5]

[4] D. Crockford, "Introducing json." Available at https://www.json.org/json-en.html, 2002. (Último acesso em 23/04/2023). [Citado na página 6]

[5] Google, "Flatbuffers." Available at https://flatbuffers.dev/. (Último acesso em 24/04/2023). [Citado na página 8]

[6] kiran kumar, "Protobuf vs json vs flatbuffers." Available at https://kiranjobmailid.medium.com/protobuf-vs-json-b2e9bc460986, June 2020. (Último acesso em 24/04/2023). [Citado na página 8]

[7] D. Juhász, "Flatbuffers vs protobufs - how they are used in java." Available at https://www.netguru.com/blog/flatbuffers-vs-protobufs, Dec. 2022. (Último acesso em 24/04/2023). [Citado na página 8]

[8] A. S. Foundation, "Apache thrift." Available at https://thrift.apache.org/. (Último acesso em 24/04/2023). [Citado na página 9]

[9] D. Gupta, "Thrift: The missing guide." Available at https://diwakergupta.github.io/thrift-missing-guide/, Nov. 2015. (Último acesso em 24/04/2023). [Citado na página 9]

[10] J. Lewis and M. Fowler, "Microservices." Available at https://martinfowler.com/articles/microservices.html, Mar. 2014. (Último acesso em 31/03/2023). [Citado na página 10]

[11] A. DevOps and martinekuan, "Microservices architecture design." Available at https://learn.microsoft.com/en-us/azure/architecture/

microservices/, Sept. 2022. (Último acesso em 31/03/2023). [Citado na página 10]

[12] M. W. Docs, "Http." Available at https://developer.mozilla.org/en-US/docs/Web/HTTP, Mar. 2023. (Último acesso em 31/03/2023). [Citado nas páginas 11 e 12]

[13] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): Message syntax and routing." Available at https://www.rfc-editor.org/rfc/rfc7230#page-5, June 2014. (Último acesso em 31/03/2023). [Citado na página 11]

[14] R. P. M. Belshe and M. Thomson, "Hypertext transfer protocol version 2 (http/2)." Available at https://www.rfc-editor.org/rfc/rfc7540, May 2015. (Último acesso em 01/04/2023). [Citado na página 12]

[15] I. Grigorik and Surma, "Introduction to http/2." Available at https://web.dev/performance-http2/, Sept. 2016. (Último acesso em 01/04/2023). [Citado na página 12]

[16] I. Grigorik, "High performance browser networking." Available at https://hpbn.co/, 2013. (Último acesso em 01/04/2023). [Citado nas páginas ix e 13]

[17] Aurora, "Http version comparison." Available at https://blog.caoyu.info/http-version.html, 2020. (Último acesso em 01/04/2023). [Citado nas páginas ix e 14]

[18] L. Rosencrance and B. Matturro, "Remote procedure call (rpc)." Available at https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC, Oct. 2021. (Último acesso em 01/04/2023). [Citado na página 14]

[19] Miraj, "Communication." Available at https://www.slideshare.net/bavachkarsg/rpc-64189031, July 2016. (Último acesso em 01/04/2023). [Citado nas páginas ix e 14]

[20] A. Stec, "The rest architecture." Available at https://www.baeldung.com/cs/rest-architecture. (Último acesso em 02/04/2023). [Citado na página 15]

[21] L. Gupta, "What is rest." Available at https://restfulapi.net/, 2018. (Último acesso em 02/04/2023). [Citado na página 15]

[22] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture." Available at https://www.ics.uci.edu/~fielding/pubs/webarch_icse2000.pdf, 2000. (Último acesso em 02/04/2023). [Citado na página 15]

[23] L. R. (Google), "grpc motivation and design principles." Available at `https://grpc.io/blog/principles/`, Sept. 2015. (Último acesso em 02/04/2023). [Citado na página 16]

[24] Google, "Core concepts, architecture and lifecycle." Available at `https://grpc.io/docs/what-is-grpc/core-concepts/`, 2015. (Último acesso em 02/04/2023). [Citado nas páginas 16, 17 e 18]

[25] K. I. D. Kuruppu, *gRPC: Up Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020. [Citado nas páginas ix, 17 e 18]

[26] S. Sarker, "Up and running with grpc." Available at `https://dev.to/mesadhan/up-and-running-with-grpc-4e6p`, Jan. 2022. (Último acesso em 02/04/2023). [Citado nas páginas ix e 19]

[27] I. 25000, "Iso/iec 25010." Available at `https://iso25000.com/index.php/en/iso-25000-standards/iso-25010`. (Último acesso em 11/06/2023). [Citado na página 30]

[28] S. Brown, "The c4 model for software architecture." Available at `https://c4model.com/`. (Último acesso em 11/06/2023). [Citado na página 31]

[29] P. Kruchten, "The 4+1 view model of architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–50, 1995. [Citado na página 31]

[30] A. Stec, "Database design in a microservices architecture." Available at `https://www.baeldung.com/cs/microservices-db-design`. (Último acesso em 11/06/2023). [Citado na página 41]

[31] M. Zhang, "grpc-spring-boot-starter documentation." Available at `https://yidongnan.github.io/grpc-spring-boot-starter/en/versions.html`. (Último acesso em 16/06/2023). [Citado na página 43]

[32] A. S. Foundation, "Glossary." Available at `https://jmeter.apache.org/usermanual/glossary.html`. (Último acesso em 16/06/2023). [Citado na página 61]

[33] J. Vieira, "grpc for service-to-service communication." Available at `https://github.com/joaovieira17/joaov_grpc_communication`, June 2023. (Último acesso em 26/06/2023). [Citado na página 67]