INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA

isep

Criação e evolução de uma API pública

PEDRO JORGE OLIVEIRA LOPES
Julho de 2018

POLITÉCNICO
DO PORTO

Criação e evolução de uma *API* pública

Pedro Jorge Oliveira Lopes

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Gráficos e Multimédia

Orientadora: Isabel Azevedo

Supervisor: Ivo Pereira

Júri:

Presidente:

Vogais:

Porto, Junho 2018

Resumo

A E-goi é uma empresa que providencia uma plataforma de marketing digital para clientes com diferentes necessidades e preferências. A *API* pública que a E-goi oferece a esses clientes possui alguns problemas. Este projeto consistiu na criação de uma nova *API* pública para a E-goi com a finalidade de substituir a anterior. A *API* necessitou de suportar o estilo arquitetural *REST* e protocolo *SOAP* corretamente, para satisfazer as necessidades de diversos clientes. Precisou também de ter respostas compreensíveis, para que a sua utilização seja facilitada e os clientes se sintam satisfeitos. Este projeto contemplou ainda um sistema de versionamento por serviço, compatível com *REST* e *SOAP* que pode ser utilizado para aumentar a longevidade da *API*.

Perante as necessidades dos clientes e da E-goi, optou-se por uma solução que suporta *REST* e *SOAP* num único projeto. Para esse fim foi realizado um estudo sobre o funcionamento de ambos e sobre as *API*, incluindo controlo de tráfego de utilização e *caching*. Foi ainda realizado um estudo sobre a evolução das *API* e da anterior *API* pública da E-goi, que foi substituída por a desenvolvida neste projeto. Para se apresentar uma visão mais concreta do negócio e do seu valor, realizou-se uma análise de valor deste projeto.

A solução desenvolvida disponibiliza diversos serviços em *REST* e *SOAP,* um sistema de erros com os códigos *HTTP* apropriados, e possui um protótipo da utilização de um sistema de versionamento por serviço num dos recursos, para que, no futuro, se a empresa assim o decidir, essa funcionalidade possa ser disponibilizada.

Para validar que a solução final é adequada, foi realizado um questionário de satisfação, testes de aceitação aos serviços e ao sistema de versionamento. Os dados do tempo de resposta foram obtidos da *API* pública e da utilização direta da privada, e analisados através de um teste estatístico. Os resultados do questionário demostraram uma maior satisfação com a nova *API* do que com a anterior.

Palavras-chave: *API*; *REST*; *SOAP;* Versionamento*;*

Abstract

E-goi is a company that provides a digital marketing platform for clients with different needs and preferences. The public API that E-goi offers to those clients has some problems. This project consisted in the creation of a new public API with the purpose of replacing the previous one. The API needed to support REST architectural style and SOAP protocol correctly, to meet customer needs. It also required to have correctly structured responses, in order to ease its use. This project also contains a service versioning system, compatible with REST and SOAP which can increase the API longevity.

According to clients' and E-goi's needs, it was chosen a solution that supports both REST and SOAP in a single project. The approach which led to the final result involved a study of both and about API, including API traffic management and caching. It was also studied API evolution and reviewed the old E-goi public API, that was replaced with the one developed in this project. In order to give a more concrete vision of the business and its value, a value analysis was performed for this project.

The developed solution provides several services in REST and SOAP, an error system with the appropriate HTTP codes, and contains a prototype of the versioning system integrated in one of the resources, so that, in the future, if the company decides it, that functionality could be delivered.

To validate if the final solution meets the requirements, it was created a questionnaire of satisfaction, developed acceptance tests to the services and the versioning system. The response time data was obtained from the public API and the direct use of the private API, and analyzed in a statistical test. The questionnaire results demonstrated a greater satisfaction with the new API than with the previous.

Keywords: API; REST; SOAP; Versioning;

Dedicatória

Quero dedicar esta tese à minha família e amigos que sempre me apoiaram.

Agradecimentos

Quero agradecer à minha orientadora Isabel Azevedo, aos meus colegas de estágio Ricardo Santos, Tiago Santos, Duarte Coelho e Maria Almeida, aos colegas de empresa Álvaro Passos, João Ferreira, ao meu supervisor Ivo Pereira, ao Vítor Tavares, à E-goi pela oportunidade proporcionada e à minha família. Todos sempre me apoiaram durante este projeto.

Índice

Resumo ... iii

Abstract ... v

Dedicatória ... vii

Agradecimentos ... ix

Índice ... xi

Lista de Figuras ... xv

Lista de Tabelas ... xvii

Lista de Excertos de Código ... xix

Lista de Acrónimos e Siglas ... xxi

1 Introdução ... 1

1.1 Contexto ... 1

1.2 Problema ... 2

1.3 Objetivos ... 3

1.4 Abordagem e Processo de Desenvolvimento ... 5

1.5 Estrutura do Documento ... 5

2 Estado da Arte ... 7

2.1 API ... 7
2.1.1 Controlo de tráfego de utilização ... 9
2.1.2 Caching ... 10
2.1.3 Evolução das API ... 12
2.1.3.1 Versionamento ... 13
2.1.3.2 Técnicas de evolução ... 16

2.2 HTTP ... 18
2.2.1 Códigos HTTP e mensagens de erro ... 20

2.3 REST, SOAP e GraphQL ... 22
2.3.1 REST ... 22
2.3.2 SOAP ... 25
2.3.3 GraphQL ... 28
2.3.4 Comparação entre os paradigmas das API ... 30

2.4 Versão 2 da API a substituir 35

3 Análise de Valor 41

3.1 Processo de Negócio e Inovação 41

3.2 Proposta de Valor 46

3.3 Valor para o cliente 46

3.4 Modelo de Negócio Canvas 49

3.5 Cadeia de valor de Porter 52

3.6 Decisão utilizando o modelo AHP 54

4 Análise e Design arquitetural 61

4.1 Análise de negócio 61
4.1.1 Modelação do Domínio 61
4.1.2 Modelação do Processo de Negócio da Solução 63

4.2 Análise de Requisitos 66
4.2.1 Requisitos Funcionais 66
4.2.2 Outros requisitos 67

4.3 Design arquitetural 68
4.3.1 Componentes 68
4.3.1.1 Alternativas Arquiteturais 72
4.3.2 Estrutura das pastas 73
4.3.3 Implantação 76

5 Design e implementação 77

5.1 Design da solução 77
5.1.1 Sistema de versionamento 77
5.1.2 Disponibilização de *REST* e *SOAP* 80
5.1.2.1 Alternativa de Interligação com o sistema de versionamento 81
5.1.3 Sistema de erros 82
5.1.4 Seleção do formato da resposta 85

5.2 Implementação 87
5.2.1 Serviços implementados 87
5.2.2 Disponibilização de *REST* e *SOAP* 89
5.2.2.1 Disponibilização de REST e SOAP com o sistema de versionamento 90
5.2.3 Sistema de versionamento 92
5.2.3.1 Seleção da versão pretendida 92
5.2.3.2 Version Manager 93
5.2.4 Visão geral dos sistemas 95
5.2.5 Resultados dos testes de software 98
5.2.5.1 Testes de aceitação 98
5.2.5.2 Testes ao tempo de resposta dos serviços 100

6 Avaliação da Solução 103

6.1 Grandezas 103

6.2 Hipóteses 104
6.3 Metodologia 104
6.4 Resultados do questionário 106

7 Conclusão 111

7.1 Objetivos atingidos.. 111
7.2 Limitações e trabalho futuro 113

Referências 115

Anexo A - Questionário do modelo AHP 121

Anexo B - Questionário de satisfação 125

Anexo C - Suporte de *GraphQL*... 129

Lista de Figuras

Figura 1 - Estrutura geral do projeto 4
Figura 2 - Estrutura da cadeia de adaptadores [24] 17
Figura 3 - Grafo de versionamento de serviço [21] 18
Figura 4 - Níveis do modelo de maturidade de Richardson [3] 24
Figura 5 - Representação de uma mensagem *SOAP* [29] 26
Figura 6 - Utilização de *GraphQL* de forma híbrida com uma base de dados e integrando um sistema existente [35] 30
Figura 7 - Argumentos de entrada do serviço *addSubscriberBulk* da versão 2 da *API* 33
Figura 8 - Argumentos de resposta do serviço *addSubscriberBulk* da versão 2 da *API* 34
Figura 9 - Serviços do recurso *subscriber* da versão 2 da *API* 36
Figura 10 - Argumentos de entrada do serviço *addSubscriberBulk* da versão 2 da *API* 37
Figura 11 - Argumentos de resposta do serviço *addSubscriberBulk* da versão 2 da *API* 38
Figura 12 - Processo de inovação de um produto [40] 41
Figura 13 - O modelo NCD produto [40] 42
Figura 14 - Percentagem de utilização dos estilos arquiteturais das *API* registadas no site ProgrammableWeb [2] 44
Figura 15 - Modelo de negócio Canvas deste projeto 51
Figura 16 - Cadeia de valor de Porter da E-goi 53
Figura 17 - Esquema do modelo *AHP* 54
Figura 18 - Comparação de valor entre critérios 55
Figura 19 - Comparação entre as soluções para cada critério 58
Figura 20 - Modelo *AHP* 60
Figura 21 - Modelo de domínio do sistema E-goi 62
Figura 22 - Diagrama de processo 65
Figura 23 - Diagrama de componentes da solução 71
Figura 24 - Diagrama de componentes alternativo 1 72
Figura 25 - Diagrama de componentes alternativo 2 73
Figura 26 - Diagrama de pacotes 75
Figura 27 - Diagrama de implantação 76
Figura 28 - Diagrama de classes do sistema de versionamento 78
Figura 29 - Diagrama de classes da relação entre *REST* e *SOAP* com os serviços comuns 80
Figura 30 - Diagrama de classes da relação entre *REST* e *SOAP* com o sistema de versionamento 82
Figura 31 - Exceções personalizadas 83
Figura 32 - Modelos de erros 85
Figura 33 – Diagrama de classe do sistema de seleção do formato de resposta 86
Figura 34 - Lista de serviços da *API* 88
Figura 35 - Diagrama de sequência do sistema *REST* interligado aos serviços 89
Figura 36 - Diagrama de sequência do sistema *SOAP* interligado aos serviços 90

Figura 37 - Diagrama de sequência de interligação do sistema *REST* com o sistema de versionamento 91
Figura 38 - Diagrama de sequência de interligação do sistema *SOAP* com o sistema de versionamento 91
Figura 39 - Diagrama de sequência da utilização do sistema de versionamento em *REST* 95
Figura 40 - Diagrama de sequência da utilização do sistema de versionamento em *SOAP* 96
Figura 41 - Gráfico de barras dos resultados da pergunta sobre o tempo de resposta 106
Figura 42 - gráficos de barras das perguntas sobre a facilidade de aprendizagem 107
Figura 43 - gráficos de barras das perguntas sobre a facilidade de utilização.......................... 108
Figura 44 - Gráfico de barras dos resultados da pergunta sobre a quantidade de funcionalidades 108
Figura 45 - Gráfico de barras dos resultados - Gráfico de barras dos resultados da pergunta sobre a utilidade das funcionalidades........ 109
Figura 46 - Diagrama de componentes alternativo 3........ 129
Figura 47 - Diagrama de componentes alternativo 4........ 130

Lista de Tabelas

Tabela 1 - Comparação dos métodos *HTTP* 19
Tabela 2 – Códigos *HTTP* frequentemente utilizados 20
Tabela 3 - Comparação entre *REST* e *SOAP e GraphQL* 30
Tabela 4 - Proposta longitudinal de valor 48
Tabela 5 - Matriz de comparação de critérios 56
Tabela 6 - Tabela de pesos de cada critério 56
Tabela 7 - Índice de Consistência Aleatória [48] 57
Tabela 8 - Comparação das soluções com base no critério "Suporta *REST* e *SOAP*" 58
Tabela 9 - Comparação das soluções com base no critério "Escalabilidade" 58
Tabela 10 - Comparação das soluções com base no critério "Reaproveitamento de código" ... 59
Tabela 11 - Índices de consistência por critério 59
Tabela 12 - Glossário 63
Tabela 13 - Lista de requisitos funcionais 66
Tabela 14 - Comportamento do sistema de versionamento com base no valor do *header* 93
Tabela 15 - Hipóteses 104

Lista de Excertos de Código

Código 1 - Exemplo de uma mensagem de erro em *JSON* 22
Código 2 - Exemplo de uma mensagem de retorno que faz uso de *HATEOAS* 25
Código 3 - Exemplo de uma mensagem *SOAP* 26
Código 4 - Exemplo de WSDL 27
Código 5 - Exemplo de uma *query GraphQL* 29
Código 6 - Exemplo da especificação da lista de versões de cada serviço.................................. 93

Lista de Acrónimos e Siglas

Sigla/acrónimo	**Descrição**
AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
CRUD	*Create, Read, Update or Delete*
DoS	*Denial Of Service*
FEI	*Front End of Innovation*
FFE	*Fuzzy Front End*
FURPS	*Functionality, Usability, Reliability, Performance and Supportability*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTML	*Hypertext Markup Language*
HTTP	*Hyper Text Transfer Protocol*
IAM	Inconsistência Aleatória Média
IC	Índice de consistência
JSON	*JavaScript Object Notation*
NCD	*New Concept Development*
NPD	*New Product Development*
RC	Relação de consistência
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
SaaS	*Software as a Service*
SDK	*Software Development Kit*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*
SOLID	*General Responsibility Assignment Software Patterns*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*
W3C	*World Wide Web Consortium*
WSDL	*Web Services Description Language*
XML	*Extensible Markup Language*

1 Introdução

Este capítulo descreve o contexto, o problema e os objetivos do projeto descrito neste documento. São ainda apresentados a abordagem e processo de desenvolvimento e a estrutura do documento.

1.1 Contexto

Esta dissertação descreve um trabalho desenvolvido no âmbito de um estágio curricular realizado na empresa E-goi, referente à unidade curricular Tese de Mestrado, do Instituto Superior de Engenharia do Porto.

A E-goi é uma empresa que disponibiliza uma plataforma *Software as a Service* (SaaS) de marketing digital multicanal, e que conta com clientes em diversas partes do mundo, com maior relevo no mercado ibérico e da américa do sul. O seu crescimento atual motivou o lançamento de uma nova versão da sua *API* pública, para possibilitar que os seus clientes integrem as funcionalidades da plataforma nos seus próprios projetos. Esta *API* será disponibilizada no estilo arquitetural *Representational State Transfer* (*REST)* e no protocolo *Simple Object Access Protocol (SOAP).*

A disponibilização de *API* públicas é crucial nos negócios digitais, uma vez que permite às empresas exporem funcionalidades e dados aos seus clientes [1]. Tal como qualquer outro sistema de software, as *API* estão em constante mudança. Seja devido a uma evolução tecnológica ou por causa de mudanças no negócio, surge a necessidade de alterar os serviços disponibilizados. Por este motivo é importante analisar a evolução das *API*, o que implica ter em

conta estas mudanças num sistema de versionamento. Este processo ajuda a garantir que as *API* tenham um tempo de vida maior.

A escolha dos estilos arquiteturais ou protocolos a utilizar é outro fator a ter em conta no processo de decisão. Nos anos mais recentes, verificou-se um aumento da utilização de *REST* e um declínio da de *SOAP,* sendo que atualmente *REST* é o mais utilizado [2]. Em *REST*, o uso de *Hypermedia As The Engine Of Application State (HATEOAS) é* considerado boa prática e auxilia a navegação da *API*. Este é um elemento necessário para atingir o nível 3 no modelo de maturidade de Richardson [3], que mede o nível de maturidade *RESTful* de uma *API* [3].

Existem diversos fatores que facilitam a utilização das *API*, nomeadamente o uso de códigos *Hyper Text Transfer Protocol (HTTP)* e o conteúdo das respostas. É importante que, quando ocorram erros, estes sejam suficientemente claros. Outros fatores a considerar são a escalabilidade do projeto e a sua eficiência, pelo que é importante investigar práticas o uso de *caching* e limitação de uso.

1.2 Problema

A E-goi disponibiliza uma *API* pública que atualmente se encontra na versão 2 para disponibilização de serviços em *REST*, *SOAP* e *XML-RPC*. A *API* atual possui vários problemas (documentados em mais detalhe na secção 2.4 deste documento), nomeadamente ao não seguir convenções tipicamente associados com o estilo arquitetural *REST* na estruturação dos seus *Uniform Resource Locator* (*URL)* [4]*,* o que causa confusão nos utilizadores.

As rotas da *API* atual representam ações em vez de recursos e os verbos *HTTP* não estão a ser utilizados, o que não faz muito sentido de um ponto de vista *RESTful* [5, 4].

Por exemplo, as rotas atuais seguem a seguintes estrutura:

- Rota *REST* actual: "*GET*: *<URL* da *API REST>*/ *getCampaigns";*

No entanto era preferível uma abordagem mais *RESTful* em que sempre que seja possível os *URL* não devem estar associados a funções *CRUD* do protocolo *HTTP* [4].

Por esses motivos, a seguinte estrutura está mais de acordo com as convenções:

- Rota *REST*: "*GET*: *<URL* da API *REST>*/*campaigns";*

Na *API SOAP* é possível utilizar *JavaScript Object Notation (JSON)* e *Extensible Markup Language (XML).* No entanto a convenção diz que *SOAP* deve utilizar *XML* [6], pelo que esta implementação está incorreta.

Para além do problema referido, as respostas não são suficientemente claras para os utilizadores, sendo que este problema é acentuado pela *API* se encontrar insuficientemente documentada. Como consequência, a E-goi recebeu várias queixas de utilizadores no seu departamento de apoio ao cliente.

Outro problema do projeto atual é que este se encontra mal estruturado e com pouca separação de classes o que implica alto acoplamento e baixa coesão, o que complica a sua manutenção e expansão, como será detalhado na secção 2.4.

Desde maio de 2014 até fevereiro de 2018, a E-goi recebeu 807 tickets de apoio ao cliente relativos à *API*, o que dá uma média de cerca de 24 por mês. Por estes motivos a E-goi decidiu que a melhor solução seria criar um novo projeto com estrutura, representação dos recursos e documentação diferentes para substituir a versão 2.

1.3 Objetivos

O objetivo do projeto em que se insere esta dissertação é o desenvolvimento de uma nova versão da *API* pública para a empresa E-goi, que não possua os problemas descritos na secção 1.2. Os objetivos do trabalho descrito neste documento são uma parte desse novo projeto.

A Figura 1 apresenta de forma resumida a estrutura do projeto incluindo as componentes que são referentes a esta dissertação (a azul), o que será feito por outros elementos (a amarelo) e o que já existe feito (a verde). A especificação dos serviços, que funcionam como um contrato, e a sua utilização para gerar a documentação, testes automáticos e os SDK de acesso à *API,* assim como o sistema de registos de utilização, serão realizados por outras pessoas e, como tal, não são apresentados neste documento.

Internamente será necessário tratar a informação e retornar ao utilizador num formato agradável, garantindo que no caso de ocorrerem erros, seja transmitida a informação necessária para auxiliar o utilizador. Os serviços devem ser disponibilizados seguindo as convenções de *REST* e *SOAP*. Deve também de ser pensada na sua evolução a longo prazo e desenvolver um sistema de versionamento por serviço, sendo estes os principais objetivos deste projeto.

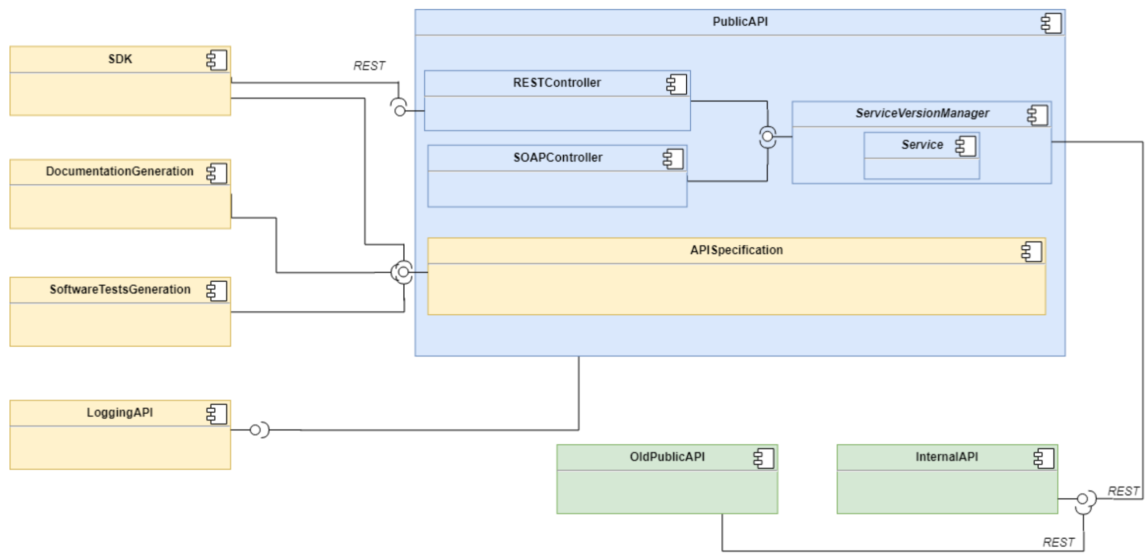


Figura 1 - Estrutura geral do projeto

Para atingir os objetivos referidos é necessário:

- Estudar conceitos, regras e recomendações na área, em especial as mensagens de erros serem úteis ao utilizador e fazerem uso dos códigos *HTTP* corretos, assim como a funcionalidade de *caching* e controlo de tráfego de utilização. Examinar o estilo arquitetural *REST* e o protocolo *SOAP*, tendo em conta as normas e boas práticas de cada um. Analisar a evolução das *API*, incluindo o versionamento e técnicas para evoluir os serviços.
- Ponderar diferentes alternativas de solução para o uso de *REST* e *SOAP*, com consideração dos seus benefícios e desvantagens, com o mínimo de duplicação de código uma vez que, com exceção das rotas e da estrutura do pedido e da resposta, a lógica de pedido é comum.
- Desenvolver a nova versão da *API* suportando *REST* e *SOAP*, pesadas as diferentes possibilidades encontradas*;*
- Desenvolver um sistema de versionamento por serviço que funcione tanto em *REST* como em *SOAP*. Este deve permitir que o utilizador especifique que versão do serviço pretende utilizar, diretamente ou através de um *wild-card*.

Existe ainda uma restrição por parte da empresa que impõe utilização do software *Zend Framework 2*.

1.4 Abordagem e Processo de Desenvolvimento

A abordagem para a resolução do problema envolve o estudo do estilo arquitetural *REST* de forma a o poder aplicar corretamente na versão 3 da *API*, dado que a quebra das regras deste estilo é um dos problemas da *API* existente. Será também necessário efetuar um estudo do protocolo *SOAP* e sobre evolução das *API*.

Seguidamente a nova *API* será desenhada, modelada e desenvolvida. Este processo será realizado de forma ágil e iterativa para que a solução final seja a melhor possível para a E-goi. Para auxiliar este processo, serão realizadas reuniões com a empresa, para analisar os problemas encontrados, o trabalho realizado e planear o que fazer de seguida. Numa fase inicial as reuniões serão diárias, uma vez que é nessa fase que existem mais duvidas. Após 2 meses as reuniões passarão a ser semanais uma vez que o desenvolvimento já estará em andamento.

Por fim, a solução desenvolvida será validada utilizando questionários de satisfação, testes de aceitação aos serviços disponibilizados e comparando os tempos de resposta em *REST* entre os pedidos à *API* desenvolvida e diretamente às *APIs* internas do E-goi.

1.5 Estrutura do Documento

Este documento está organizado nos seguintes capítulos:

1. **Introdução:** neste capítulo será explicado o contexto em que se enquadra este projeto, assim como o problema a resolver e os objetivos propostos. Serão ainda apresentados a abordagem e processo desenvolvimento utilizados;
2. **Estado da Arte:** neste capítulo será explicado o que é uma *API* e o protocolo *HTTP*. Serão ainda analisados e comparados *REST*, *SOAP* e *GraphQL* e abordados os temas limitação de uso e *caching*. Será também efetuado um estudo sobre evolução de *API*, incluindo versionamento e técnicas para evoluir as *API*. Por fim será realizada uma análise e documentação dos problemas da atual *API* da E-goi;
3. **Análise de Valor:** realização da análise de valor utilizando os seguintes pontos: explicação e aplicação do modelo *New Concept Development* (NCD) [7], explicação dos conceitos de valor; valor para o cliente; valor percebido e apresentação de uma tabela longitudinal de valor; apresentação da proposta de valor deste projeto;

modelo *Canvas* [8] aplicado a este projeto; cadeia de valor de *Porter* [9]; explicação e utilização do modelo *Analytic hierachy process* (*AHP*) [10] para selecionar a melhor alternativa para resolver um problema;

4. **Análise e Design arquitetural :** serão efetuadas análises de negócio e de requisitos do sistema a ser desenvolvido, apresentado o design arquitetural e discussão das diferentes alternativas arquiteturais;
5. **Design e implementação:** apresenta o design da solução onde são expostos diagramas de classe referentes às componentes mais importantes do projeto. Exibe ainda artefactos relativos à implementação e as decisões tomadas. Apresentam-se também os resultados dos testes de aceitação e do tempo de resposta;
6. **Avaliação da Solução:** apresentação das grandezas utilizadas para testar a solução, as hipóteses que se pretende testar, a metodologia de avaliação e os resultados do questionário de satisfação;
7. **Conclusão**: contém a conclusão do que foi realizado neste projeto, os resultados obtidos e o trabalho futuro.

O documento contém ainda os seguintes anexos:

1. **Anexo A:** o questionário utilizado no modelo AHP;
2. **Anexo B:** apresenta o questionário de satisfação comparativa entre a *API* pública anterior e a nova;
3. **Anexo C:** contém duas alternativas arquiteturais sobre como poderia ser disponibilizado *GraphQL* no futuro.

2 Estado da Arte

Neste capítulo serão descritos de forma detalhada vários conceitos e tecnologias relevantes para o trabalho de mestrado descrito neste documento. Primeiramente serão explicados os conceitos de *API e* o protocolo *HTTP.* De seguida será analisado o estilo arquitetural *REST,* o protocolo *SOAP* e *GraphQL.* Por fim será apresentada uma comparação entre ambos. Durante a criação das *API* públicas é importante ter em conta algumas noções para que a sua utilização seja mais agradável para os utilizadores. Desta forma, neste capítulo serão abordados o modelo de maturidade de Richardson [3] e o uso de códigos *HTTP* e as mensagens de erro.

Serão também analisados os conceitos de *caching*, limitação de uso assim como o versionamento das *API* e as diferentes soluções existentes. Estas técnicas tentam resolver o problema de que ao longo do ciclo de vida de uma *API*, esta vai sofrer alterações que devem ser o mais transparente possíveis para o utilizador. Serão ainda analisadas algumas *API* públicas de empresas concorrentes do E-goi e a versão anterior da *API* pública que será substituída pela produzida neste projeto. Por fim serão também apresentadas as tecnologias utilizadas no desenvolvimento.

2.1 API

Uma *API* define um contrato para aplicações comunicarem através da internet sem interação de utilizadores [11] *e* oferece uma forma simples para exceder, conectar ou integrar software [12]. Em termos técnicos, uma *API* define o contrato de uma componente de software, relativamente ao protocolo utilizado, formato dos dados e rotas disponibilizadas, para que duas aplicações de computador possam comunicar entre si através de uma rede [11]. Uma vez que as únicas pessoas

que interagem diretamente com uma *API* são desenvolvedoras, que querem integrar as funcionalidades oferecidas nas suas aplicações, isso implica que as *API* sejam construídas com os eles em mente.

O conceito de *API* é mais antigo que a era da computação pessoal. Desde os primeiros dias do processamento de dados que existe a necessidade de permitir que uma aplicação interaja com outro sistema. No entanto, o advento da *internet* fez com que a utilidade destes sistemas aumentasse drasticamente [13]. Atualmente o número de *API* para uso na internet tem crescido num ritmo sustentado [14].

Em 1990 Tim Berners-Lee começou um projeto de *software* ao qual ele chamou "WorldWideWeb", em que criou os seguintes elementos: *Uniform Resource Identifier (URI)*; o protocolo *HTTP*; a linguagem *Hypertext Markup Language (HTML*); o primeiro servidor *web*; o primeiro navegador [4].

No espaço de 5 anos o número de utilizadores da *internet* subiu para 40 milhões. O número crescia exponencialmente e a certo ponto duplicava a cada 2 meses. Este crescimento era mais rápido que o crescimento da infraestrutura da internet e havia um problema de escalabilidade para ser resolvido. Por esse motivo alguns estilos arquiteturais foram propostos [4].

No final de 1993, Roy Thomas Fielding, um cofundador do projeto "Apache *HTTP* Server", identificou 6 restrições chave que teriam de ser uniformemente satisfeitas de forma a resolver os problemas de escalabilidade. As 6 restrições são as seguintes [4]:

- **Cliente-Servidor:** a *Internet* é baseada em sistemas cliente-servidor. Clientes fazem pedidos a servidores e os servidores respondem;
- **Interface uniforme:** as interações entre componentes *web* devem seguir as seguintes quatro restrições;
    - **Identificação de recursos:** cada componente *web* deve ser identificado por um identificador único;
    - **Manipulação de recursos através de representações:** os recursos podem estar representados em diferentes linguagens (*e.g. HTML*, *XML, JSON*) e a sua manipulação deve ser feita através destas representações e não diretamente;
    - **Mensagens auto descritivas:** as mensagens do cliente devem incluir o estado desejado dos recursos e meta data adicional. O servidor deve decidir se aceita ou não o pedido e retornar uma mensagem com o estado final do recurso;

- *HATEOAS:* A mensagem representativa do estado do recurso, deve incluir links para operações relativas a esse recurso ou outros recursos de interesse.

- **Sistema de camadas:** nenhuma camada tem conhecimento além da outra imediatamente a seguir ou da anterior;
- **Cache:** os dados de resposta devem poder ser guardados localmente, aumentando a disponibilidade e confiabilidade da aplicação cliente e reduzir a carga na aplicação servidora;
- **Sem estado:** as aplicações servidoras não têm obrigação de memorizar o estado das aplicações clientes e é responsabilidade destas transmitir as informações contextuais necessárias na mensagem do pedido;
- **Código sob demanda:** esta restrição é a única que é considerada opcional uma vez que reduz a visibilidade. Esta restrição possibilita que que os servidores decidam como algumas coisas vão ser feitas no cliente (*e.g.* o cliente descarregar código para ser executado), o que aumenta a flexibilidade deste.

O conceito de *web API* surgiu em meados do ano 2000, durante o movimento *Service Oriented Architecture* (*SOA*). Os desenvolvedores necessitavam de uma forma padrão de permitir que as aplicações comunicassem entre si, o que levou à criação das *API* [13]. Enquanto *SOA* ajuda na agilidade e velocidade com que o software é distribuído, as *API* ajudam na velocidade de inovação para construir aplicações [11]. De um ponto de vista técnico, *SOA* é relacionado com *XML* e *SOAP* e pode ser descrito usando *WSDL*, enquanto *API* são relacionadas com *REST* e *JSON* e podem ser descritas usando *Swagger* ou *RAML* [11].

2.1.1 Controlo de tráfego de utilização

Sendo que as *API* recebem pedidos de diversos utilizadores e estão a correr numa máquina física com limitações, surge a necessidade de controlar o seu uso. Para além de permitir lidar com as limitações físicas dos servidores, estes sistemas devem podem ter outras considerações. Por exemplo, uma empresa pode criar diferentes planos de contas de utilizador a preços distintos, e atribuir a cada plano um limite de uso diferente. Desta forma estes sistemas têm também o potencial de trazer lucro.

A gestão do tráfego permite às empresas oferecer diferentes valores de negócio para diferentes clientes [11]. Por exemplo, as empresas podem permitir que clientes gratuitos disponham de uma quantidade limitada de pedidos por dia e que os clientes pagantes, possuam um limite

maior ou mesmo ilimitado. Outro exemplo é a criação de um limite de pedidos, durante horas de pico na utilização da *API*. As *API* devem disponibilizar as seguintes capacidades para controlar o tráfego [11]:

- **Quota de utilização:** define a quantidade de pedidos que uma aplicação pode fazer num determinado período. Quando atingida a quota, o tráfego pode ser abrandado ou bloqueado. Podem existir diferentes quotas por tipo de cliente, consoante a política de negócio da empresa [11];
- ***Spike Arrest***: identifica subidas inesperadas no tráfego da *API* e ajuda a proteger sistemas que não possuem capacidade para lidar com uma carga tão alta, ou a proteger de ataques *Denial Of Service (DoS)* [11];
- **Limitação de uso:** um mecanismo para abrandar os pedidos seguintes. Este sistema pode ajudar a aumentar a performance geral e reduzir o impacto durante horas de grande utilização [11]. Existem três formas de limitar o tráfego [15]:
    - **Por utilizador:** este limite está diretamente ligado à *apikey* ou *acess token* do utilizador. Um exemplo desta utilização é permitir a cada utilizador 30 pedidos por hora;
    - **Por servidor:** limitar diretamente o uso de servidores específicos;
    - **Por região**: com base na região de onde o cliente executou o pedido.
- **Priorização de tráfego:** ajuda a decidir que classe de clientes deve receber uma prioridade superior. Pedidos de clientes com uma prioridade superior, devem processados primeiro [11].

2.1.2 Caching

Caching é um mecanismo para otimizar a performance ao responder aos pedidos com respostas estáticas guardadas na memoria [11]. Esta é uma funcionalidade construída no topo do protocolo *HTTP* e uma das restrições mais importantes da arquitetura da *web,* uma vez que permite:

- reduzir a carga nos servidores das *API*;
- reduzir latência percebida pelo cliente;
- aumentar a fiabilidade da *API*.

Por outras palavras, “No geral, *caching* reduz o custo da *web”* [4]. A *cache* pode existir em qualquer parte da rede entre o cliente e o servidor [4]. Pode ser utilizado em pedidos que utilizem os métodos *HTTP GET* e *HEAD*, uma vez que estes servem para retornar dados, pretendendo-se que as respostas sejam guardadas e geradas assim menos vezes.

Caching pode ser usado em *REST,* mas não em *SOAP*. Tal sucede porque no caso de uso de *SOAP* utilizando o protocolo *HTTP* como protocolo de transporte, os pedidos são realizados utilizando o método *HTTP POST*, de forma a poder transmitir o *SOAP Envelope* no *body* da mensagem *HTTP*.

Existem os seguintes *HTTP headers* referentes a cache [16]:

- ***Cache-Control:*** este *header* notifica o cliente ou browser para utilizar cache. Pode ser definido como *private* que permite que apenas o cliente final utilize *cache* dos dados, ou *public* que também permite que *proxies*, ente o cliente e a *API*, também o façam;
- ***Max-age:*** especifica um limite para quanto tempo manter a *cache* do recurso;
- ***Expires:*** indica uma data para quando a cache do recurso deve ser considerada inválida. Se enviado em conjunto com o *header max-age,* este é precedente;

É possível realizar pedidos condicionais de forma a averiguar se o servidor possui uma cópia mais recente do recurso. O cliente envia informação sobre a cópia que possui e o servidor determina, com base em diferentes estratégias, se a cópia é a mais recente. Estas estratégias podem ser:

- **Por tempo:** para ativar estra estratégia, a *API* deve especificar a última data em que um recurso foi modificado através do *HTTP header Last-Modified*. O cliente pode depois realizar um pedido enviando o *header If-Modified-Since* para obter a nova versão, caso o recurso tenha sido modificado ou uma resposta sem conteúdo caso não tenho;
- **Por conteúdo:** esta estratégia funciona através da utilização do *header ETag*, cujo valor é uma representação do conteúdo do recurso (*e.g. hash*). O cliente pode depois realizar um pedido utilizando o *header If-Modified-Since* para comparar a representação atual com a *ETag* e obter o recurso, caso seja diferente, de forma similar à estratégia por tempo.

Caso, não faça sentido usar cache para uma certa resposta, poderá ser retornado o atributo “*no-cache*”, mas regra geral, é preferível retornar um valor inferior em “*max-age*” e encorajar de *caching* [4].

2.1.3 Evolução das API

Tal como qualquer outro sistema de software, as *API* estão em constante mudança. Seja devido a uma evolução tecnológica ou por causa de mudanças no negócio, surge a necessidade de alterar os serviços disponibilizados. Isto cria um problema, uma vez que as aplicações dependentes da *API* também têm de mudar para a se adaptarem ao novo modo de utilização. Diversos estudos foram realizados sobre este tema [17] e apresentam-se seguidamente alguns deles e as suas conclusões, para que se possa entender melhor esta temática.

Num estudo realizado por Tiago Espinha, Andy Zaidman e Hans-Gerhard Gross foram entrevistadas algumas pessoas, baseado nos seus projetos *open source* que utilizam outras *API*. Concluiu-se com o estudo que que manter os projetos integrados com as *API* que sofreram alterações é mais trabalhoso do que integrar com as *API* da primeira vez [18]. Esta conclusão demostra que as mudanças nas *API* causam um problema aos seus utilizadores.

Num estudo empírico realizado por Shaohua Wang, Iman Keivanloo e Ying Zou foram listadas as alterações mais comuns que ocorrem ao longo do ciclo de vida das *API*. Para tal foram selecionadas as *API* mais populares listadas no site *ProgrammableWeb*, com base nos seguintes critérios: As *API* têm pelo menos duas versões; As *API* são de diferentes domínios de aplicação; As *API* são de diferentes empresas [19].

Seguidamente para cada uma das API selecionada, foram analisadas as documentações das várias versões e colecionados dados de discussões no site *StackOverflow*, sobre mudanças nas *API,* utilizando uma pesquisa com base em palavras-chave especificas para cada *API* analisada. O estudo identificou que as mudanças mais comuns são as seguintes [19]:

- Ao nível da *API*:
    - Formato de resposta alterado (mudança de atributos ou de formato);
    - Mudança de URL (*e.g* api.twitter.com/1 para api.twitter.com/1.1);
    - Modificação ou adição de um método de autenticação;
    - Alteração nos limites de utilização;
    - Remoção ou adição de um formato de resposta (*e.g. XML*).
- Ao nível dos serviços da *API*:
    - Nome do serviço alterado;
    - Alteração na estrutura da resposta;

- Serviço adicionado ou removido;
- Adição de códigos de erro.

Um dos fatores chave para o sucesso das *API* é garantir um nível adequado de estabilidade. Mudanças na *API* podem quebrar as aplicações dos clientes e por esta razão garantir a estabilidade da *API* ajuda a reduzir a necessidade de mudanças constantes [20]. A coesão da *API* afeta a probabilidade de mudanças assim como a sua facilidade de manutenção e legibilidade [20]. Existem os seguintes diferentes níveis de coesão nas *API*: coincidência, logica, temporal, comunicacional, externa, implementação, sequencial e conceptual [20]. Num estudo realizado por Daniele Romano [20] focado na coesão interna, que mede a coesão das operações e comunicacional, que mede a extensão com que as operações são usadas pelos clientes, foram identificados os seguintes *antipatterns* [20]:

- *Multiservice:* uma *API* com esta *antipattern* expõe operações relacionadas com diferentes entidades através de uma única interface. Estas *API* possuem baixa coesão interna e podem ter problemas de performance [20];
- *Fat:* ocorre quando operações desconectadas são invocadas por diferentes tipos de clientes. Estas *API* devem ser divididas em *API* mais pequenas, de acordo com os as operações a que cada tipo de cliente tem acesso [20].

O estudo ainda analisou em que cenários, as *API* com problemas de coesão comunicacional e coesão interna, os desenvolvedores decidem alterar a *API*. Concluiu-se que as *FAT API* têm grande probabilidade de ser alteradas para facilitar a sua manutenção e melhorar a sua legibilidade. As *MultiService API* também possuem grande probabilidade de serem alteradas, porque as mudanças numa entidade tendem a afetar todos os clientes, e também possuem problemas de legibilidade. A probabilidade dos serviços ou estruturas de dados serem alteradas é alta em ambos os casos [20].

2.1.3.1 Versionamento

Para o problema das *API* serem inevitavelmente alteradas, os provedores de serviços têm duas soluções. Podem providenciar e manter diferentes versões da sua *API*, pelo que os clientes têm de escolher qual querem usar. Alternativamente alguns clientes podem querer não ter de lidar de forma explicita com diferentes versões e neste caso terá de ser providenciado um sistema de versionamento dos serviços da *API* e uma forma de dinamicamente aceder às diferentes versões sem necessidade alterar o código cliente [21].

Apesar dos problemas causados pelas mudanças nas API, estas são muitas vezes necessárias por questões de segurança ou performance. No entanto a compatibilidade com serviços anteriores é desejável e o utilizador deve saber se existem incompatibilidades ao utilizar versões mais recentes. A responsabilidade de informar os utilizadores é do criador da *API* e uma forma de o fazer é utilizando o sistema de versionamento semântico (SemVer[1]) [22]. Neste sistema criado em 2010 por Tom Preston-Werner (fundador do GitHub), a versão de cada serviço é um numero no formato "*Major.Minor.Patch*" (*e.g.* versão 1.2.4). Cada número possui o seguinte significado [22, 23]:

- ***Major:*** deve ser incrementado caso seja realizada uma mudança que causa problemas de incompatibilidade;
- ***Minor:*** deve ser incrementado caso seja adicionada uma funcionalidade, mas mantenha a compatibilidade;
- ***Patch:*** deve ser incrementado caso seja corrigido algum erro, mantendo a compatibilidade.

Para os desenvolvedores que utilizam as *API* é ideal que seja óbvio qual versão dos serviços estão a utilizar. Os dados necessários para selecionar a versão do serviço podem ser passados das seguintes formas [11]:

- ***URL***: nesta forma a versão de um recurso é diretamente enviada no *URL,* o que funciona bem se não existirem novas versões frequentemente. Nesta opção o cliente especifica a versão da seguinte forma [11]:

  "http://www.api.com/<version>/resource"

  Esta opção possui como principal vantagem a alta visibilidade da versão dos recursos [11]. Um problema com esta solução é que o versionamento é feito por recurso em vez de ser por serviço e por esse motivo não pode ser aplicado neste projeto;
- ***HTTP header***: nesta opção é utilizado um *HTTP header* para indicar qual a versão pretendida. São vantagens deste sistema a versão ser mantida longe do *URI*, que é utilizado para referir recursos, e a possibilidade de poder ser ignorado caso o utilizado não pretenda especificar a versão [11]. A versão pode ser enviada através do *header Accept* ou através de um *header* personalizado [11].

[1] URL do site SemVer: https://semver.org/

- **Parâmetros de *query*:** esta opção é bastante comum e o cliente especifica a versão da seguinte forma [11]:

  "http://www.api.com/resource?version=<version>"

  Tal como a opção de utilizar o *HTTP header*, uma vantagem desta opção é que a passagem da versão é opcional [11].

- **Nome de domínio:** outra opção é utilizar um domínio diferente para cada versão da *API*, no entanto esta opção só é utilizada caso existam grandes mudanças na *API*. Os problemas desta versão é que para alem da mudança do *URL*, pode necessitar ainda de mudanças nas configurações de segurança do lado do cliente, devido à mudança de domínio, ou ainda a necessidade de uma nova infraestrutura para suportar a nova versão. A principal vantagem é que a estrutura das rotas da nova *API* pode ser alterada uma vez que a *API* anterior pode continuar a receber os pedidos dos clientes [11]. A e-goi utilizou esta opção na criação da *API* pública versão 2 e vai utilizar de novo para a 3 de forma a poder alterar as rotas. No entanto existe a possibilidade de poder suportar versionamento por serviço na nova *API*.

Independentemente do sistema de versionamento adotado, este deve seguir os seguintes princípios gerais [11]:

- Uma nova versão não deve afetar as aplicações dos clientes. De outra forma pode frustrar os clientes e atrasar a adoção da *API*;
- Tentar evitar o lançamento de uma nova versão da *API* ao máximo. As empresas que lançam uma nova versão gastam dinheiro ao manter ambas as versões em simultâneo de forma a facilitar a transição e os clientes gastam tempo e dinheiro a alterar as suas aplicações para utilizarem a nova versão;
- Tentar realizar alterações que sejam compatíveis com as anteriores. Por exemplo, adicionar um elemento opcional não necessita de alterações do lado do cliente, ao contrário de adicionar um novo argumento obrigatório;
- O versionamento da *API* não deve estar diretamente ligado ao versionamento do *software.* Uma vez que o software evoluiu frequentemente, se a versão dos serviços estivesse diretamente ligada à versão do software, isso implicaria o lançamento de uma nova versão do serviço frequentemente.

2.1.3.2 Técnicas de evolução

O problema de alterar serviços disponibilizados é que obriga a que os clientes tenham de alterar as suas próprias aplicações. No entanto, quando as mudanças são necessárias e suficientemente grandes, obrigam à criação de uma nova versão da *API*. A *API* pública da E-goi vai ser completamente restruturada em relação à versão anterior, e como tal, a E-goi decidiu que uma nova versão seria criada. Para utilizar a nova versão os utilizadores acedem "*URL* da *API*/V3/" em vez de "*URL* da *API*/V2/". No entanto, faz sentido que a nova API possua um sistema de versionamento por serviço de forma a evitar a necessidade de criar uma nova versão no futuro.

Piotr Kaminski, Hausi Müller e Marin Litoiu propuseram em 2006 uma técnica chamada cadeira de adaptadores. Este sistema permite que o código fonte dos serviços evolua de uma versão para a próxima através de um adaptador. Para aplicar esta técnica os desenvolvedores têm de seguir os seguintes passos [24]:

1. Duplicar a interface do serviço numa rota diferente, mantendo a estrutura do original, mas sem ter nenhuma relação formal com o seu parente. Esta interface será a partir de agora chamada de "interface v1";
2. Criar uma implementação da interface v1 que reencaminha o pedido e os dados para a interface da versão anterior, alterando os dados seguindo as seguintes regras:
    - Se a nova interface requer novos parâmetros, o adaptador deve providenciar um valor por defeito;
    - Se a estrutura de dados é alterada, o adaptador deve traduzir os dados entre as versões;
    - Se uma operação for removida, esta deve ser disponibilizada novamente com base nos serviços disponíveis no momento.
3. Publicar a interface v1 como a versão estável do serviço.

A Figura 2 apresenta a estrutura da cadeia de adaptadores após várias versões terem sido disponibilizadas.

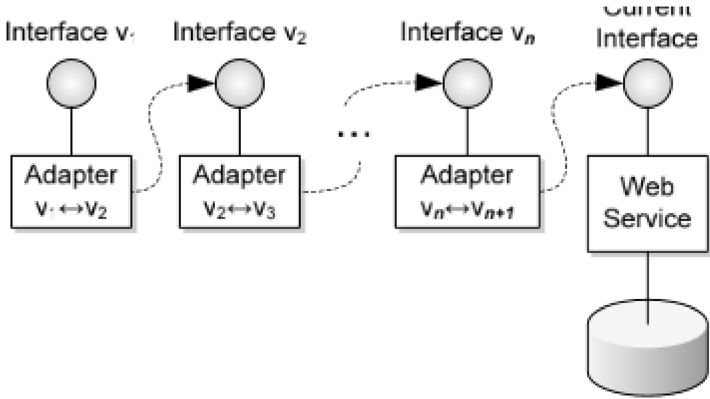


Figura 2 - Estrutura da cadeia de adaptadores [24]

Esta técnica permite que várias versões do mesmo serviço sejam disponibilizadas concorrentemente, evitando duplicação de código e apresentando um design simples [24].

Outra técnica proposta por Philip Leitner, Anton Michlmayr, Florian Rosenberg e Schahram Dustdar apresentaram uma técnica de versionamento automático baseada nos ficheiros *WSDL,* que foi integrada no sistema orientado a serviços VRESCo. Neste artigo foram identificados três tipos de alterações de alto nível, sendo que as alterações semânticas não são suportadas no sistema. Os tipos de alterações são as seguintes [21]:

1. Alterações não funcionais: alterações na interface não funcional do serviço e que não requerem necessariamente uma nova definição do serviço;
2. Alterações de interface: alterações na interface funcional do serviço que se encontra descrita no *WSD.* Cada serviço é descrito com um nome, uma lista de argumentos de entrada e de resposta. Cada argumento possui um nome e um tipo e podem ser opcionais ou mandatários;
3. Alterações semânticas: alterações que não estão descritas na descrição *WSDL* do serviço.

O sistema utiliza o conceito de grafos de versionamento de serviços, apresentado na - Grafo de versionamento de serviço Figura 3, para guardar as revisões dos serviços e como estas se relacionam. Os grafos são constituídos por vértices que representam revisões concretas dos serviços e arestas para representar as ligações entre as versões. O grafo engloba ainda os conceitos de *branche*, que representam situações onde duas ou mais variantes dos serviços envolvem em paralelo e *merge* onde duas ou mais revisões são consolidadas num único vértice. As revisões podem ainda conter uma *tag* para descrever as funcionalidades, estabilidade e nível de maturidade do serviço (*e.g INITIAL, STABLE, LATEST, DEPREC*) [21].

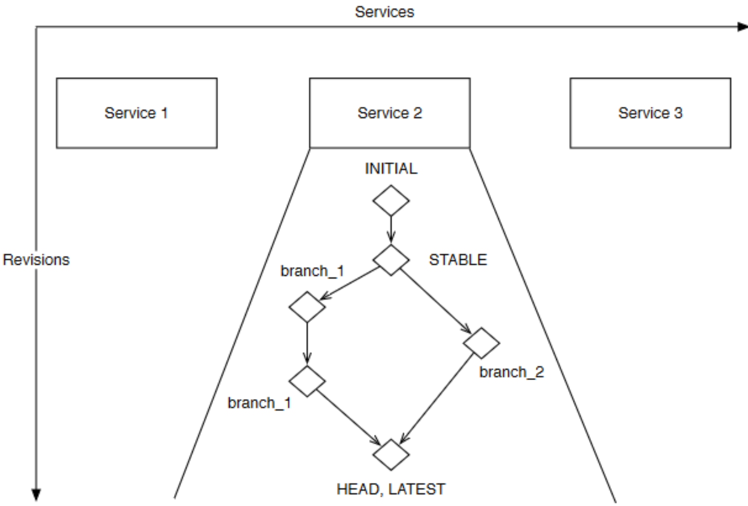


Figura 3 - Grafo de versionamento de serviço [21]

No artigo é proposta a utilização de *service proxies* que estão ligados a uma estratégia de seleção e dinamicamente escolhem qual a melhor versão do serviço a ser chamada. As estratégias podem ser por exemplo a escolha da versão mais recente ou a última estável. Esta camada deve ainda de permitir aos clientes escolher manualmente a versão a ser escolhida, alcançando assim um versionamento transparente.

2.2 HTTP

HTTP é um protocolo para distribuição colaborativa de sistemas *hypertext* [25], definido e mantido pela organização *World Wide Web Consortium* (*W3C*). O estilo arquitetural *REST* é disponibilizado sobre este protocolo e o protocolo *SOAP* também o pode utilizar. A transferência de informação é realizada através da utilização de mensagens *HTTP*, onde os clientes fazem pedidos aos servidores que por sua vez lhes respondem.

As mensagens de pedido e resposta seguem uma estrutura similar e possuem os seguintes elementos [25]:

- ***Start-line***: nas mensagens de pedido este elemento contém informação sobre o *URI*, a versão do protocolo e o método *HTTP* utilizado. No caso de uma resposta contém a versão do protocolo e o código estado retornado;
- ***Header***: estes elementos são opcionais e seguem um formato constituído por um nome, seguido de 2 pontos (":"), seguido de um valor. Esta camada serve para enviar informações adicionais para o servidor e existe um conjunto de *headers* predefinidos na

convenção (*e.g.* “*Content-Type*”, “*Coockie*”). É também possível ao cliente criar ou utilizar *headers* próprios, que podem ser usados pela *API*, caso se encontre programada para tal;

- ***Body*:** neste elemento, caso este exista, é transportada a entidade associada com o pedido ou a resposta. Um exemplo de utilização seria na criação de um novo elemento de um dado recurso. Este elemento seria representado e transmitido no *body* e utilizado pelo *API* para inserir o novo elemento na base de dados.

O protocolo *HTTP* define um conjunto de métodos, que definem a ação a ser executada que são transmitidos na mensagem. Estes métodos são classificados com base nas seguintes propriedades:

- **Método seguro:** não altera os dados;
- **Método Idempotente:** se utilizado mais que uma vez tem o mesmo efeito que uma só vez.

Na Tabela 1 estão listados os métodos *HTTP* (documentados no protocolo *HTTP* [5]), a sua função e como se enquadram nas propriedades previamente definidas.

Tabela 1 - Comparação dos métodos *HTTP*

Método	**Função**	**Seguro**	**Idempotente**
GET	Ler a informação disponibilizada pelo *URI*	Sim	Sim
POST	Pede à *API* para escrever novos dados	Não	Não
PUT	Criar ou atualizar dados disponibilizados pelo *URI*	Não	Sim
PATCH	Atualizar parcialmente os dados disponibilizados pelo *URI*	Não	Não
DELETE	Apaga os dados disponibilizados pelo *URI*	Não	Sim
OPTIONS	Ler os métodos *HTTP* permitidos pelo *URI*	Sim	Sim
HEAD	Ler apenas os *headers* da informação disponibilizada pelo *URI*	Sim	Sim
TRACE	Executa uma mensagem *loop-back* de teste no *URI*, retornando o que estava no *body*	Não	Sim
CONNECT	Estabelece um túnel de comunicação com o *URI*	Não	Sim

Após o pedido ser recebido e interpretado, a *API* deve responder com uma mensagem onde para além de apresentados os dados requisitados, terá de ter associado um código de resposta. Estes códigos vão ser falados em mais detalhe na secção 2.2.1.

Os códigos dividem-se nas seguintes cinco categorias [4]:

- 1xx Informação: comunica informação ao nível do protocolo de transferência;
- 2xx Sucesso: indica que o pedido do cliente foi aceite com sucesso;
- 3xx Redirecção: indica que os clientes têm de realizar ações adicionais de forma ao seu pedido poder ser completado;
- 4xx: Erro do cliente: lista erros cuja culpa é do cliente;
- 5xx: Erro do servidor: lista erros cuja culpa é do servidor.

2.2.1 Códigos HTTP e mensagens de erro

Os códigos de resposta *HTTP* servem para indicar o resultado do pedido, e em caso de erros dar alguma informação sobre o erro. Os códigos começados por 2 são utilizados em caso de sucesso, os iniciados por 4 utilizados em caso de erros cuja culpa é do utilizador, e o código 500 quando ocorre um erro cuja culpa é do sistema. Os seguintes códigos são frequentemente utilizados [4, 26]:

Tabela 2 – Códigos *HTTP* frequentemente utilizados

Código de erro	***Descrição***
***200* (“OK”)**	Indicar sucesso genérico e retornar algo no *HTTP Body* *(e.g.* no caso de um *HTTP GET* de todas as listas de contactos
***201* (“Created”)**	Assinalar que um determinado recurso foi criado
***202* (“Accepted”)**	Indicar que o pedido foi recebido e que uma função assíncrona foi iniciada
***204* ("No Content")**	Mostrar que o pedido foi um sucesso, mas sem retornar texto na resposta (*e.g.* no caso de um *delete)*
***400* (“Bad Request”)**	Ocorreu um erro do lado do cliente sem o especificar

Código de erro	*Descrição*
401 **(“Unauthorized”)**	Problemas de autenticação do cliente
403(“**Forbidden*****”):***	Indicar que o cliente, que se encontra autenticado, não possui permissões para realizar a operação que pretende
404 **(“Not Found”):**	O *URI* indicado pelo cliente não existe ou que o recurso pretendido também não existe
405 (“**Method Not Allowed*****”)***	O método *HTTP* utilizado não existe, para o recurso escolhido
406 (“**Not Acceptable*****”)***	Indicar que o tipo de dados no qual o cliente pretende receber a resposta, não é aceite pela *API* (*e.g.* numa *API REST* que apenas suporte *JSON*. Se o cliente indicar que pretende receber uma resposta em *XML*, este código deve ser retornado)
409 (**"Conflict"*****)***	O pedido não pode ser concluído devido a um conflito com o estado de um recurso
410 (“Gone”)	O recurso alvo já não existe e esta condição é provavelmente permanente
415 (“**Unsupported Media Type”)**	Indicar que o tipo de dados enviado pelo cliente e especificado usando o *http header” Content-Type”*, não é suportado pela *API (e.g.* enviar *XML* quando apenas *JSON é* suportado*)*
422 ("Unprocessable Entity")	Indicar que o sistema compreende o tipo de dados pedidos pelo cliente, mas que os dados que o cliente passou, apesar de estarem sintaxicamente corretos, contem semânticos
500 (“**Internal Server Error*****”):***	indicar um problema na *API*

No caso de ocorrerem erros, a mensagem retornada deve transmitir claramente que erro ocorreu e o motivo. Deve ainda transmitir informação sobre como o utilizador deve proceder e onde pode consultar mais informações. No extrato de Código 1, podemos ver uma mensagem de erro usada no caso de uma chamada a um *URI* para um recurso não existente.

```
{
  "Error": "Resource Not Found",
  "Reason": "The id you sent doesn’t correspond to an existing
element",
  "Solution": "Use an id of an element that exists"
}
```

Código 1 - Exemplo de uma mensagem de erro em *JSON*

2.3 *REST, SOAP e GraphQL*

Nesta secção será analisado o protocolo *SOAP*, o estilo arquitetural *REST* e a tecnologia *GraphQL*. Por fim será apresentada uma comparação entre cada um.

2.3.1 REST

Uma *web API* que siga o estilo arquitetural *REST* é conhecida por *REST API*. A arquitetura foi proposta em 2000 por Roy Thomas Fielding na sua dissertação de doutoramento, na qual apresentou 6 restrições chave que teriam de ser uniformemente implementadas pela *internet* [27]. Esta arquitetura respeita essas restrições e todas as regras do protocolo *HTTP*.

Uma *API REST* consiste numa montagem de recursos interligados, conhecidos como modelo de recursos REST [4]. Estas *API* representam recursos e não ações. Existem os seguintes tipos de recursos [4]:

- **Documento:** é um conceito singular que é semelhante a uma instância de objeto ou base de dados. Por exemplo: "www.api.e-goi";
- **Coleção:** é um diretório de recursos. Os clientes podem propor adicionar novos recursos à coleção, mas fica ao critério desta decidir. Por exemplo: "GET: /lists";
- **Loja:** uma loja permite aos clientes colocar recursos, ler os recursos ou eliminá-los. Por exemplo: "PUT: /lists/1/contacts";
- **Controlador:** é um recurso que representa uma função, com argumentos de entrada e de saída. Por exemplo: "GET: /lists/filterByEmail".

As *API REST* devem seguir o seguinte conjunto de características relativas a *URLs* [4]:

- O separador "/" deve ser usado para representar uma relação hierárquica. Este separador não pode ser o último elemento;
- O carater “-“ deve ser usado para aumentar a legibilidade do *URL;*

- O carater "_" não deve ser usado;
- Letras minúsculas devem ser preferidas;
- Extensões de ficheiros não devem ser incluídas. Esta informação deve ser passada através do *HEADER* "*Content Type*";
- Devem ser usados nomes de subdomínio consistente. O primeiro subdomínio deve ser "*API*" (*e.g.* www.api.e-goi.com);
- Devem representar recursos (*e.g lists*) em vez de ações;
- Um documento deve ter um nome singular;
- Uma coleção deve ter um nome plural;
- Uma loja deve ter um nome plural;
- Um controlador deve ter um verbo no seu nome;
- Os nomes das funções de *Create*, *Read*, *Update* ou *Delete* (*CRUD*) (*e.g. POST*, *PUT*) não devem ser usados no *URL.*

Este estilo arquitetural interage com o protocolo *HTTP* e usa os métodos descritos anteriormente. O único caso excecional é que o método *POST*, para além de ser usado para adicionar novos recursos em coleções, deve também de ser usado para chamado recursos do tipo controlador. Além disso, os códigos de erro usados são os mesmos do protocolo *HTTP*.

Para medir o nível de maturidade de uma *REST API,* o modelo de maturidade de Richardson [3] foi criado. O modelo de maturidade de Richardson mede o nível de maturidade de uma *API*, classificando-a de nível 0 a nível 4 e descrevendo os 3 passos necessários para a *API* ser verdadeiramente *RESTful* [3]. O esquema da Figura 4 representa o modelo de maturidade de Richardson.

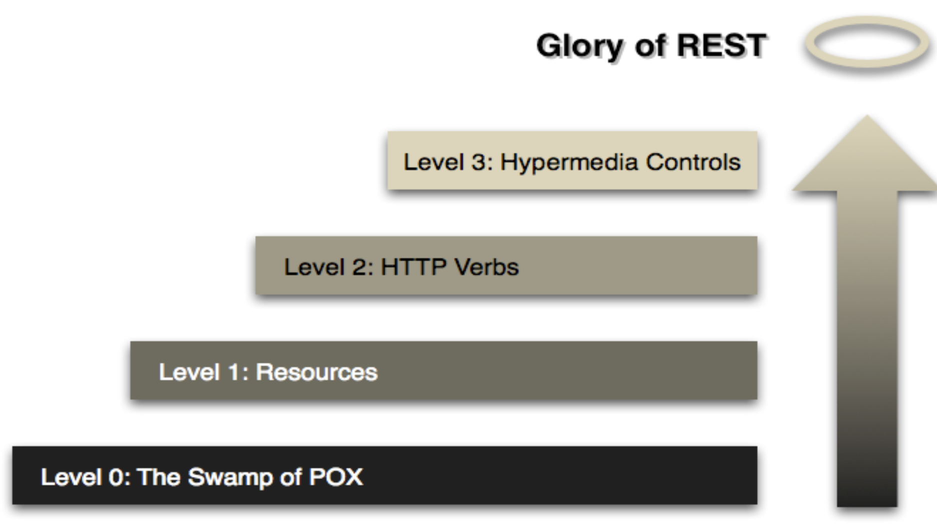


Figura 4 - Níveis do modelo de maturidade de Richardson [3]

Os quatros níveis de maturidade são os seguintes [3]:

- **Nível 0:** o protocolo *HTTP* deve ser utilizado como protocolo de transporte, para chamar procedimentos remoto. O servidor deve disponibilizar rotas para a chamada dos serviços pelo que neste nível o sistema é um sistema de *RPC* típico;
- **Nível 1:** o primeiro passo para uma *API RESTful* é a introdução do conceito de recursos. Em vez de disponibilizar uma rota para cada procedimento, existe uma por recurso. Enquanto no nível 0 seria possível disponibilizar um serviço que envolva a manipulação de dois recursos diferentes, neste é necessário disponibilizar uma rota por cada recurso;
- **Nível 2**: neste nível são utilizados e respeitados os verbos *HTTP* (*e.g. POST*, *PUT*, *GET, DELETE*). Numa rota de um determinado recurso, dependendo do verbo *HTTP* utilizado, a ação executada é diferente. Este nível também implica o uso de códigos *HTTP* e respeita as convenções do protocolo;
- **Nível 3**: o último nível implica o uso de *Hypermedia As The Engine Of Application State (HATEOAS)* que facilita a navegação da *API*. Este sistema atua como um motor de navegação construído dinamicamente com base nos pedidos efetuados por um utilizador. A ideia deste sistema é juntar à mensagem de retorno links que indicam o que se pode ou não fazer e estejam relacionados com o pedido realizado [4].
  O extrato de Código 2 apresenta um exemplo do uso de *HATEOAS* onde o utilizador pede informação sobre um carro. Juntamente com essa informação é retornado um *URI* onde ele pode encomendar o carro.

```
{
  "data": {
    "id": 3,
    "marca": "Tesla",
    "modelo": "Model",
    "preço": "35 000 USD",
    "links": [
      {
        "rel": "encomendar",
        "href": "/encomendar/3"
      }
    ]
  }
}
```

Código 2 - Exemplo de uma mensagem de retorno que faz uso de *HATEOAS*

2.3.2 SOAP

SOAP é uma série de convenções da W3C que definem um protocolo de mensagens para traduzir a informação numa mensagem de *web service* antes de a transferir pela *internet* [28]. Para além de ser um estilo arquitetural, *SOAP* é também um protocolo de transporte.

As principais características deste protocolo são simplicidade e extensibilidade [6]. Segundo a convenção, as mensagens *SOAP* devem ser codificadas em *XML* e têm de incluir os seguintes elementos:

- ***SOAP Envelope:*** é o elemento no topo da mensagem *XML* e têm de conter a seguinte informação:
    - Um nome local: "Envelope";
    - Um nome de *namespace*: "https://www.w3.org/2003/05/soap-envelope/";
    - Zero ou mais itens de qualificação de *namespace* na propriedade *attributes*;
    - Um elemento filho *SOAP Body* e opcionalmente um elemento *SOAP Header*.
- ***SOAP Body:*** é obrigatoriamente filho do elemento *SOAP Envelope* e deve conter a seguinte informação:
    - Um nome local: "*Body*";
    - Um nome de *namespace*: "https://www.w3.org/2003/05/soap-envelope/";
    - Zero ou mais itens de qualificação de *namespace* na propriedade *attributes*;
    - Zero ou mais itens de qualificação de *namespace* entre a propriedade *children*;
    - Elementos filho a representar a estrutura dos dados de entrada.

A mensagem deve ainda de conter elemento opcional *SOAP Header.* Os seus elementos filho são passados para o recetor, mas sem acordo prévio entre ambos os lados. A convenção *SOAP* define alguns elementos filho de *SOAP Header* que podem ser usados e a forma como devem ser processados.

Para além do elemento *SOAP Header* a convenção define também o elemento *SOAP Fault* que é usado para transportar informação relativa a erros.

A Figura 5 apresenta de forma resumida o conteúdo de uma mensagem *SOAP*, com os elementos referidos anteriormente.

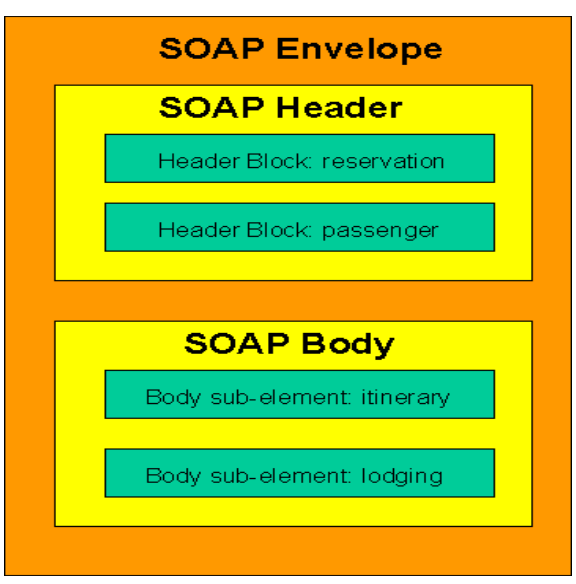


Figura 5 - Representação de uma mensagem *SOAP* [29]

O exemplo seguinte mostra uma mensagem *SOAP* a definir uma mensagem de notificação, como exemplificado na convenção [28]:

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-
envelope">
 <env:Header>
  <n:alertcontrol xmlns:n="http://example.org/alertcontrol">
   <n:priority>1</n:priority>
   <n:expires>2001-06-22T14:00:00-05:00</n:expires>
  </n:alertcontrol>
 </env:Header>
 <env:Body>
  <m:alert xmlns:m="http://example.org/alert">
   <m:name>Pick up Mary at school at 2pm</m:name>
  </m:alert>
 </env:Body>
</env:Envelope>
```

Código 3 - Exemplo de uma mensagem *SOAP*

Tipicamente, para descrever os serviços disponibilizados, utiliza-se em simultâneo com *SOAP* a linguagem *Web Services Description Language (WSDL)*. Esta linguagem descreve serviços *web* a um nível abstrato onde é descrito ao nível das mensagens que recebe e envia. Existe também um nível concreto onde são descritos em detalhe os formatos de transporte e ligação [30].Para realizar este processo, são utilizados os seguintes elementos:

- ***Message:*** descreve o conteúdo de uma mensagem;
- ***PortType:*** especifica para uma determinada operação, as mensagens de entrada e resposta;
- ***Binding:*** este elemento possui os atributos *name* e *type* onde é indicado um "*portType"* correspondente. Dentro deste elemento utiliza-se o elemento *soap:binding* [31];
- ***SOAP Binding:*** este elemento possui os atributos *style* que pode ter o valor *RPC* ou *document*, e o elemento *transport* que especifica o protocolo de transporte utilizado (*e.g. HTTP);*
- ***Operation:*** define cada operação que o elemento *portType* expõe.

O extrato de Código 4 apresenta um exemplo de utilização da linguagem *WSDL*. Este exemplo descreve a uma ação e os argumentos de entrada e de resposta, assim como os elementos previamente referidos.

```
<message name="getNameRequest">
  <part name="term" type="xs:string"/>
</message>
<message name="getNameResponse">
  <part name="value" type="xs:string"/>
</message>
<portType name="nameTerms">
  <operation name="getName">
    <input message=" getNameRequest "/>
    <output message=" getNameResponse "/>
  </operation>
</portType>
<binding type=" nameTerms " name="b1">
   <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http" />
   <operation>
     <soap:operation
soapAction="http://example.com/getName"/>
     <input><soap:body use="literal"/></input>
     <output><soap:body use="literal"/></output>
  </operation
</binding>
```

Código 4 - Exemplo de WSDL

2.3.3 GraphQL

GraphQL é uma tecnologia emergente criada pelo Facebook em 2012 para descrever capacidades e requisitos de modelos de dados em aplicações cliente-servidor. O desenvolvimento começou em 2015 e ainda se encontra ativo [32]. Por esse motivo *GraphQL* ainda não é muito popular [2].

Esta tecnologia segue os seguintes princípios de design [32]:

- **Hierárquico:** as *queries* são estruturadas hierarquicamente;
- **Centrado no produto:** o desenvolvimento é centrado nas necessidades dos engenheiros *front-end;*
- **Linguagem fortemente tipificada**: cada servidor *GraphQL* define tipos especificados na aplicação e as *queries* são executadas no contexto desses tipos;
- ***Queries* especificadas pelo cliente:** os servidores *GraphQL* publicam as capacidades que os clientes podem consumir. É o cliente que tem a responsabilidade de especificar como vai consumir essas capacidades;
- **Introspetivo:** um servidor de *GraphQL* pode ser especificado e possibilitar executar *queries* utilizando *GraphQL*.

Cada servidor *GraphQL* têm as seguintes componentes chave que descrevem o seu funcionamento [33]:

- ***Schema***: define que *queries* podem ser feitas pelo cliente, os tipos de dados que existem e as suas relações;
- ***Resolve Functions:*** especificam como os tipos de dados e os seus atributos estão conectados aos vários *backends* (*e.g.* base de dados, micro serviços). As resolve function de *GraphQL* podem conectar-se a qualquer tipo de *backend*.

A tecnologia *GraphQL* permite o envio de *queries* para obter os dados necessários da *API* sem a necessidade de fazer múltiplos pedidos às rotas disponibilizadas, como no caso de *REST.* Por exemplo, numa *API* que guarda uma lista de contactos e os seus amigos, se um utilizador quiser os nomes dos amigos do contacto cujo id é “1”, numa *API REST* teria de fazer algo semelhante ao seguinte:

- Aceder ao contacto cujo id é “1”: “*GET*: <*URL* da *API REST*>/contacts/1”*;*
- Para cada um dos *ids* dos amigos, retornado na resposta, fazer um pedido semelhante ao seguinte: “*GET*: <*URL* da *API REST*>/contacts/1/friends/<id de um amigo>”*;*
- Para cada uma das respostas guardar o atributo nome.

Essa solução obriga a executar múltiplos pedidos. No entanto em *GraphQL*, para obter um resultado semelhante, bastaria fazer a *query* apresentada no extrato de Código 5:

```
{
  contacts(id : “1” ){
   friends {
    name
   }
  }
}
```

Código 5 - Exemplo de uma *query GraphQL*

No estilo arquitetural *REST,* é apenas possível passar um único conjunto e argumentos, no entanto em *GraphQL* cada campo e cada objeto filho ou os seus filhos (hierarquicamente) pode ter os seus próprios parâmetros. Esta característica faz com que *GraphQL* seja um substituto de múltiplos pedidos a uma *API* [34].

GraphQL funciona sobre qualquer protocolo de transporte (*e.g. HTTP*, *TCP*, *WebSockets*) e serve como uma camada que se pode ligar a uma base de dados. Esta tecnologia também pode ser utilizada para integrar diversos sistemas sobre uma única *API*. Alternativamente é possível utilizar *GraphQL* de forma híbrida (ilustrado na Figura 6), conectando a *API* a uma base de dados própria e a outros sistemas [35].

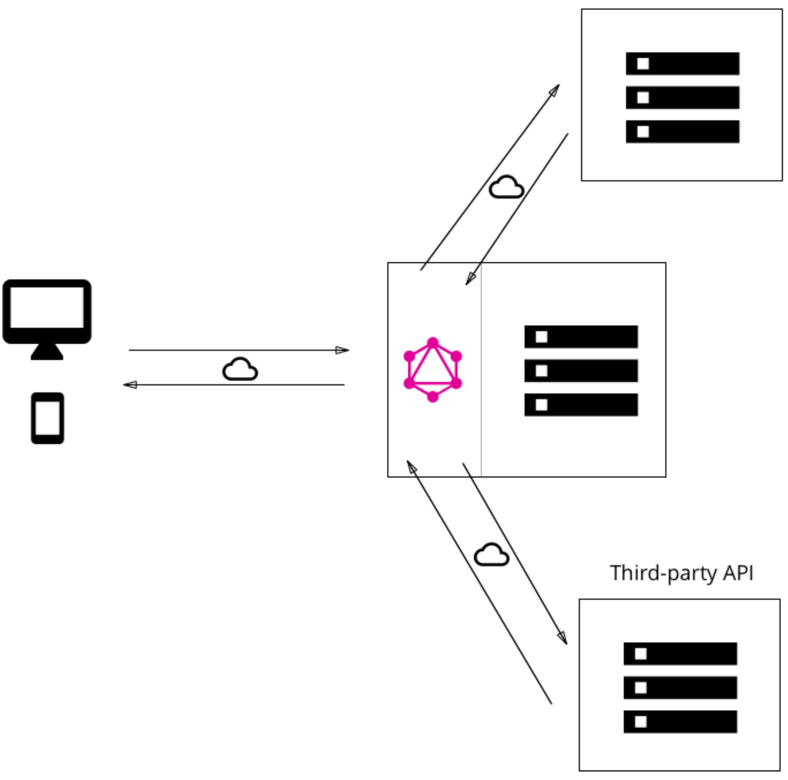


Figura 6 - Utilização de *GraphQL* de forma híbrida com uma base de dados e integrando um sistema existente [35]

2.3.4 Comparação entre os paradigmas das API

Ao contrário do estilo arquitetural *REST, SOAP* é um protocolo. Por exemplo, uma *API SOAP* pode utilizar o protocolo de transporte *STOMP* tal como pode utilizar o protocolo *HTTP* ou qualquer outro. Por este motivo não faz sentido comparar diretamente *SOAP* e *GraphQL* com *REST*. No entanto, uma vez que no contexto desta dissertação, ambos usarão como protocolo de transporte o protocolo *HTTP* e servem propósitos semelhantes, serão ao longo deste documento descritos como estilos arquiteturais. Uma vez que servem propósitos idênticos, é possível uma comparação entre algumas das características de cada um. A Tabela 3 apresenta e forma resumida as diferenças entre *REST*, *SOAP* e *GraphQL*.

Tabela 3 - Comparação entre *REST* e *SOAP e GraphQL*

Elemento	***REST***	***SOAP***	***GraphQL***
Protocolo de transporte	*HTTP*	Nenhum em específico (1)	Nenhum em específico, HTTP é a escolha habitual

Elemento	***REST***	***SOAP***	***GraphQL***
Linguagem descritiva da estrutura de dados	Nenhuma em específico (nota 1)	*XML* (*SOAP* Envelope)	*GraphQL* (nota *2)*
Representa	Recursos	Procedimentos	Entidades
Rotas	Múltiplas	Opcional (nota 3)	Única
Definição dos dados retornados	Do lado do servidor	Do lado do servidor	Do lado do cliente
Caching	Sim	Não	Sim [36]
Versionamento	Sim	Sim	Sim (nota 4)

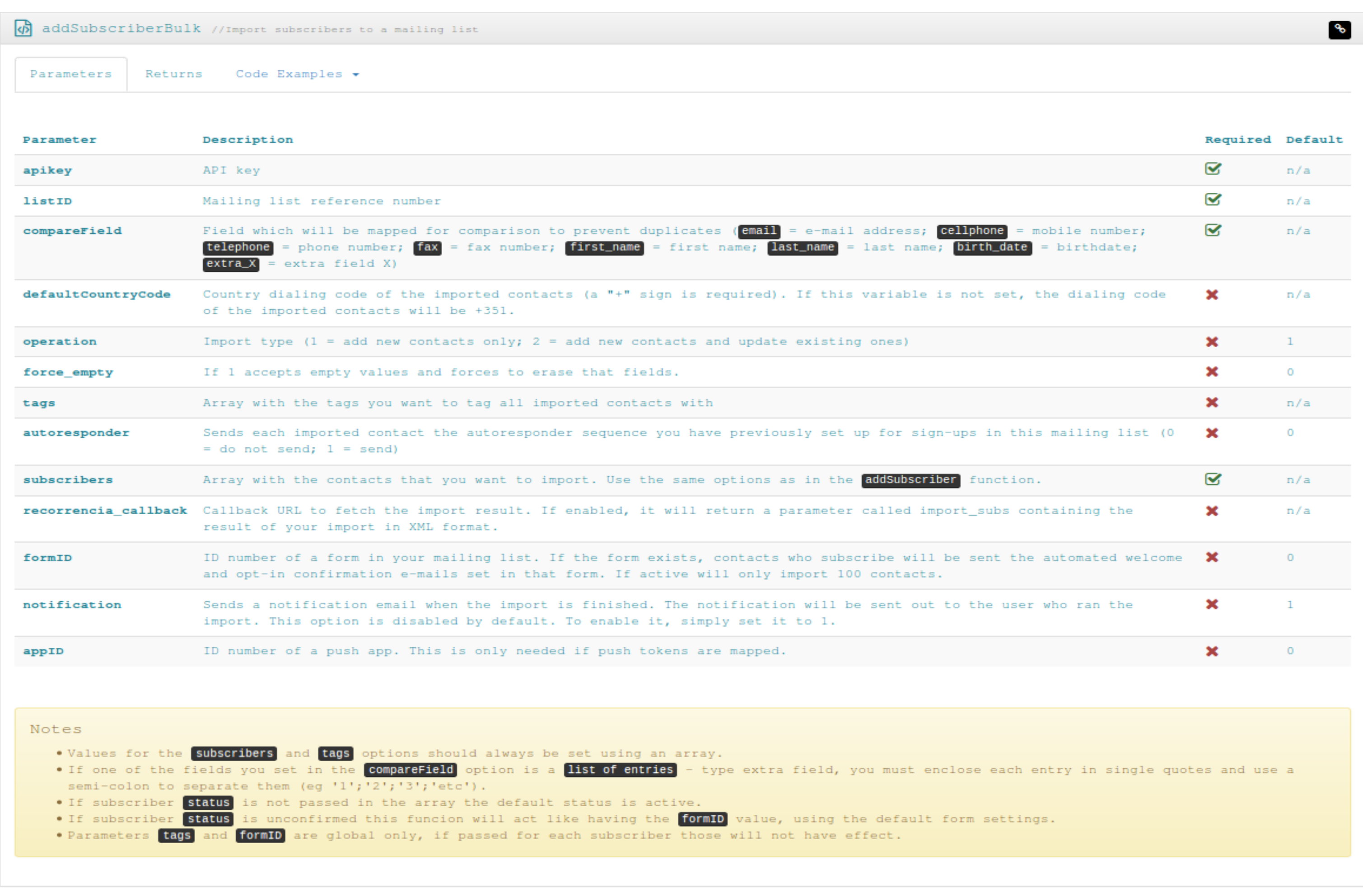
Relativamente à Tabela 3, algumas notas devem ser tidas em consideração:

1. No contexto desta dissertação, a linguagem utilizada será *JSON*;
2. O retorno é em o resultado de uma serialização em *JSON* de um *Map;*
3. Os serviços podem ser disponibilizados na mesma rota, sendo que a escolha do serviço é realizada através da informação no *SOAP Envelope* ou disponibilizados em rotas diferentes [37];
4. No entanto não é necessária uma vez que em *GraphQL* o versionamento é automático devido à forma como este funciona [38].

Relativamente à popularidade de cada um, segundo um estudo realizado com base na base de dados das *API* registadas do site *ProgrammableWeb*[2], até 26 de novembro de 2017. *REST* é o mais utilizado, com cerca de 81.53% das *API* a seguirem este estilo arquitetural. *REST* e *SOAP* em conjunto com *Remote Procedure Call* (*RPC*) representa cerca de 9.36% das *API* listadas. *GraphQL* representa cerca de 0.04% das *API* [2], o que se deve ao facto de ser uma tecnologia recente.

[2] *URL* do *site* ProgrammableWeb: www.programmableweb.com

Embora seja possível utilizar os estilos arquiteturais *REST*, *SOAP* e *GraphQL* em simultâneo, dada a baixa taxa de utilização atual, não é desejável para a E-goi disponibilizar no contexto deste projeto essas funcionalidades. No entanto, no futuro, com o aumento da taxa de utilização de *GraphQL*, poderá fazer mais sentido criar esse módulo e é por esse motivo que esta análise foi realizada. Dessa forma no futuro, a *API* pode passar a disponibilizar funcionalidades em *REST*, *SOAP* e *GraphQL*. Por esse motivo, na secção 4.3.1.1 apresentam-se algumas alternativas arquiteturais de como esse módulo poderia ser eventualmente integrado neste projeto.



addSubscriberBulk //Import subscribers to a mailing list

Parameters | Returns | Code Examples

Parameter	Description	Required	Default
apikey	API key	✔	n/a
listID	Mailing list reference number	✔	n/a
compareField	Field which will be mapped for comparison to prevent duplicates (email = e-mail address; cellphone = mobile number; telephone = phone number; fax = fax number; first_name = first name; last_name = last name; birth_date = birthdate; extra_X = extra field X)	✔	n/a
defaultCountryCode	Country dialing code of the imported contacts (a "+" sign is required). If this variable is not set, the dialing code of the imported contacts will be +351.	✖	n/a
operation	Import type (1 = add new contacts only; 2 = add new contacts and update existing ones)	✖	1
force_empty	If 1 accepts empty values and forces to erase that fields.	✖	0
tags	Array with the tags you want to tag all imported contacts with	✖	n/a
autoresponder	Sends each imported contact the autoresponder sequence you have previously set up for sign-ups in this mailing list (0 = do not send; 1 = send)	✖	0
subscribers	Array with the contacts that you want to import. Use the same options as in the addSubscriber function.	✔	n/a
recorrencia_callback	Callback URL to fetch the import result. If enabled, it will return a parameter called import_subs containing the result of your import in XML format.	✖	n/a
formID	ID number of a form in your mailing list. If the form exists, contacts who subscribe will be sent the automated welcome and opt-in confirmation e-mails set in that form. If active will only import 100 contacts.	✖	0
notification	Sends a notification email when the import is finished. The notification will be sent out to the user who ran the import. This option is disabled by default. To enable it, simply set it to 1.	✖	1
appID	ID number of a push app. This is only needed if push tokens are mapped.	✖	0

Notes

- Values for the subscribers and tags options should always be set using an array.
- If one of the fields you set in the compareField option is a list of entries - type extra field, you must enclose each entry in single quotes and use a semi-colon to separate them (eg '1';'2';'3';'etc').
- If subscriber status is not passed in the array the default status is active.
- If subscriber status is unconfirmed this funcion will act like having the formID value, using the default form settings.
- Parameters tags and formID are global only, if passed for each subscriber those will not have effect.

Figura 7 - Argumentos de entrada do serviço *addSubscriberBulk* da versão 2 da *API*

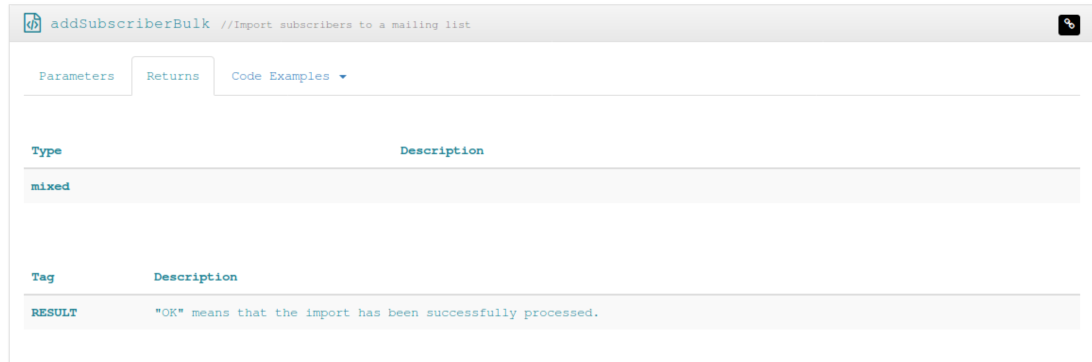


Figura 8 - Argumentos de resposta do serviço *addSubscriberBulk* da versão 2 da *API*

Um dos principais problemas identificados é documentação dos serviços ser confusa e existirem regras complexas para os argumentos dos serviços. A Figura 7 apresenta a documentação do serviço “*addSubscriberBulk*” e ilustra este problema, uma vez que existem diversas condições que podem causar confusão aos utilizadores. Por exemplo, no argumento “*compareField*” existe a seguinte condição: "I*f one of the fields you set in the compareField option is a list of entries - type extra field, you must enclose each entry in single quotes and use a semi-colon to separate them*" [39].

A estrutura da resposta apresentada na Figura 8, está incompleta (apenas diz *“Mixed”*) pelo que o utilizador não tem como saber qual é seu formato sem efetuar o pedido. Outro problema é que não são utilizados códigos de erro do protocolo *HTTP*. Em vez de isso existe uma lista de erros documentada em separado dos serviços, e a sua respetiva descrição. Os problemas descritos ocorrem em muitos dos serviços disponibilizados e motivam insatisfação e confusão nos utilizadores.

Outro problema nesta *API* é que a forma como os parâmetros de *query* são passados é confusa e não existe nenhuma forma de obter essa informação na documentação a não ser analisando um exemplo de pedido, caso esteja disponível para o serviço pretendido, ou a forma como os *SDKs* disponibilizados efetuam o pedido. Por exemplo, o serviço *getAutobots* necessita do argumento obrigatório *“apikey”* e pode receber opcionalmente: *“listID”*; *“start”*; *“limit”*. Para realizar este pedido é necessário efetuar um pedido com os seguintes argumentos:

- *URL* base da *API REST*;
- *apikey*: pode ser enviada num *HTTP header* ou no URL como parâmetro de *query*.

Os argumentos opcionais podem ser enviados como parâmetros de *query*, no entanto o seu formato deve o seguinte: *“functionOptions[*<Nome do argumento>*]=*<Valor do argumento>*”.* Por exemplo, para enviar o argument *“listID”* com o valor 26, a estrutura seria

"functionOptions[listID]=26". Esta estrutura não faz muito sentido uma vez que poderia ser enviado no formato mais simples: "*listID=26*". No caso de *HTTP POST* ou *HTTP PUT* os argumentos de entrada devem seguir o mesmo formato, mas podem ser enviados através *HTTP* Body ou como argumentos de *query* no *URL*.

Na *API SOAP* é possível utilizar as estruturas de dados *JSON* e *XML,* o que vai contra as regras definidas nas recomendações do W3C para *SOAP.* Os *SOAP Envelope* devem ser definidos em *XML* [6], pelo que esta implementação está incorreta.

Para além dos problemas com a documentação e estruturação dos recursos e rotas, ficou concluído após reuniões com responsáveis da empresa que o próprio projeto da *API* se encontra mal estruturado, o que torna a sua manutenção e expansão complicados. Foi ainda falado que o projeto utiliza tecnologias que já estão desatualizadas e possui problemas de performance, que causaram reclamações de clientes.

O projeto atual está dividido em 10 ficheiros PHP, sendo os de maior relevo os seguintes:

- ***API:*** este ficheiro contém a implementação de todos os serviços disponíveis na *API.* É neste ficheiro que ocorre a comunicação direta com a base de dados. Possui também a responsabilidade de chamar o serviço pretendido com base nos argumentos recebidos;
- ***REST:*** recebe os pedidos *REST* e reencaminha para o ficheiro *API.php*;
- ***SOAP:*** envia a informação do pedido *SOAP* para o ficheiro *API.php*;
- ***xmlRPC*:** este ficheiro transfere os dados contidos no pedido *XML-RPC* o ficheiro *API.php*;
- ***Error*:** contém métodos responsáveis por gerar mensagens de erro.

A estrutura da *API* não possui uma separação de responsabilidades suficiente o que aumenta o acoplamento e baixa a coesão do projeto. Por causa deste problema é difícil alterar ou acrescentar funcionalidades. O ficheiro *API.php* possui mais de 16000 *linhas* e está programado seguindo uma lógica procedimental. Este problema torna difícil encontrar o local onde está o serviço que se pretende alterar, e depois de encontrado complica a sua alteração, pela possibilidade de afetar outros serviços que não são diretamente relacionados.

2.4 Versão 2 da API a substituir

A análise aqui apresentada foi realizada do ponto de vista de um utilizador que consulta a documentação e tenta utilizar a *API* e através de reuniões com as pessoas responsáveis na E-goi.

A *API* anterior suporta o estilo arquitetural *REST* e os protocolos *SOAP e XML-RPC* [39] [3], através de URL diferentes (*e.g.* para aceder à *API RESTful*, a rota utilizada é http://api.e-goi.com/v2/rest.php). Nesta *API* são disponibilizados cinquenta e seis serviços, agrupados nos recursos: *Accounts; Automations; Campaigns; Common; Forms; Lists; Reports; Segments; Senders; Subscribers; Tags*. A Figura 9 mostra os serviços do recurso *Subscribers* como disponível na documentação e permite identificar alguns problemas.

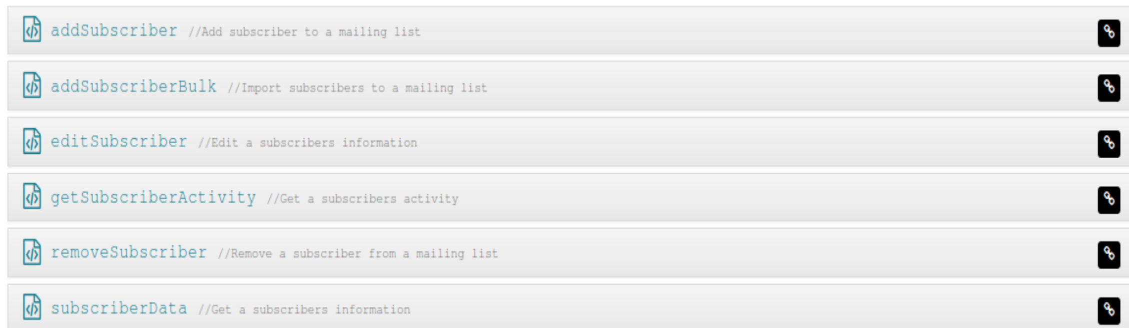
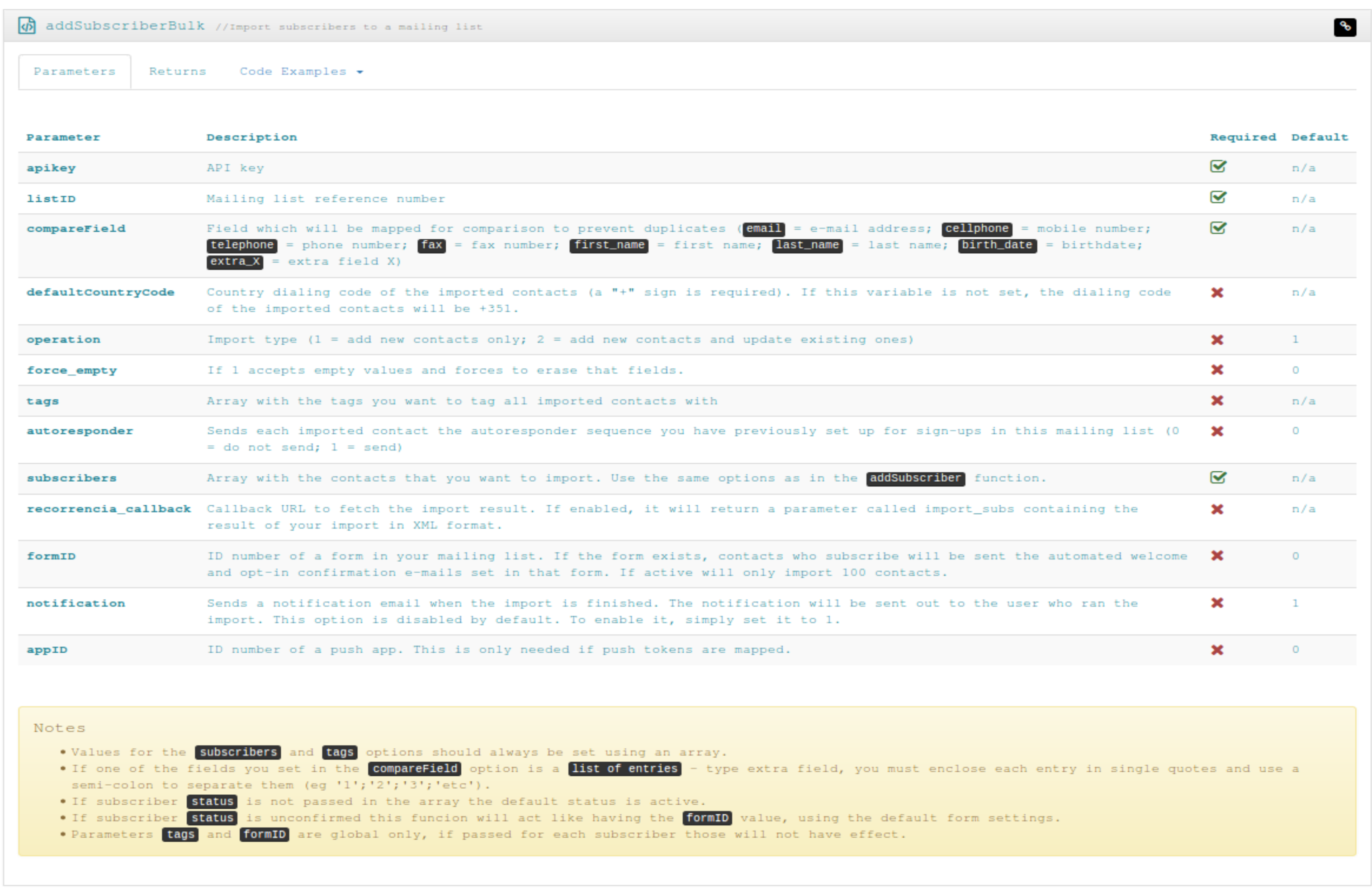


Figura 9 - Serviços do recurso *subscriber* da versão 2 da *API*

A estrutura apresentada não é compatível com convenções tipicamente associados com o estilo arquitetural *REST*, uma vez que o utilizador deve indicar o nome de um serviço (*e.g. addSubscriber*) em vez de um recurso e um verbo *HTTP* para descrever a ação pretendida [4]. Uma forma mais *RESTful* de disponibilizar o serviço "*addSubscriber"* seria através de um *HTTP POST* para a rota "*<URL* da *API REST>/Subscriber"*.

A Figura 7 e a Figura 8 apresentam respetivamente a documentação dos argumentos de entrada e de resposta do serviço "*addSubscriberBulk"*. Este serviço é exemplificativo de como por vezes a *API* é complicada de se utilizar.

[3] URL da documentação da versão 2 da API pública da E-goi: https://api-docs.e-goi.com/


addSubscriberBulk //Import subscribers to a mailing list

Parameters | Returns | Code Examples

Parameter	Description	Required	Default
apikey	API key	✔	n/a
listID	Mailing list reference number	✔	n/a
compareField	Field which will be mapped for comparison to prevent duplicates (email = e-mail address; cellphone = mobile number; telephone = phone number; fax = fax number; first_name = first name; last_name = last name; birth_date = birthdate; extra_X = extra field X)	✔	n/a
defaultCountryCode	Country dialing code of the imported contacts (a "+" sign is required). If this variable is not set, the dialing code of the imported contacts will be +351.	✖	n/a
operation	Import type (1 = add new contacts only; 2 = add new contacts and update existing ones)	✖	1
force_empty	If 1 accepts empty values and forces to erase that fields.	✖	0
tags	Array with the tags you want to tag all imported contacts with	✖	n/a
autoresponder	Sends each imported contact the autoresponder sequence you have previously set up for sign-ups in this mailing list (0 = do not send; 1 = send)	✖	0
subscribers	Array with the contacts that you want to import. Use the same options as in the addSubscriber function.	✔	n/a
recorrencia_callback	Callback URL to fetch the import result. If enabled, it will return a parameter called import_subs containing the result of your import in XML format.	✖	n/a
formID	ID number of a form in your mailing list. If the form exists, contacts who subscribe will be sent the automated welcome and opt-in confirmation e-mails set in that form. If active will only import 100 contacts.	✖	0
notification	Sends a notification email when the import is finished. The notification will be sent out to the user who ran the import. This option is disabled by default. To enable it, simply set it to 1.	✖	1
appID	ID number of a push app. This is only needed if push tokens are mapped.	✖	0

Notes

- Values for the subscribers and tags options should always be set using an array.
- If one of the fields you set in the compareField option is a list of entries - type extra field, you must enclose each entry in single quotes and use a semi-colon to separate them (eg '1';'2';'3';'etc').
- If subscriber status is not passed in the array the default status is active.
- If subscriber status is unconfirmed this funcion will act like having the formID value, using the default form settings.
- Parameters tags and formID are global only, if passed for each subscriber those will not have effect.

Figura 10 - Argumentos de entrada do serviço *addSubscriberBulk* da versão 2 da *API*

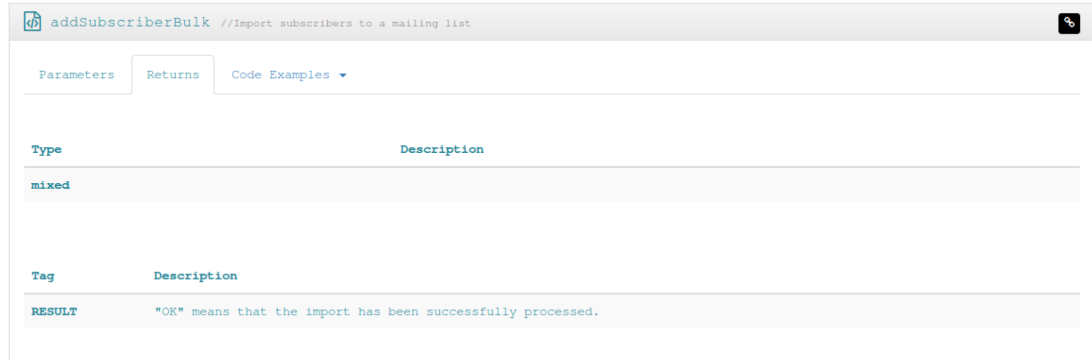


Figura 11 - Argumentos de resposta do serviço *addSubscriberBulk* da versão 2 da *API*

Um dos principais problemas identificados é a documentação dos serviços ser confusa e existirem regras complexas para os argumentos dos serviços. A Figura 7 apresenta a documentação do serviço "*addSubscriberBulk*" e ilustra este problema, uma vez que existem diversas condições que podem causar confusão aos utilizadores. Por exemplo, no argumento "*compareField*" existe a seguinte condição: "*If one of the fields you set in the compareField option is a list of entries - type extra field, you must enclose each entry in single quotes and use a semi-colon to separate them*" [39].

A estrutura da resposta apresentada na Figura 8 está incompleta (apenas diz *"Mixed"*) pelo que o utilizador não tem como saber qual é seu formato sem efetuar o pedido. Outro problema é que não são utilizados códigos de erro do protocolo *HTTP*. Em vez de isso existe uma lista de erros documentada em separado dos serviços, e a sua respetiva descrição. Os problemas descritos ocorrem em muitos dos serviços disponibilizados e motivam insatisfação e confusão nos utilizadores.

Outro problema nesta *API* é que a forma como os parâmetros de *query* são passados é confusa e não existe nenhuma forma de obter essa informação na documentação a não ser analisando um exemplo de pedido, caso esteja disponível para o serviço pretendido, ou a forma como os *SDKs* disponibilizados efetuam o pedido. Por exemplo, o serviço *getAutobots* necessita do argumento obrigatório "*apikey*" e pode receber opcionalmente: "*listID*"; "*start*"; "*limit*". Para realizar este pedido é necessário efetuar um pedido com os seguintes argumentos:

- *URL* base da *API REST*;
- *apikey*: pode ser enviada num *HTTP header* ou no URL como parâmetro de *query*.

Os argumentos opcionais podem ser enviados como parâmetros de *query*, no entanto o seu formato deve o seguinte: "*functionOptions[*<Nome do argumento>*]=*<Valor do argumento>*"*. Por exemplo, para enviar o argument "*listID*" com o valor 26, a estrutura seria

"functionOptions[listID]=26". Esta estrutura não faz muito sentido uma vez que poderia ser enviado no formato mais simples: "*listID=26*". No caso de *HTTP POST* ou *HTTP PUT* os argumentos de entrada devem seguir o mesmo formato, mas podem ser enviados através *HTTP* Body ou como argumentos de *query* no *URL*.

Para além dos problemas com a documentação e estruturação dos recursos e rotas, ficou concluído após reuniões com responsáveis da empresa que o próprio projeto da *API* se encontra mal estruturado, o que torna a sua manutenção e expansão complicados. Foi ainda falado que o projeto utiliza tecnologias que já estão desatualizadas e possui problemas de performance, que causaram reclamações de clientes.

O projeto atual está dividido em 10 ficheiros PHP, sendo os de maior relevo os seguintes:

- ***API:*** este ficheiro contém a implementação de todos os serviços disponíveis na *API.* É neste ficheiro que ocorre a comunicação direta com a base de dados. Possui também a responsabilidade de chamar o serviço pretendido com base nos argumentos recebidos;
- ***REST:*** recebe os pedidos *REST* e reencaminha para o ficheiro *API.php*;
- ***SOAP:*** envia a informação do pedido *SOAP* para o ficheiro *API.php*;
- ***xmlRPC***: este ficheiro transfere os dados contidos no pedido *XML-RPC* o ficheiro *API.php*;
- ***Error***: contém métodos responsáveis por gerar mensagens de erro.

A estrutura da *API* não possui uma separação de responsabilidades suficiente o que aumenta o acoplamento e baixa a coesão do projeto. Por causa deste problema é difícil alterar ou acrescentar funcionalidades. Esta solução segue o *antipattern multiservice*, devido ao ficheiro *API.php* expor todas entidades [20]. O ficheiro *API.php* possui mais de 16000 *linhas* e está programado seguindo uma lógica procedimental. Este problema torna difícil encontrar o local onde está o serviço que se pretende alterar, e depois de encontrado complica a sua alteração, pela possibilidade de afetar outros serviços que não estão diretamente relacionados.

3 Análise de Valor

Neste capítulo será realizada uma análise de valor relativa à *API* a desenvolver. Para tal o capítulo foi dividido nas seguintes secções: Processo de negócio e inovação; Proposta de valor; Valor para o cliente; Modelo de negócio Canvas; Cadeia de valor de Porter; Modelo AHP.

3.1 Processo de Negócio e Inovação

A inovação em modelos de negócio é uma metodologia para implementar inovação no processo de negócio. Esta inovação é no fundo sobre criar valor, seja para os clientes, para a empresa ou para a sociedade em geral [8]. O processo de inovação de um produto (apresentado na Figura 12) passa por três fases distintas.

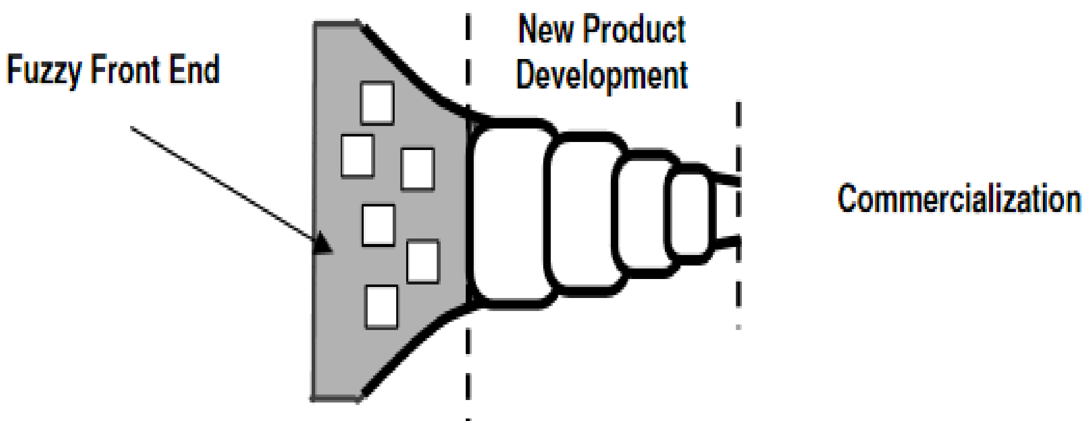


Figura 12 - Processo de inovação de um produto [40]

As três fases do processo de inovação de um produto são:

1. ***Fuzzy Front End*** (***FFE***): esta fase é constituída por atividades geralmente caóticas e com data de comercialização incerta ou imprevisível. Esta fase é geralmente tida como uma das maiores oportunidades para melhorar o processo geral de inovação;
2. ***New Product Development*** (***NPD)***: esta fase, por contraste com a anterior, é um processo mais disciplinado e orientado a objetivos sendo possível indicar uma data de comercialização do produto;
3. **Comercialização:** a última fase é a comercialização do produto [40].

O *New Concept Development* [7] é um modelo que foi criado de forma a poder comparar a fase inicial do processo de inovação. A Figura 13 apresenta a estrutura do modelo NCD. As cinco partes interiores são elementos e não processos. A forma circular é para representar que é esperado que as ideias circulem entre os cinco elementos, em qualquer ordem ou combinação. Esta aleatoriedade é representada pelas setas interiores [7].

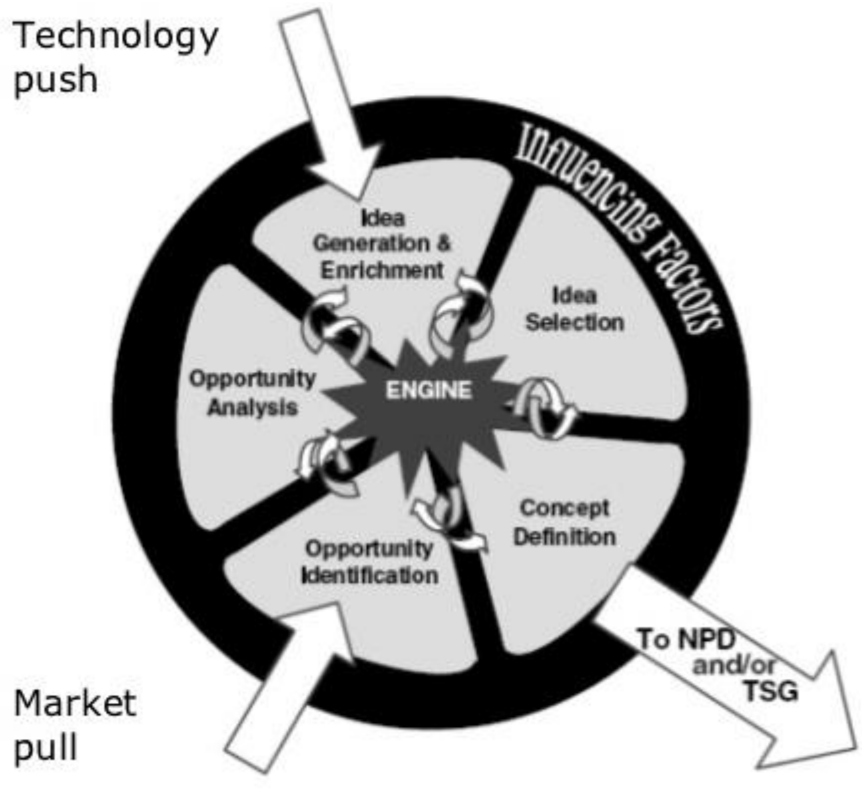


Figura 13 - O modelo NCD produto [40]

O modelo *NCD* atua na fase *FFE* e é constituído pelas seguintes três partes [7]:

1. As áreas internas definem os 5 elementos constituintes do *Front End of Innovation* (*FEI*). Estes elementos são a géneses da ideia, a seleção da ideia, o desenvolvimento do conceito e da tecnologia, a identificação de oportunidades e análise de oportunidades. Para cada uma destas fases existem diversos métodos que podem ser usados;
2. O motor que move os 5 elementos e é impulsionada pela liderança e cultura da organização;
3. Os fatores externos ou ambiente que influenciam a organização. São constituídos pela estratégia de negócio, o ambiente externo, as capacidades da organização e o estado atual da ciência e tecnologia.

Os cinco elementos do modelo NCD são os seguintes [40]:

- **Identificação de oportunidades:** a empresa identifica as oportunidades que quer aproveitar. Para realizar esta identificação a empresa pode utilizar os seguintes métodos: *roadmapping*; análise da tendência de tecnologias; análise das tendências dos clientes; análise de inteligência competitiva; pesquisa de mercado. No caso deste projeto, para a criação da nova *API* pública da empresa E-goi, verifica-se que a tendência dos estilos arquiteturais mais utilizados (como pode ser visto na Figura 14) mostra que *REST* é o mais usado com 81.53% e que *SOAP* e *RPC* juntos 9.36%. O estudo de onde vieram estes números foi realizado em 2017 por Wendell Santos [2]. Nesse estudo foram analisadas 18279 *API* registadas no site ProgrammableWeb[4].

[4] *URL* do *site* ProgrammableWeb: www.programmableweb.com

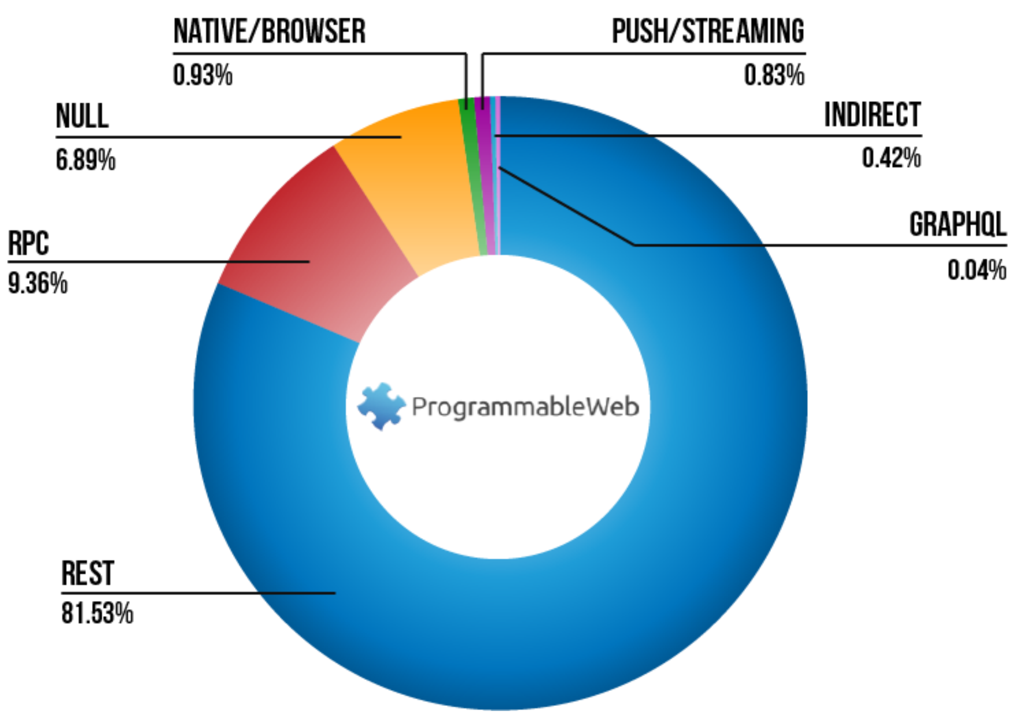


Figura 14 - Percentagem de utilização dos estilos arquiteturais das *API* registadas no site ProgrammableWeb [2]

No caso deste projeto, identifica-se que a API pública versão 2 da E-goi, se encontra desatualizada tecnologicamente e não segues as boas práticas do estilo arquitetural *REST*, pelo que se identifica a oportunidade de ter um cuidado especial na implementação do *REST*, uma vez que este estilo é o mais utilizado;

- **Análise de oportunidade:** neste ponto, a empresa verifica se as oportunidades identificadas valem ou não a pena. Tal pode ser realizado usando os mesmos métodos utilizados na identificação de oportunidades.

  Neste projeto analisa-se que é necessário criar uma nova versão da API para resolver os problemas identificados, dando especial importância ao estilo arquitetural *REST*, mas também faz sentido suportar *SOAP*, dado que a sua taxa de utilização é considerável;

- **Geração e aperfeiçoamento de ideias:** neste elemento ocorre a geração e enriquecimento das ideias. Para ajudar neste processo podem ser usados os seguintes métodos: métodos para identificar possíveis clientes; métodos para identificação de ovas soluções tecnológicas; uma variedade de incentivos para estimular ideias.

Após diversas reuniões com as partes interessadas, apareceram as seguintes ideias para a criação da nova *API* pública da empresa E-goi:

  - Fazer 2 projetos diferentes, um para *REST* e outro para *SOAP*;
  - Fazer apenas um projeto que suporte *REST*;
  - Fazer um projeto que suporta *REST* e *SOAP,* aproveitando as partes comuns;

- **Seleção de ideias:** nesta fase ocorre a seleção das melhores ideias. Tal pode ser realizado usando métodos como: portefólio de metodologias, baseado em múltiplos fatores; seleção de ideias a partir de feedback; usar a teoria de opções para avaliar. No caso deste projeto, os critérios relevantes para a escolha da melhor solução são:
  - Escalabilidade do projeto: o quão escalável é a solução desenvolvida;
  - Reaproveitamento de código: quanto código se pode reaproveitar entre a *API* de *REST* e de *SOAP*;
  - Suporte dos estilos arquiteturais *REST* e *SOAP:* indicar se a solução suporta ambos os estilos arquiteturais;

  Verificou-se que a melhor opção é criar um projeto que suporta os estilos arquiteturais *REST* e *SOAP*, uma vez que esta é a melhor solução, com bases nos critérios definidos. Este processo de seleção pode ser visto em detalhe na secção 3.6 Decisão utilizando o modelo AHP;

- **Desenvolvimento do Conceito e da Tecnologia:** este é o elemento final e envolve a definir os passos necessários para que a solução tenha sucesso. Tal pode ser feito usando os seguintes métodos: definição de objetivos; avaliação rápida de inovações de alto potencial; envolvimento precoce dos clientes.

  Desde o inicio do desenvolvimento deste projeto existiu envolvimento com as partes interessadas, de forma a que o produto desenvolvido fosse ao encontro das necessidades da empresa.

3.2 Proposta de Valor

A proposta de valor é uma promessa explícita, feita pela empresa aos seus clientes, de que esta lhe vai entregar um conjunto particular de benefícios que criam valor [41]. Segundo Paul Fifield, uma definição para proposta de valor é "uma declaração por escrito que foca todas as atividades de marketing da organização em elementos críticos do cliente, que criam um diferencial significativo no seu processo de decisão, para este preferir e/ou comprar a oferta da organização em vez de a de um concorrente" [42]. Por outras palavras, a proposta de valor é a promessa e explicação de uma empresa, que o seu produto é a melhor opção para suprir as necessidades do cliente.

A proposta de valor deste projeto é a criação de uma *API* pública para a empresa E-goi. Esta vai permitir que os seus clientes utilizem funcionalidades do E-goi, nos seus próprios projetos. Em vez de necessitarem de implementar as suas próprias versões destas funcionalidades, podem passar a depender desta *API*.

Esta *API* pode ser utilizada, usando o estilo arquitetural *REST* ou *SOAP* e foi pensada de forma a ser fácil de navegar, ao fazer uso de *HATEOAS*, mensagens de erro explicativas, códigos *HTTP* a seguir as convenções e ao ser bem documentada. Para ter acesso à *API* é necessário criar uma conta no E-goi de forma a obter uma *apikey*.

A *API* executa as operações pedidas e responde com os dados necessários. A partir da *API* pública do E-goi, o cliente dispõe de funcionalidades que lhe permitem realizar campanhas em múltiplos canais (*e.g. email*, *push* de notificações) e realizar a entrega de mensagens aos seus clientes de forma eficiente e eficaz. Este processo permite um aumento de eficiência do cliente e consequente poupança de tempo.

Os aspetos diferenciadores deste projeto em relação à versão 2 da *API* já existente são a correta aplicação do estilo arquitetural *REST* e a maior facilidade de navegação e utilização da *API*.

3.3 Valor para o cliente

Antes de fazer uma análise da proposta de valor deste projeto é necessário entender os conceitos de "valor", "valor para o cliente" e "valor percebido":

- **Valor:** o conceito de valor foi definido "em diferentes contextos teóricos como necessidade, desejo, interesse, critérios, crenças, atitudes e preferências" [43];

- **Valor para o cliente:** o conceito de valor para o cliente pode ser definido, segundo Woodruff (1997) como "a preferência percebida pelo cliente do produto, e avaliação dos atributos desse produto, performance desses atributos e consequências que advém do uso e que facilitam, ou bloqueiam, os clientes de atingir os o seus objetivos e propósitos nas situações de uso" [44].
  A pesquisa no valor percebido pelo cliente é construída na assunção que os clientes querem maximizar os benefícios percebidos e minimizar os sacrifícios percebidos [45];
- **Valor percebido:** O valor percebido pode ser definido, segundo Doyle como "o valor percebido de um produto consiste em três elementos: os benefícios oferecidos pela empresa do produto, menos o preço do produto e menos os outros custos de o usar ou possuir".
  A proposta de valor indica o conjunto de produtos (tangíveis) ou serviços (intangíveis) que criam valor para um segmento específico de clientes [46].

Segundo Woodall o valor divide-se nas seguintes fases [47]:

- **Antes da compra:** antes do cliente comprar o produto;
- **Durante a compra:** na altura da compra ou experimentação;
- **Depois da compra:** depois do cliente ter comprado o produto:
- **Após uso:** na altura do descarte.

Com base nestas definições, a Tabela 4 apresenta a proposta longitudinal de valor.

Tabela 4 - Proposta longitudinal de valor

	Benefícios	**Sacrifícios**
Antes da compra	Expectativa de aceder às funcionalidades do E-goi a partir de uma *API*;	Tempo despendido à procurada da melhor solução no mercado;
Durante a compra	Aquisição de uma boa solução para aceder às funcionalidades do E-goi a partir de uma *API* através de *REST* ou *SOAP*	Tempo despendido a entender como funciona a *API* e a ler a documentação
Depois da compra	Conhecimento total da *API* e das suas funcionalidades e de como interagir com ela	Dependência do E-goi uma vez que a API disponibiliza funcionalidades do E-goi
Após uso	Utilização de *SDKs* para abstrair das atualizações dos serviços	Ter de atualizar o seu software para utilizar uma nova API substituta desta

Para cada uma das quatro fases, existem benefícios e sacrifícios. No caso desta *API* verifica-se o seguinte:

1. **Antes da compra:** o cliente tem a expectativa de aceder às funcionalidades de marketing digital do e-goi utilizando uma *API*. No entanto teve de gastar tempo para procurar a solução que mais se adequa às suas necessidades no mercado;
2. **Durante a compra:** o cliente reconhece que a *API* lhe permite aceder às funcionalidades do E-goi (nos estilos arquiteturais *REST* e *SOAP)* e fica satisfeito. No entanto teve de gastar tempo a ler a documentação da *API* para a aprender a usar;
3. **Depois da compra:** o cliente possui um conhecimento de como utilizar as funcionalidades disponibilizadas pela *API.* No entanto teve de sacrificar possuir uma dependência do E-goi no seu projeto;
4. **Depois de usar**: o cliente pode utilizar *SDKs* para aceder à *API* e abstrair-se das alterações efetuadas nos serviços, atrasando ao máximo a necessidade de usar uma nova versão da *API*. No entanto caso a *API* seja descontinuada para ser criada uma nova versão, o cliente tem a necessidade de atualizar os seus softwares.

3.4 Modelo de Negócio Canvas

Face a uma ideia de negócio é necessário ponderar algumas questões. O modelo de negócio Canvas é uma ferramenta muito útil para responder a estas questões de forma organizada com definição do modelo de negócios de um qualquer projeto [8] . Este modelo está dividido nos seguintes elementos que devem ser preenchidos: segmentos de clientes; proposta de valor, canas; relacionamento com clientes; fontes de receita; recursos principais; atividades-chave; parecerias principais; estrutura de custo.

No contexto destes projetos, os elementos encontrados em cada ponto, foram os seguintes:

- **Segmentos de clientes**: clientes da E-goi que queiram integras as funcionalidades disponibilizadas pela empresa nos seus próprios projetos. Estes clientes têm de estar dispostos a ter uma dependência externa.
- **Proposta de valor**: como referido e detalhado na secção 3.2, a proposta de valor é a criação de uma *API* pública para a E-goi que disponibilize funcionalidades do E-goi em *REST* e *SOAP* e faça uso dos códigos *HTTP* corretos e de mensagens de erro adequadas;
- **Canais**:
    - A plataforma E-goi contém informação sobre a *API*;
    - A documentação da *API* é interativa e permite realizar pedidos à *API*;
    - O *blog* da E-goi vai publicitar a *API*;
    - A E-goi realiza *webinars* onde explica funcionalidades e-goi. Poderão existir *webinars* a falar sobre a *API*;
    - Anúncios sobre a *API.*
- **Relacionamento com clientes**:
    - *Help desk* da E-goi;
    - Longo prazo: o cliente e a empresa estabelecem uma relação de longo prazo e interagem de forma recorrente;
    - Serviços automatizados: a empresa disponibiliza serviços automáticos para o cliente.
- **Fontes de receita**: lucros provenientes do uso da *API* seja por ajudar a manter clientes satisfeitos ou por poder atrair clientes que só querem usar o E-goi por causa da *API*;

- **Recursos principais**:
  - Zend *framework* como *framework* de desenvolvimento;
  - Apache *ActiveMQ como sistema de filas;*
  - Outro software que seja necessário no desenvolvimento (*e.g. IDEs* de desenvolvimento, Jira.);
  - Hardware necessário para o desenvolvimento;
  - Hardware necessário para o deploy da *API*.
- **Atividades-chave**:
  - Investigar o estado de arte e as tecnologias necessárias;
  - Desenhar a nova versão da *API* de forma iterativa;
  - Implementar a nova *API;*
  - Testar o *software* implementado;
  - Avaliar a solução desenvolvida para ver se cumpre os requisitos propostos.
- **Parcerias principais**:
  - *Zend Technologies* como patrocinadora da *Zend Framework 2* (a *framework* utilizada para desenvolver a *API*);
  - Apache Software Foundation como criadora do Apache *ActiveMQ* (o sistema de filas usado neste projeto);
- **Estrutura de custo**: Custos relacionados com o desenvolvimento, manutenção e hardware necessário.

A Figura 15 ilustra o modelo de negócio Canvas elaborado para o projeto desta tese.

Figura 15 - Modelo de negócio Canvas deste projeto

3.5 *Cadeia de valor de Porter*

As atividades realizadas em empresas que são necessárias nas fases de design, produção, marketing e entrega e suporte de novos produtos, podem ser listadas usando a cadeia de valor de Porter [9]. Este diagrama, proposto por Michael Porter, divide estas atividades em atividades de suporte e atividades primárias.

As atividades primárias contêm os seguintes elementos:

- **Logística de entrada**: atividades relacionadas com a entrada dos elementos necessários para desenvolver o produto e o seu armazenamento;
- **Logística de saída**: atividades relacionadas com o armazenamento do produto e a sua distribuição;
- **Operações**: atividades relacionadas com a transformação dos elementos de entrada no produto;
- **Serviço**: atividades relacionadas com providenciar um serviço de forma manter o valor do produto (*e.g. help desk*);
- **Marketing e vendas**: atividades relacionadas com a promoção do produto e como o produto vai chegar ao cliente.

As atividades secundárias contêm os seguintes elementos:

- **Infraestrutura**: atividades que suportam todas as outras atividades da cadeia (*e.g*. contabilidade, planeamento);
- **Gestão de recursos humanos**: "atividades relacionadas com recrutamento, contratação, treino, desenvolvimento e compensação de colaboradores" [21];
- **Desenvolvimento de tecnologia**: lista de atividades realizadas com o objetivo de melhorar o produto e o processo;
- **Obtenção de recursos**: atividades relacionadas com a obtenção dos recursos necessários para criar os produtos.

Na Figura 16 é possível visualizar a cadeia de valor de Porter preenchida relativamente à empresa E-goi.

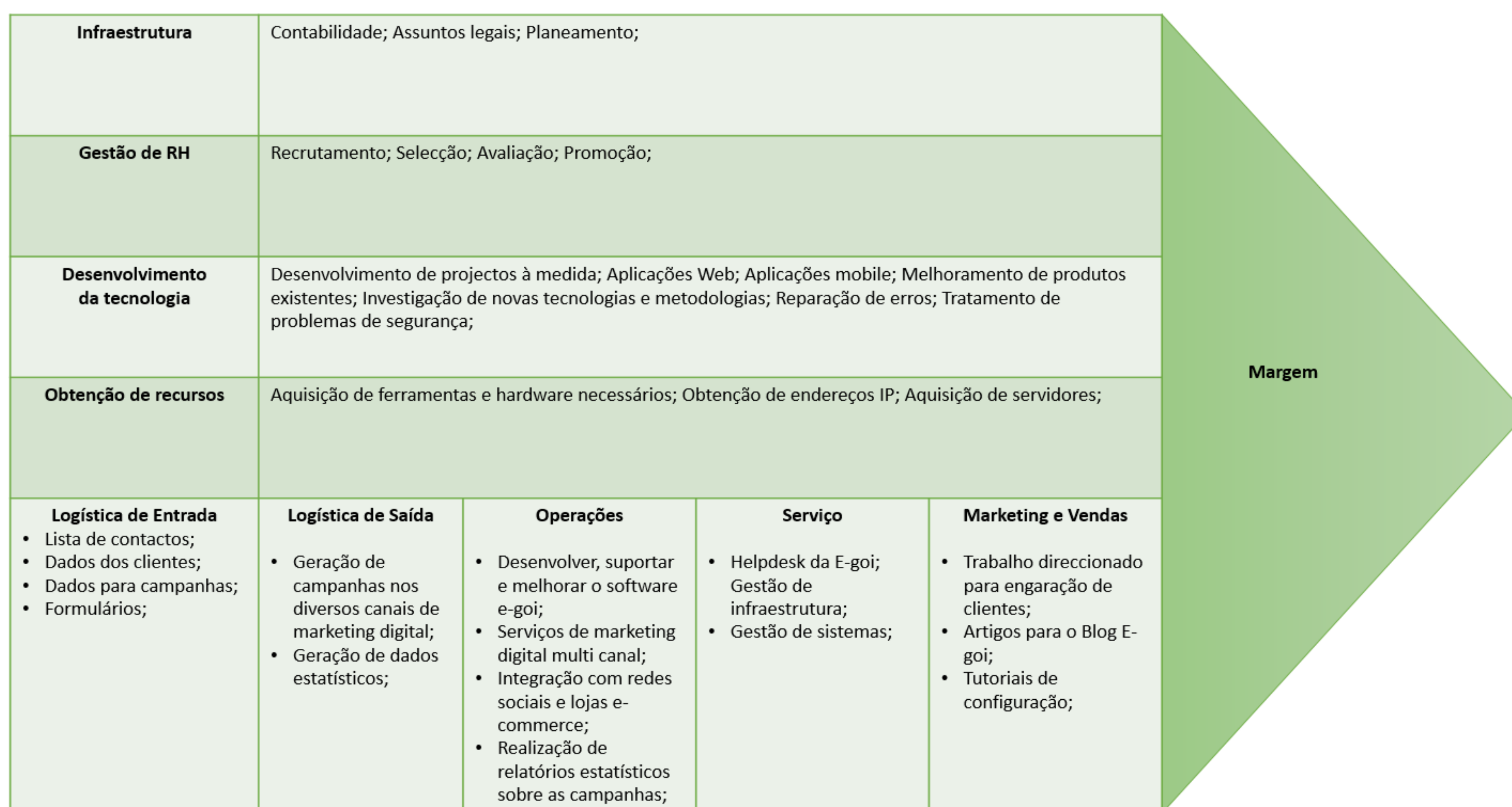


Figura 16 - Cadeia de valor de Porter da E-goi

3.6 Decisão utilizando o modelo AHP

O modelo *AHP* (*Analytic Hierachy Process*) é método que permite a tomar decisões de forma organizada. Para tal é necessário efetuar os seguintes passos [10]:

1. Definir o problema e determinar o conhecimento pretendido;
2. Estruturar a decisão hierarquicamente do topo com o objetivo, seguido dos critérios e as suas cotações e por fim as alternativas;
3. Comparar cada elemento dos critérios com os outros e calcular com base nas matrizes de comparação, o valor de cada critério. Cada elemento dos critérios será usado para comparar os elementos da camada de soluções;
4. Usar as prioridades obtidas nas comparações para pesar as alternativas. Depois para cada solução, comparar entre si relativamente a cada critério de forma a obter o valor final;
5. Fazer uma análise de consistência de opiniões.

A Figura 17 apresenta um esquema do modelo AHP, relativamente a este projeto.

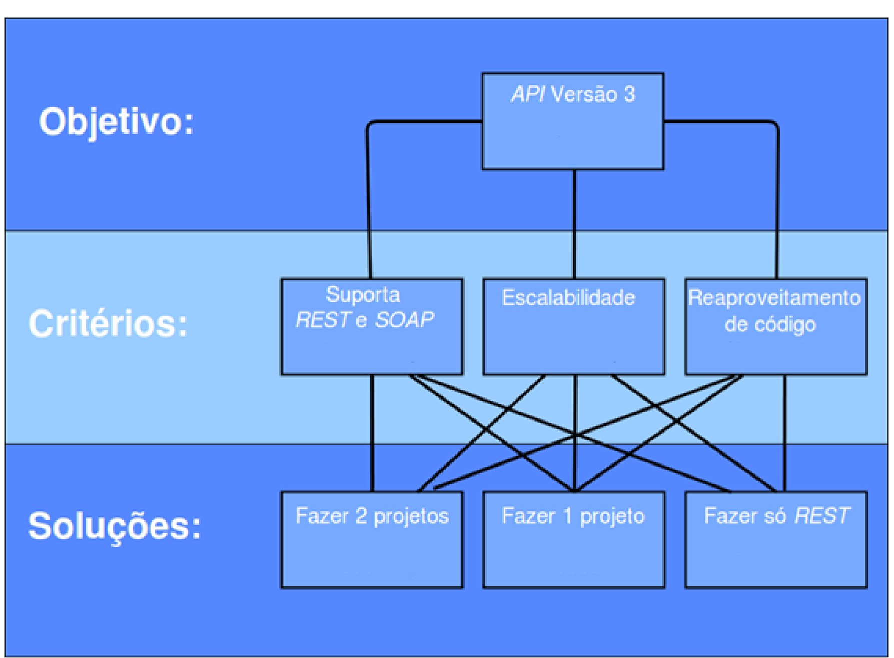


Figura 17 - Esquema do modelo *AHP*

Para comparar os elementos entre si usa-se uma escala de um a nove, em que um significa que os 2 elementos são equivalentes em valor. Nove significa que o elemento que tem o nove é

extremamente mais importante que o outro elemento. Esta comparação também é feita para pesar cada solução entre si relativamente a cada critério.

A decisão que se pretende tomar no caso deste projeto é escolher entre as seguintes soluções:

- **Fazer 2 projetos:** fazer 2 projetos, um para cada *API* de cada estilo arquitetural;
- **Fazer 1 projeto:** fazer um projeto que suporte os 2 estilos arquiteturais;
- **Fazer só a *API REST*:** fazer apenas um projeto que suporte apenas *REST*, uma vez que este estilo arquitetural é o mais popular.

A lista de critério, que têm peso na escolha da melhor solução, são os seguintes:

- **Suporta os 2 estilos arquiteturais:** indica se as soluções respeitam as regras dos estilos arquiteturais *REST* e *SOAP*;
- **Escalabilidade:** indica o quão escalável é a solução;
- **Reaproveitamento de código:** indica quanto código pode ser reaproveitado.

O primeiro passo do modelo é pesar os critérios uns contra os outros. Para tal foi criado o questionário disponível no Anexo A deste documento, que foi respondido pelas pessoas com conhecimento do projeto. Na Figura 18 podemos visualizar a médias desta comparação.

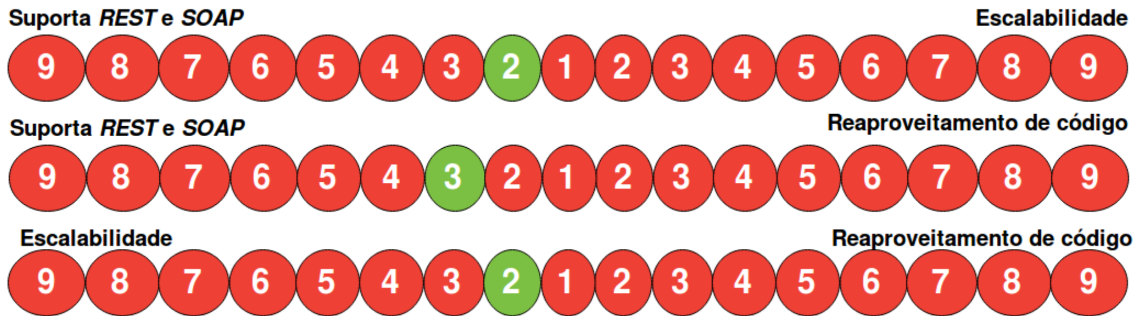


Figura 18 - Comparação de valor entre critérios

Com base na comparação dos critérios realizada na Figura 18, foi criada a matriz de comparação apresentada na Tabela 5. O critério "Suportar *REST* e *SOAP*" vale mais que o critério "escalabilidade" e de "reaproveitamento de código", uma vez que este é um requisito. Entre estes dois últimos, a aplicação ser escalável têm um peso ligeiramente superior.

Tabela 5 - Matriz de comparação de critérios

	Suporta *REST* e *SOAP*	**Escalabilidade**	**Reaproveitamento de código**
Suporta REST e SOAP	1	2	3
Escalabilidade	1/2	1	2
Reaproveitamento de código	1/3	1/2	1

Com base na Tabela 5, foram calculados os pesos de cada critério, calculando o *EigenVector* resultante. Para isso, normalizou-se os resultados coluna a coluna, tal como se pode visualizar na seguinte equação matricial:

$$\begin{bmatrix} 1 & 2 & 3 \\ \frac{1}{2} & 1 & 2 \\ \frac{1}{3} & \frac{1}{2} & 1 \end{bmatrix} \rightarrow \text{Normalizar} \rightarrow \begin{bmatrix} 0.545454 & 0.571429 & 0.500000 \\ 0.272727 & 0.285714 & 0.333333 \\ 0.181818 & 0.142857 & 0.166667 \end{bmatrix}$$

Por fim efetuou-se a média de cada uma das linhas da matriz normalizada, obtendo o *EigenVector,* apresentado na Tabela 6.

Tabela 6 - Tabela de pesos de cada critério

Critério	***Weight (EigenValue)***	**Importância**
Suporta REST e *SOAP*	0.539615	Maior
Escalabilidade	0.296961	Intermédio
Reaproveitamento de código	0.163424	Menor

Depois de estabelecidas as igualdades, os valores de cada critério foram calculados através do processo descrito previamente, após 3 iterações. O critério "Suporta *REST* e *SOAP*" vale 54%, "Escalabilidade" 29.7% e "Reaproveitamento de código" 16,3%.

O próximo passo é realizar uma análise de consistência das matrizes apresentadas na Tabela 6, o que irá permitir verificar se as matrizes são consistentes. Os cálculos foram realizados seguindo os seguintes passos [48]:

1. Multiplicar cada valor de *cada* coluna pelo valor correspondente do E*igenVector.* De seguida somar as linhas da matriz resultado e dividir pelo *EigenVector* e calcular a média do soma do vetor resultado. para obter o λmax;
2. Calcular um índice de consistência (IC) com a seguinte fórmula: $IC = \frac{\lambda max - n}{n-1}$, onde o n é o número de linhas;
3. Calcular a relação de consistência (RC) com a seguinte fórmula: $RC = \frac{IC}{IAM}$.

A Inconsistência Aleatória Média (IAM) é uma constante dependente da dimensão da matriz em análise e pode ser visto na Tabela 7. É desejável que a relação de consistência seja inferior ou igual a 0,1 [48].

Tabela 7 - Índice de Consistência Aleatória [48]

Dimensão da matriz	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
IAM	0.00	0.00	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

Assim sendo, foram efetuados os seguintes cálculos:

$$\begin{bmatrix} 1 & 2 & 3 \\ \frac{1}{2} & 1 & 2 \\ \frac{1}{3} & \frac{1}{2} & 1 \\ SC1 & SC2 & SC3 \end{bmatrix}$$, onde o SCn é a soma da coluna n.

Obtendo o seguinte resultado para a soma de cada coluna:

$$SC1 \approx 1.833333; SC2 = 3.5; \mathrm{SC3} = 6;$$

De seguida o cálculo do λ foi realizado:

$$\lambda\mathrm{max} \approx \mathrm{SC1} * 0.539615 + \mathrm{SC2} * 0.296961 + \mathrm{SC3} * 0.163424 \Leftrightarrow \lambda\mathrm{max} \approx 3.009201$$

Utilizando esse resultado, calculou-se um IC:

$$IC \approx \frac{3.009201-3}{3-1} \Leftrightarrow IC \approx 0.004601$$

Por fim calculou-se a RC:

$$RC \approx \frac{0.004601}{0.58} \Leftrightarrow RC \approx 0.007931$$

Como podemos verificar, a relação de consistência da matriz de comparação de critérios é inferior a 0.1, pelo que é possível avançar.

De seguida realizou-se uma comparação entre as soluções para cada um dos critérios. O resultado das médias do questionário do Anexo A, foi compilado na Figura 19.

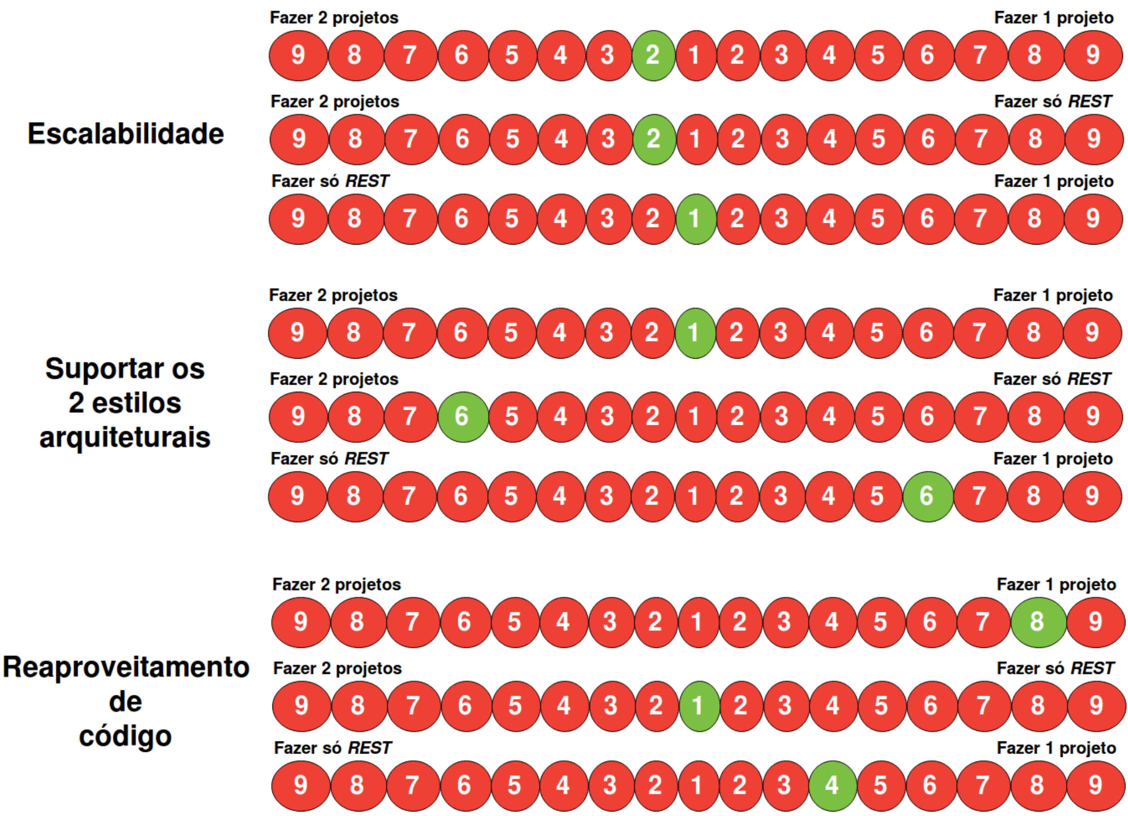


Figura 19 - Comparação entre as soluções para cada critério

Os resultados apresentados na Figura 19 foram compilados nas seguintes tabelas:

Tabela 8, Tabela 9; Tabela 10.

Tabela 8 - Comparação das soluções com base no critério "Suporta *REST* e *SOAP*"

	Fazer 2 projetos	**Fazer 1 projeto**	**Fazer só *REST***	***EigenValue***
Fazer 2 projetos	1	1	6	0.461538
Fazer 1 projeto	1	1	6	0.461538
Fazer só REST	1/6	1/6	1	0.0769231

Tabela 9 - Comparação das soluções com base no critério "Escalabilidade"

	Fazer 2 projetos	**Fazer 1 projeto**	**Fazer só *REST***	***EigenValue***
Fazer 2 projetos	1	2	2	0.5
Fazer 1 projeto	1/2	1	1	0.25

	Fazer 2 projetos	**Fazer 1 projeto**	**Fazer só *REST***	***EigenValue***
Fazer só REST	1/2	1	1	0.25

Tabela 10 - Comparação das soluções com base no critério “Reaproveitamento de código”

	Fazer 2 projetos	**Fazer 1 projeto**	**Fazer só *REST***	***EigenValue***
Fazer 2 projetos	1	1/8	1	0.116150
Fazer 1 projeto	8	1	4	0.737509
Fazer só REST	1	1/4	1	0.146340

De seguida, para cada uma das matrizes apresentadas, foram calculados os IC e as RC, da mesma forma que anteriormente. Estes resultados são apresentados na Tabela 11.

Tabela 11 - Índices de consistência por critério

Matriz	IC	RC
Suporta os 2 estilos arquiteturais	0	0
Escalabilidade	0	0
Reaproveitamento de código	0.0268108	0.046226

Como se pode verificar, todas as matrizes possuem uma relação de consistência inferior a 0.1, pelo que é possível continuar.

Por fim, multiplicando a matriz, resultante da agregação dos *EigenVector* de cada uma das matrizes que comparam as soluções por critério (Tabela 8, Tabela 9 e Tabela 10), pelo *EigenVector* da matriz de comparação de critérios (Tabela 6) obtemos o peso final de cada solução:

$$\begin{bmatrix} 0.461538 & 0.50000 & 0.116150 \\ 0.461538 & 0.250000 & 0.737509 \\ 0.076923 & 0.250000 & 0.146340 \end{bmatrix} * \begin{bmatrix} 0.539615 \\ 0.296961 \\ 0.163424 \end{bmatrix} \approx \begin{bmatrix} 0.37981 \\ 0.413937 \\ 0.206252 \end{bmatrix}$$

Os resultados obtidos deram origem ao resultado apresentado na Figura 20, onde os pesos de cada critério e o peso final de cada solução foram preenchidos.

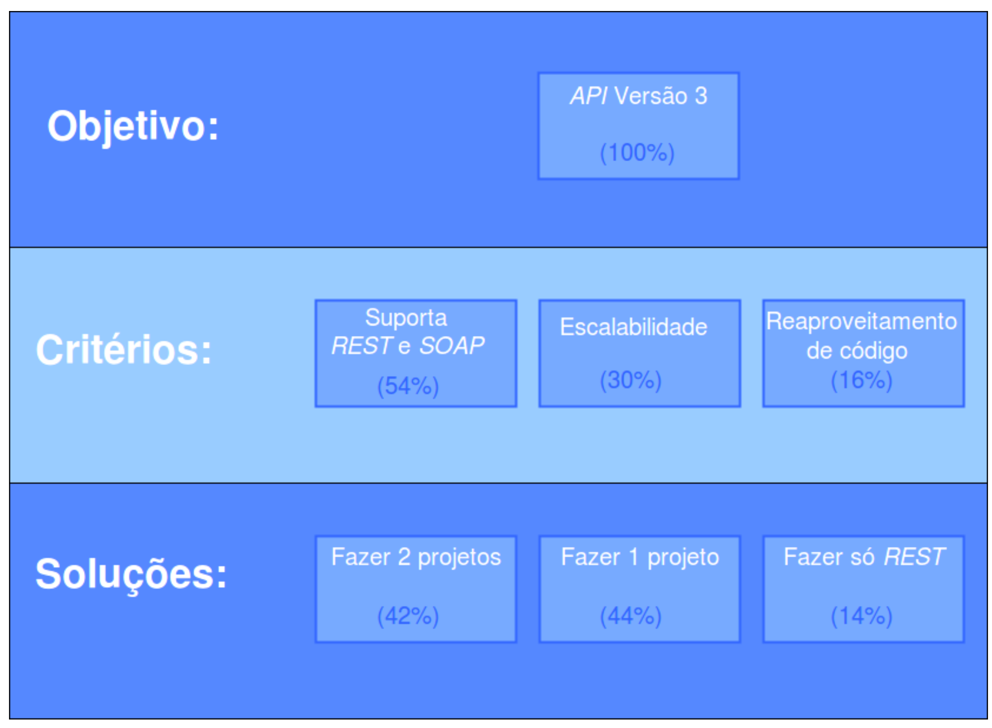


Figura 20 - Modelo *AHP*

Os resultados apresentados mostram que a melhor solução é fazer 1 projeto que suporte os estilos arquiteturais *REST* e *SOAP* com um resultado final de 44% em comparação com 42% para fazer 2 projetos separados e 14% para suportar só *REST*.

4 Análise e Design arquitetural

Neste capítulo são apresentadas a análise de negócio e análise de requisitos. A análise de negócio contem o modelo de domínio e o processo de negócio. A análise de requisitos apresenta os atores do sistema, os requisitos funcionais e outros requisito e é descrita a arquitetura da solução desenvolvida, considerados os requisitos anteriormente descritos.

4.1 Análise de negócio

Nesta secção são apresentados o modelo de domínio da plataforma E-goi e que será maioritariamente espelhado pela nova *API*, e o modelo de processos exemplificativo do fluxo ocorrido quando um utilizador realiza um pedido à *API*.

4.1.1 Modelação do Domínio

O modelo de domínio ou modelo de classes conceptual é um artefacto que representa as entidades conceptuais do domínio, os seus atributos e as suas relações [49]. Na Figura 21 estão representados os principais conceitos do domínio do sistema E-goi.

A nova *API* pública da E-goi vai disponibilizar parte das funcionalidades atualmente existentes no sistema E-goi, através de serviços *REST* e *SOAP.* Por este motivo o domínio desta *API* termina por ser uma versão resumida do refente ao sistema E-goi.

Existem várias entidades do sistema E-goi que não foram representadas na Figura 21 para simplificar a leitura, uma vez não serão disponibilizadas pela *API* no contexto deste projeto e não são necessárias para o seu entendimento.

O sistema permite aos clientes a criação de listas de contactos e o envio de campanhas de *fax*, *email*, voz e *SMS*. Os utilizadores podem posteriormente consultar relatórios com dados referentes ao envio das campanhas.

As listas de contactos podem possuir segmentos que representam parte da lista (*e.g.* todos os contactos Portugueses, todos os contactos com a etiqueta "Importante"). Cada contacto pode ainda estar associado uma ou mais etiquetas para que possa ser facilmente agrupado.

Cada utilizador possui uma conta que está associada a um ou mais remetentes com os dados necessários para identificar o remetente das campanhas. As contas E-goi podem ter associados múltiplos utilizadores sendo que só um deles é o administrador. Esse sistema possibilita que vários clientes partilhem uma só conta.

Outra funcionalidade da E-goi que será disponibilizada na *API* é a criação de automações. Estas descrevem *workflows* para automatizar o envio de campanhas. É ainda disponibilizada a funcionalidade de criação e envio de formulários, que permitem que novos utilizadores se inscrevem em listas de contactos.

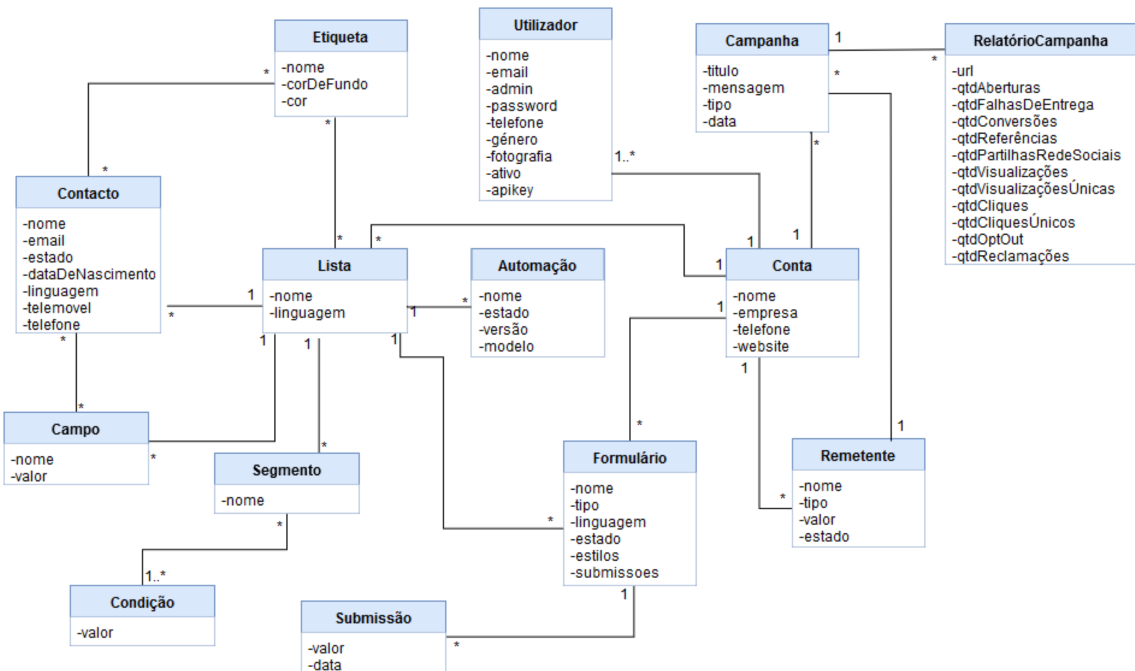


Figura 21 - Modelo de domínio do sistema E-goi

A elaboração do glossário do domínio tem a finalidade de permitir um melhor entendimento das entidades de domínio. Na Tabela 12 são descritas as entidades de negócio mais relevantes das mesmas para o domínio do problema.

Tabela 12 - Glossário

Entidade	**Descrição**
Automação	Sistema que pode ser criado no e-goi que descreve um *workflow* para o envio automático de campanhas
Campanha	Uma campanha pode ser de *email*, *fax*, *SMS* ou voz e é enviada para contactos de uma lista
Campo	Campos extra dos contactos para os contactos de uma lista, para além de nome, email e os restantes já disponibilizados
Conta	Representa uma conta E-goi
Contacto	Um elemento de uma lista de contactos
Etiqueta	Etiqueta associado a um contacto que permite que estes sejam agrupados
Formulário	Formulário que pode ser criado e enviado através do E-goi e que permite que utilizadores se inscrevam em listas de contactos
Lista	Uma lista de contactos e as suas informações para onde serão enviadas as campanhas
RelatórioCampanha	Representa um relatório de uma campanha e contém informação e estatísticas que permite determinar o grau de sucesso de uma campanha
Remetente	Representa os dados necessários para identificar quem lançou a campanha
Segmento	Um segmento de uma lista de contactos é uma parte de uma lista de contactos (*e.g.* Todos os contactos Portugueses da lista)
Utilizador	Representa um utilizador associado a uma conta E-goi. Cada conta pode ter múltiplos utilizadores, mas só um deles pode ser *admin*

4.1.2 Modelação do Processo de Negócio da Solução

Um processo de negócio é o trabalho realizado por uma organização de forma a atingir um propósito ou objetivo específico [50].

A Figura 22 mostra o diagrama de processos segundo a notação *Business Process Modelling Notation (BPMN)*, onde está representado o fluxo do processo executado quando um utilizador realiza um pedido à *API*.

Dependendo se o pedido realizado foi *REST* ou *SOAP*, o fluxo é direcionado de acordo. Seguidamente a *apikey* é analisada e, caso seja inválida, é retornada uma mensagem de resposta ao utilizador com essa informação e o processo é terminado. Se for válida, os argumentos enviados são analisados e caso o seu formato seja incorreto, ou sejam inválidos, é retornada uma mensagem a avisar do problema e o processo termina.

Em caso de sucesso nas validações os dados são transmitidos para o sistema de versionamento que os envia posteriormente para o serviço e versão correspondentes. De seguida a *API* pública realiza os pedidos necessários à *API* interna da E-goi e transforma os dados das respostas para o formato pretendido. A informação relevante é transmitida para o sistema de filas para ser posteriormente processada e arquivada. Por fim a mensagem de resposta é retornada ao utilizador com a informação requisitada.

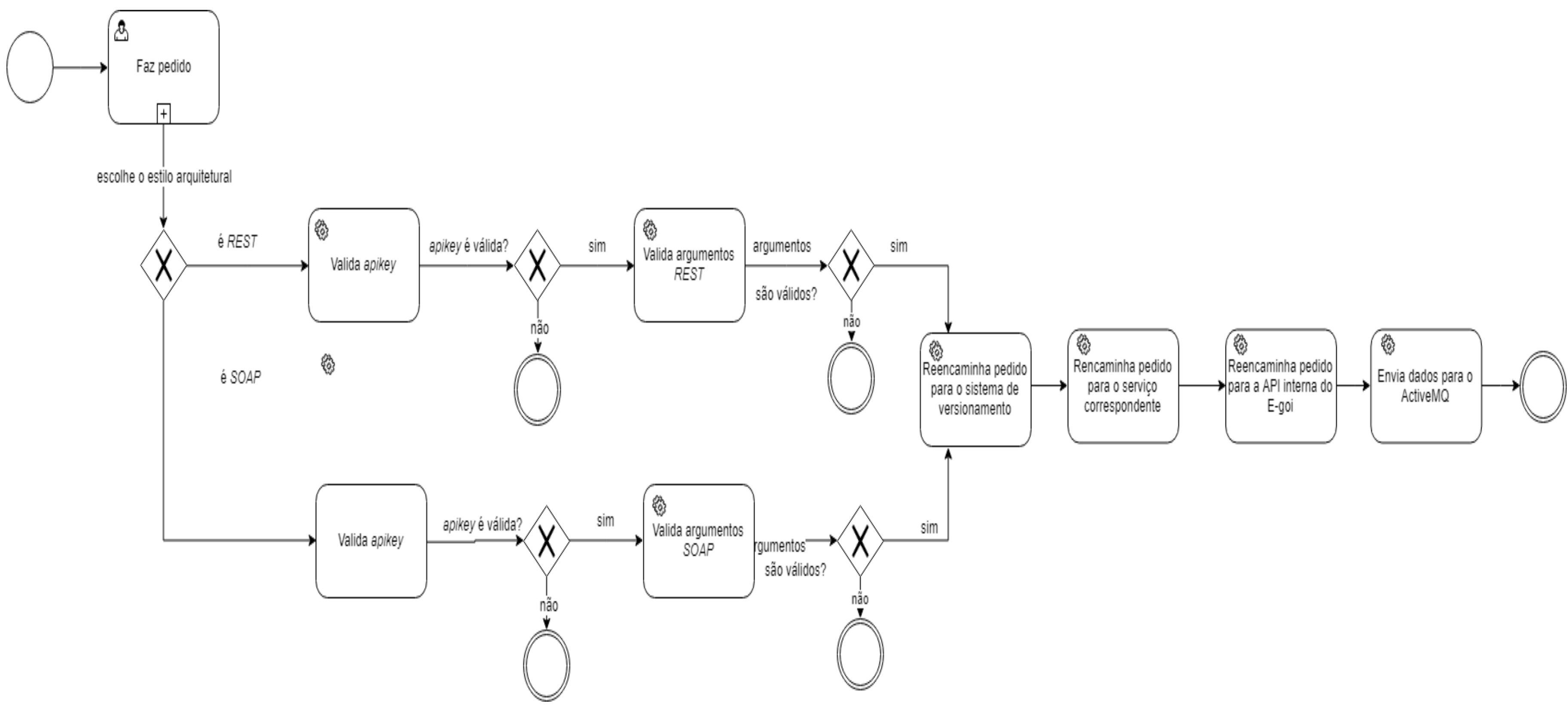


Figura 22 - Diagrama de processo

4.2 Análise de Requisitos

A Engenharia de Requisitos é um processo de Engenharia de *Software* que estuda os requisitos que devem ser satisfeitos por um sistema, de forma a resolver um problema. Este processo envolve o estudo da criação, análise, desenvolvimento e manutenção dos requisitos. Envolve ainda o estudo das necessidades do utilizador, de modo a encontrar uma definição correta e completa de um produto de *software*. O principal objetivo deste processo é compreender e documentar as necessidades das partes interessadas. Uma vez que todas as fases posteriores de desenvolvimento de uma solução dependem da análise de requisitos, esta é essencial para o sucesso das restantes fases.

4.2.1 Requisitos Funcionais

O único ator de sistema da *API* é o "cliente E-goi" e esta deve disponibilizar parte das funcionalidades do *back-end* do sistema E-goi. Após diversas reuniões com as partes interessadas, identificou-se os seguintes requisitos funcionais representados na Tabela 13, referentes às apresentadas na secção 4.1.1.

Tabela 13 - Lista de requisitos funcionais

Identificação	**Entidade**	**Requisito**
RF001	Automação	Criar e retornar uma lista de automações
RF001.1		Criar uma automação a partir de um *template*
RF001.2		Editar as opções de uma automação
RF001.3		Iniciar uma automação
RF002	Contacto	Criar, editar, remover, retornar contactos ou retornar um contacto
RF002.1		Importar um conjunto de contactos para uma lista
RF002.2		Retornar uma lista de atividades de um contacto
RF003	Lista	Criar, editar ou remover, retornar uma lista, retornar todas as listas
RF004	Segmento	Criar, remover e retornar todos os segmentos
RF005	Campo	Criar, editar, remover retornar todos os campos
RF006	Remetente	Criar, editar, remover e retornar todos os remetentes.
RF007	Etiqueta	Criar, editar, remover e retornar uma lista de etiquetas.
RF008	Utilizador	Criar, remover, retornar um utilizador por identificador e retornar os utilizadores
RF008.1		Promover um utilizador a administrador da conta
RF009	Campanha	Criar, editar, remover, retornar por identificador e retornar as campanhas
RF010	RelatórioCampanha	Retornar os relatórios das campanhas de *email, SMS*, *voz*, *push* e *web-push*

Nesta fase não é necessário disponibilizar na *API* públicas todas as funcionalidades existentes na plataforma E-goi, pelo que nem todas as entidades estão presentes. A principal responsabilidade desta *API* é reencaminhar o pedido para a *API* interna, executando os pedidos necessários, e transformar os dados da resposta para um novo formato. Todos os serviços devem ser criados com base na especificação *Open API* desenvolvida por outros elementos, pelo que as suas rotas e formato de resposta devem ser os especificados.

4.2.2 Outros requisitos

Nesta secção são capturados os requisitos não funcionais não incluídos nas descrições dos casos de uso e requisitos não funcionais, utilizando o modelo *Functionality, Usability, Reliability, Performance and Supportability Plus (FURPS+*). Tal como será explicado na secção 5.1.2, ainda não existe decisão se será disponibilizado um sistema de versionamento pelo que este não é um requisito de todo o projeto e apenas foi aplicado n recurso *Lists*.

- **Funcionalidade:**
  - Disponibilizar funcionalidades do sistema E-goi no estilo arquitetural *REST* e no protocolo *SOAP*;
  - Disponibilizar um sistema de versionamento por serviço que funcione em *REST* e em *SOAP* e que possibilite a utilização de *wild-cards*.
- **Usabilidade:**
  - As mensagens de erro devem ser claras o suficiente para o utilizador entender o problema e como resolver e fazer uso dos códigos *HTTP* apropriados (em *REST*).
- **Desempenho:**
  - A *API* deve ser eficiente e a sua utilização não deve representar um aumento médio superior a 20% em relação aos pedidos diretos à *API* privada.
- **Suportabilidade:**
  - O sistema deve ser escalável e flexível.
- **Outros+:**
  - **Limitações de design:**

        - O desenho da aplicação deve seguir os padrões/princípios *General Responsibility Assignment Software Patterns* (*GRASP*) e *SOLID*;
        - Seguir as convenções e boas práticas de *REST e SOAP.*
    - **Limitações de implementação:**
        - A *API* deve ser desenvolvida utilizando a *Zend Framework 2*.

4.3 Design arquitetural

Nesta secção são apresentados alguns artefactos do design arquitetural da solução. Estes são o diagrama de componentes da solução e alternativas arquiteturais, o diagrama de pacotes e o diagrama de implantação.

4.3.1 Componentes

A Figura 23 apresenta o diagrama de componentes da solução pretendida neste projeto. A *API* é dividida em módulos, referentes aos recursos a disponibilizar, como é norma na *Zend Framework 2*.

Cada módulo referente a um recurso (*e.g.* listas, contactos) possui as seguintes componentes:

- ***RESTController***: representa o controlador *REST* de um recurso. É esta componente que recebe os pedidos dos utilizadores e chama o serviço pretendido. Uma vez que os serviços estão associados a um sistema de versionamento, a responsabilidade de escolher o serviço é reencaminhada para a componente *VersionManager*;
- ***SOAPController:*** representa o controlador *SOAP*. Não está representado dentro da componente module uma vez que todos os serviços *SOAP* vão poder ser acedidos através da mesma rota, sendo que a escolha do serviço a chamar é realizada através do *SOAP Envelope.* Esta componente também disponibiliza o *WSDL* com a lista e descrição dos serviços disponíveis;
- ***VersionManager***: possui a responsabilidade de, com base num nome de um serviço e a versão pretendida, disponibilizar esse serviço para o controlador. Se o utilizador não especificar que versão pretende, a última versão do serviço é disponibilizada;
- ***ServiceVersion***: representa uma versão de um serviço e todas as componentes necessárias para o mesmo funcionar corretamente. Divide-se nas seguintes subcomponentes:

- *InternalAPIModel*: é o modelo de dados que será transmitido para a *API* interna. Possui a responsabilidade de transformar os argumentos de entrada de um pedido num formato válido para ser transmitido para os serviços necessários da *API* interna;
- ***InputValidator***: disponibiliza para a componente *InternalAPIModel* a funcionalidade de validar os dados de entrada do pedido realizado à *API* pública. Verifica se são sintaticamente e semanticamente corretos para que no caso de serem inválidos, as mensagens de erro apropriadas possam ser retornadas;
- ***Model***: representa o modelo de dados da resposta que será retornada ao utilizador, com base nas respostas da *API* privada;
- ***OutputValidator***: valida a resposta da *API* privada para verificar se ocorreu algum erro ocasional que só possa ser detetado após realizar o pedido (*e.g. Conflict* ou *Internal Server Error*);
- ***Service***: esta componente representa o serviço e a lógica necessária para o mesmo funcionar. Primeiramente utiliza a *InternalAPIModel* para transformar e validar os dados de entrada no formato necessário para realizar os pedidos necessários à *API* interna. Posteriormente utiliza a componente *Model* para transformar e validar os dados recebidos na resposta final que será retornada para o utilizador.

- ***API Errors***: representa a componente responsável por garantir que os erros retornados ao utilizador indicam o problema que ocorreu e como o utilizador deve proceder para o resolver. Divide-se nas seguintes subcomponentes:
    - ***HTTPErrors***: responsável por disponibilizar retornos em caso de erros ocorridos na *API REST*. Esta componente utiliza os códigos *HTTP* apropriados e retorna toda a informação necessária;
    - ***SOAPErrors***: esta componente é semelhante à *HTTPErrors*, mas disponibiliza a resposta através de *SOAP Faults* com a informação necessária.

As seguintes componentes não são desenvolvidas neste projeto, mas são utilizadas por esta *API*. Por este motivo a sua representação é necessária para o permitir o correto entendimento do projeto:

- ***InternalAPI:*** representa a *API* interna que é utilizada pelo *web site* da e-goi e disponibilizará os serviços para a *API* pública. Os serviços disponibilizados por esta componente não têm uma estrutura suficientemente adequada para serem disponibilizados de forma pública. Também

por uma questão de segurança, uma vez que existirá uma dupla verificação dos argumentos enviados nos pedidos, faz sentido disponibilizar os serviços através da nova *API* pública;

- ***DB:*** representa a base de dados onde estão guardados os dados dos clientes. Estes dados são disponibilizados para a *API* interna da E-goi e posteriormente para a nova *API* pública;

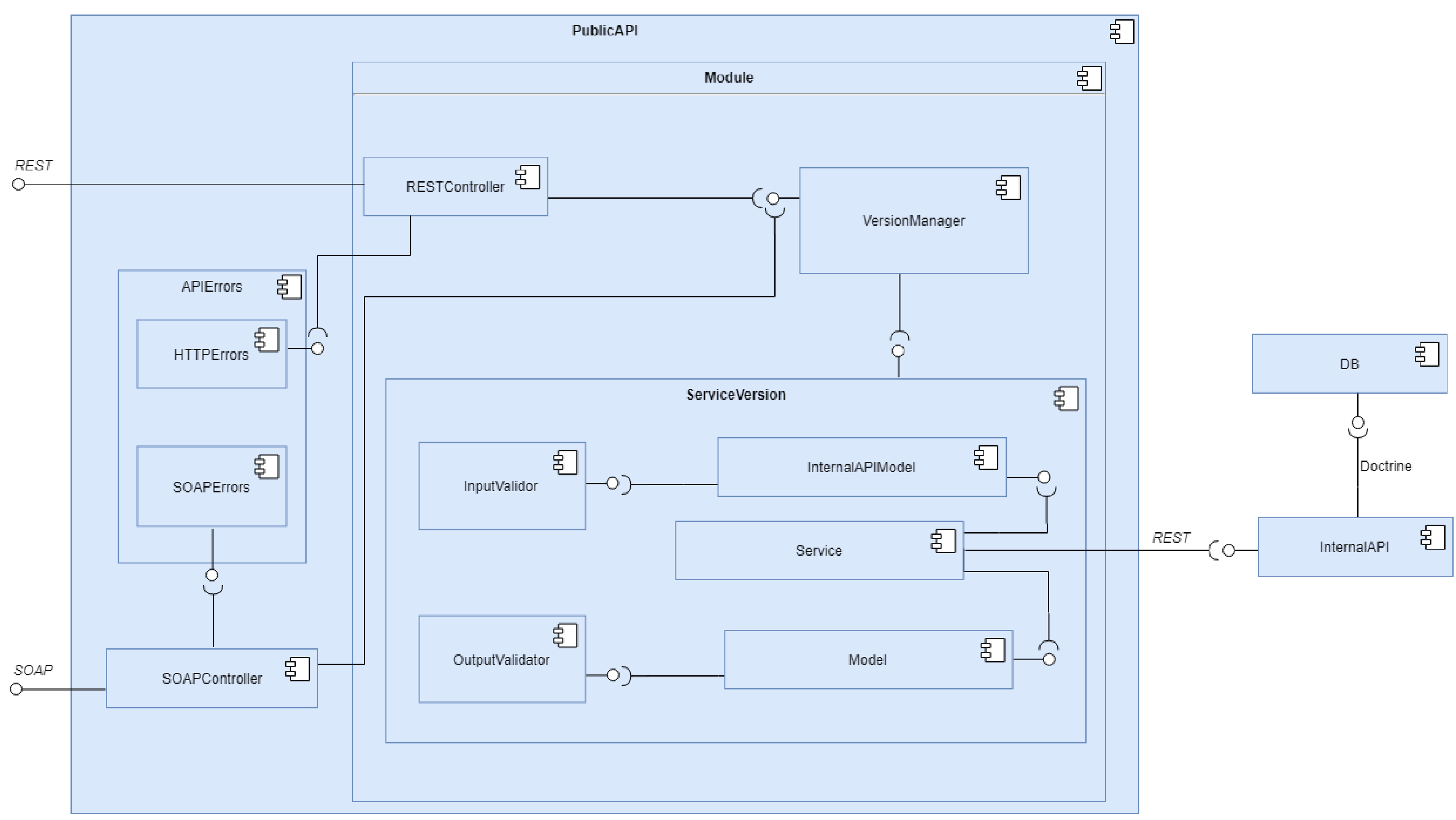


Figura 23 - Diagrama de componentes da solução

4.3.1.1 Alternativas Arquiteturais

As alternativas arquiteturais refletem as alternativas discutidas na secção 3.6, e no Anexo C duas alternativas arquiteturais sobre como poderia ser suportada a tecnologia *GraphQL* no futuro.

A Figura 24 representa a alternativa com 2 projetos separados, um para a *API REST* e outro para a *SOAP*. Neste caso, ambas as *API* comunicavam separadamente com a *API* interna. Enquanto esta alternativa é mais escalável do que a opção escolhida, não permite reaproveitamento do código comum entre *REST* e *SOAP.*

Dado que as rotas são comuns em ambos os casos, e a funcionalidade é a de reencaminhar o pedido da mesma forma para a API privada, esta não é a melhor alternativa para a empresa porque requer replicação dessa lógica.

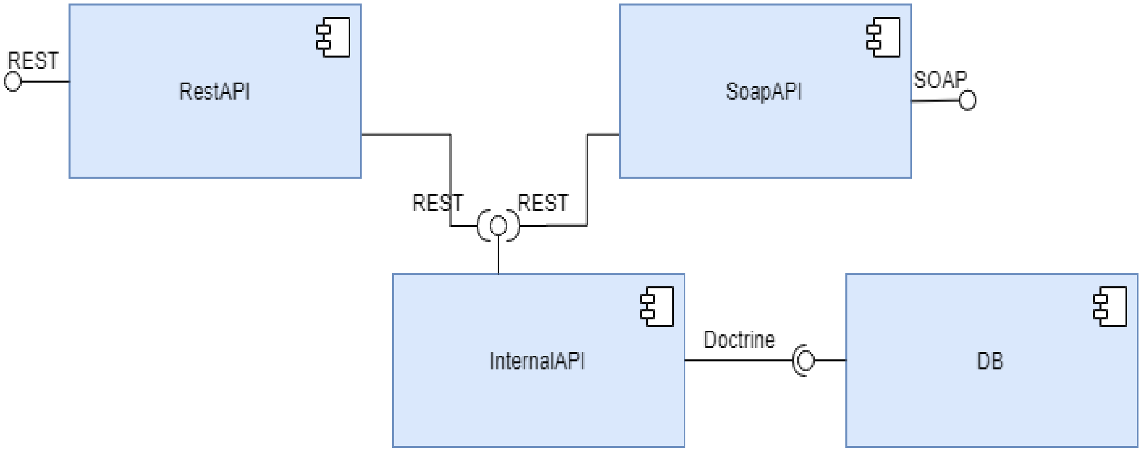


Figura 24 - Diagrama de componentes alternativo 1

A Figura 25 representa outra possibilidade coma *API SOAP* a utilizar a *API REST.* Esta solução é muito semelhante à anterior, mas é mais escalável. Em vez de ambas as *API* utilizarem a *API* interna, apenas a de *REST* o faz*.* Esta solução possibilita a criação da *API REST* em primeiro lugar e posteriormente a de *SOAP*.

Dado que a maioria dos clientes E-goi que utilizam a sua *API* pública usam *REST* e que a *REST* é atualmente o estilo mais utilizado nas *API* [2], esta opção foi considerada. Outra vantagem desta da solução é que a *API SOAP* está duplamente validada dado os seus *inputs* serão também validados na *API REST*.

O principal problema com esta solução é a quantidade de pedidos realizados por pedido *SOAP*. Cada pedido *SOAP* implica dois pedidos *REST* para funcionar. Dada a alta taxa de utilização da *API* pública da E-goi, esta não é do ponto de vista da empresa a melhor opção.

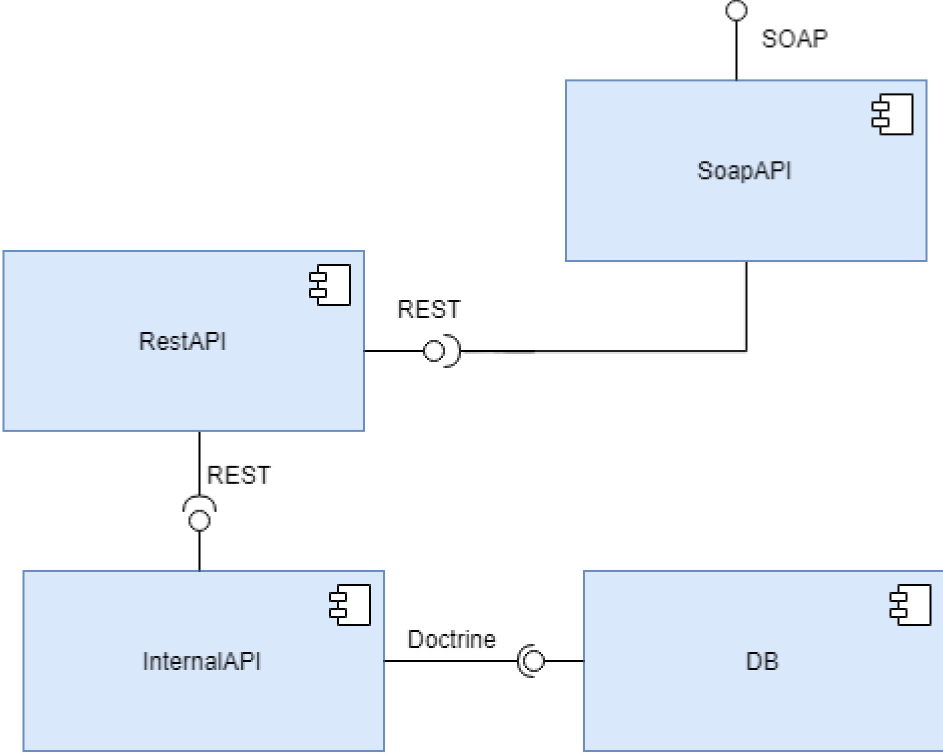


Figura 25 - Diagrama de componentes alternativo 2

4.3.2 Estrutura das pastas

A Figura 26 apresenta o diagrama de pacotes da solução onde se pode visualizar a estrutura de pastas da solução. O projeto está organizado segundo a estrutura da *Zend Framework* 2. A pasta *contacts* é representativa da organização do modulo C*ontacts*, sendo que os outros módulos possuem estruturas semelhantes. Esta pasta contém as seguintes subpastas:

- ***config***: os ficheiros de configuração do módulo;
- ***tests***: os ficheiros referentes a testes do *PHPUnit*;
- ***src***: esta pasta contém outra pasta com o mesmo nome da pasta pai e essa subpasta contém as seguintes pastas:
    - ***Factory***: as classes *factory* necessários para criar outras classes, como por exemplo as operações ou os modelos;

- ***Controller*:** contém o controlador *REST* do módulo;
- **Versions:** possui as componentes do sistema de versionamento por serviço. Possui as seguintes subpastas:
    - ***Operations*:** as classes cuja responsabilidade é realizar os pedidos às *API* interna da E-goi;
    - ***Models*:** a classe com os modelos de resposta dos serviços;
- ***InternalAPIModels*:** as classes com o modelo de dados necessário para realizar pedidos à *API* interna.

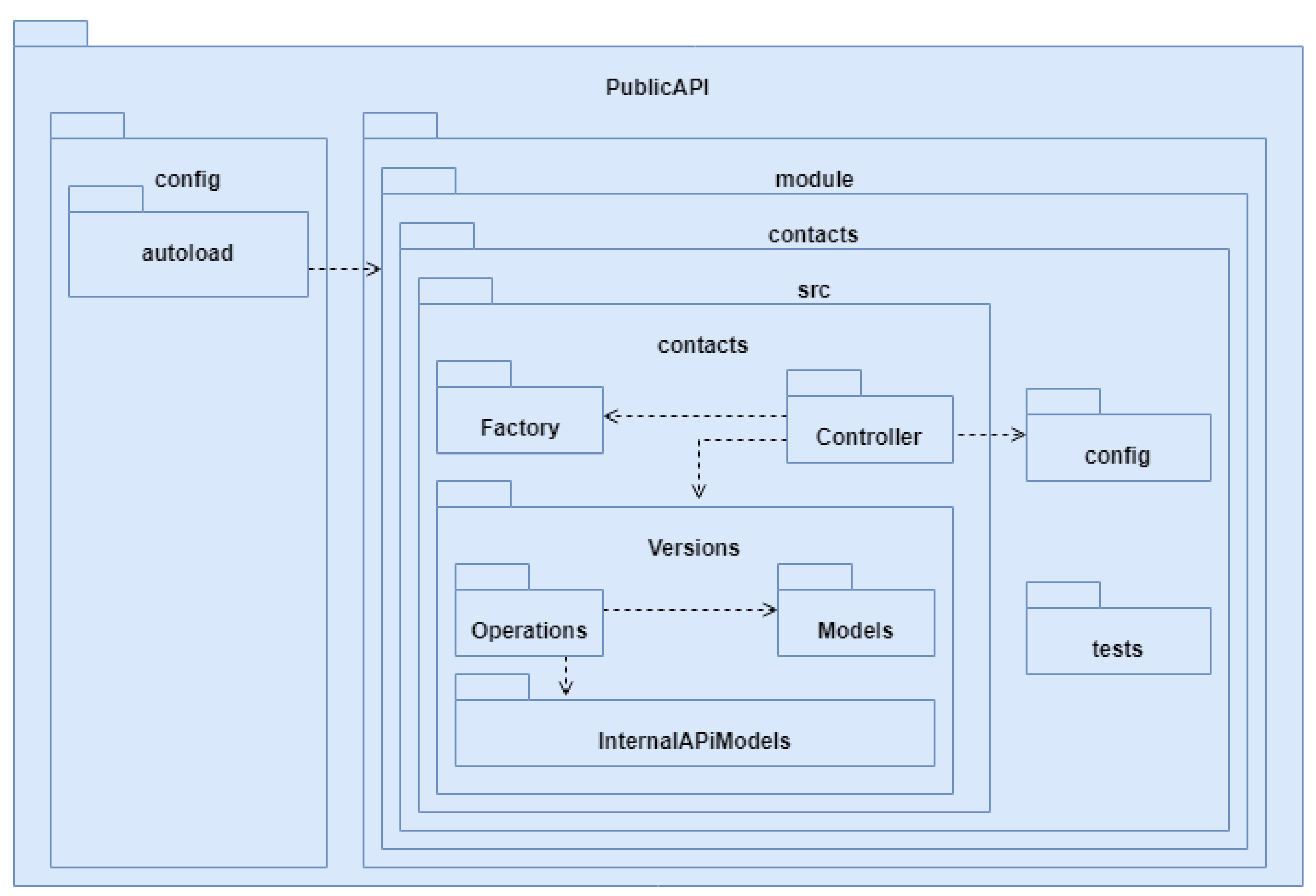


Figura 26 - Diagrama de pacotes

4.3.3 Implantação

A Figura 27 apresenta o diagrama de implantação da solução. O cliente pode utilizar a *API* a partir da sua própria máquina, fazendo um pedido *REST* ou *SOAP*. A *API* desenvolvida corre num servidor privado da E-goi que utilizado apenas para este efeito, e configurado pelos administradores de sistema utilizando Nginx. Dependendo da quantidade de pedidos, poderá ser necessário colocar num *cluster*.

A *API* privada encontra-se num cluster devido à carga causada pelos serviços disponibilizados pelo E-goi (*e.g.* envios de campanhas de *email*, voz, *SMS*, *push* e *web-push*) e à quantidade de utilizadores diários.

As bases de dados onde se encontram os dados dos clientes estão divididas num *cluster*, sendo que existe uma base de dados global onde estão guardados dados resumidos de estatísticas, e uma base de dados por cada cliente com os seus próprios dados das contas.

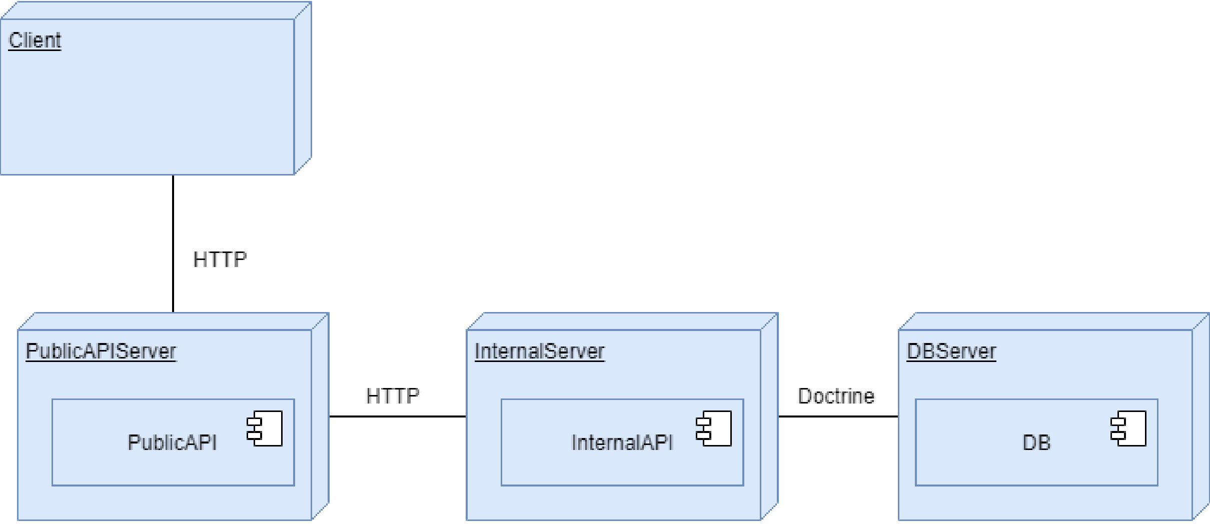


Figura 27 - Diagrama de implantação

5 Design e implementação

Neste capítulo apresenta-se o design da solução e a implementação. O Design da solução apresenta diagramas de classes representativos da solução de forma a explicar alguns conceitos e decisões, e tenta ser escalável e flexível. A implementação disponibiliza diagramas de sequência de forma a explicar detalhes de implementação e documenta decisões que foram tomadas.

5.1 Design da solução

Nesta secção é exibido o design das componentes mais importantes do trabalho. Estas são o sistema de versionamento, a disponibilização de *REST* e *SOAP*, o sistema de erros e a forma como a *API REST* suporta vários formatos de resposta (*e.g. XML e JSON*). Este design foi pensado de forma a ser escalável e de fácil manutenção, para que os problemas correspondente da *API* anterior, indicados na secção 2.4 não existam na nova *API*.

5.1.1 Sistema de versionamento

A Figura 28 representa o diagrama de classes do sistema de versionamento implementado na *API* criada neste projeto. Neste diagrama exemplificativo são apresentadas algumas classes do módulo *Lists*, o único módulo que foi integrado com o sistema de versionamento, como será explicado na secção 5.1.2.

A classe *ListsRestController* apresentada representa o controlador *REST* do módulo *Lists*. O sistema de versionamento utiliza as regras do versionamento semântico descritas na secção 2.1.3.1 que estrutura a versão no formato *Major.Minor.Patch* [22].

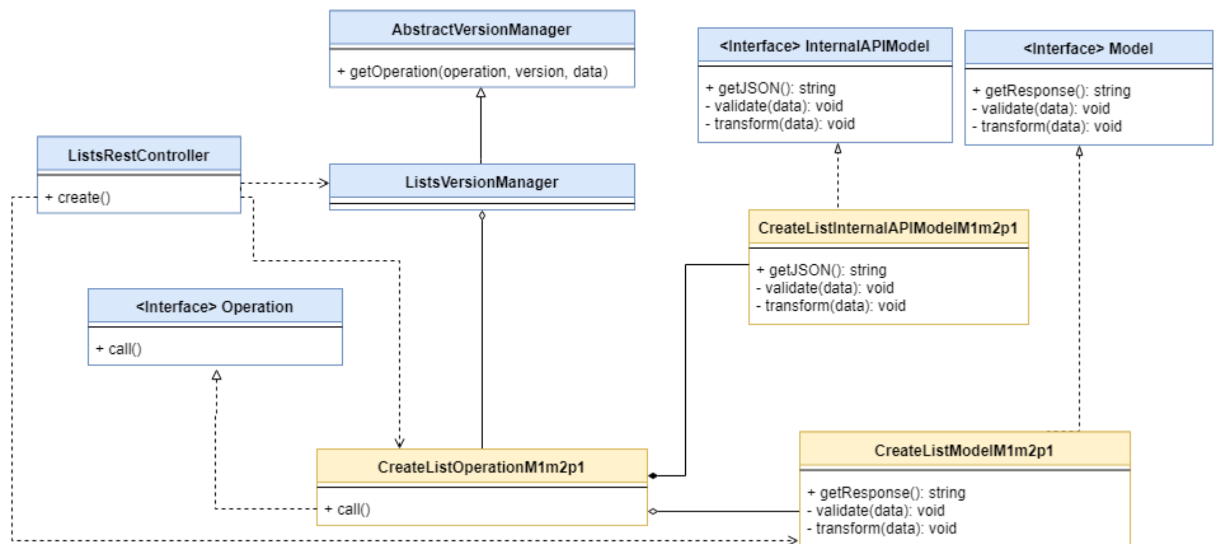


Figura 28 - Diagrama de classes do sistema de versionamento

No sistema de versionamento proposto neste documento, as classes representadas a amarelo terão de ser criadas sempre que ocorra uma atualização de um serviço. As classes necessárias, por cada versão de um serviço, estão representadas com o sufixo *M1m2p1* que significa m*ajor* 1, *minor* 2 e *patch* 1.

No exemplo, o serviço apresentado chama-se *CreateList* e a versão do serviço apresentada é a 1.2.1. As três classes que são criadas para o lançamento de uma versão, e que seguem esta nomenclatura são:

- ***Operation:*** possui a responsabilidade de instanciar e utilizar a classe *InternalAPIModel*, de forma a obter os dados no formato necessário para efetuar pedidos à *API* interna. Contém a função *call* por intermédio da interface *Operation* que implementa. Esta função despoleta os pedidos necessários para a *API* interna da E-goi de forma a obter os dados essenciais para a resposta. Por fim a operação instância o modelo da resposta e retorna-o para o controlador;
- ***InternalAPIModel:*** representa o modelo gerado com os dados necessários para os pedidos a realizar à *API* interna. Ao ser instanciada, esta classe valida os dados enviados pelo utilizador, de forma a verificar se são sintaticamente ou semanticamente incorretos. De seguida transforma a sua estrutura para ser compatível com os serviços da *API* interna. A classe contém um método público *getJson* que retorna os dados na estrutura apropriada e no formato *JSON,* uma vez que este é o formato usado pela *API*. Esta classe implementa a interface *InternalAPIModel* para garantir que os métodos imprescindíveis são implementados;
- ***Model:*** é o modelo gerado utilizando os dados retornados pela *API* interna, esta classe implementa a interface *Model* para garantir que os métodos necessários são implementados.

Os dados da resposta da *API* Interna são transformados e validados nesta classe. As validações verificam se ocorreu algum erro *(e.g. Internal Server Error ou Conflict)* e caso não tenha ocorrido, transforma os dados para o formato da resposta. Por fim a classe possui o método *getResponse* que retorna a resposta em *JSON* ou *XML* consoante os *Headers* enviados no pedido do utilizador.

Existe ainda a seguinte classe que terá de ser criada em cada módulo (*e.g Contacts, Lists*), para que o sistema funcione:

- ***VersionManager***: possui a lista de serviços do seu módulo e para cada um, a lista de versões disponíveis assim como informação sobre qual é a última versão estável. Com base no nome do serviço e versão enviada pelo utilizador, esta classe dinamicamente instancia a operação pretendida, ou a sua última versão, caso não seja enviado nenhuma versão no *Header* ou uma mensagem de erro caso a versão enviada seja inválida. O sistema permite ainda a utilização de *wild-cards*, como será explicado na secção 5.2.3. Os dados sobre as versões dos serviços são atualizados manualmente por os desenvolvedores. Esta classe estende a classe *AbstractVersionManager* onde se encontra a implementação do método *getOperation* responsável por retornar a *operation* correta.

5.1.2 Disponibilização de *REST* e *SOAP*

Apenas o módulo Lists foi integrado com o sistema de versionamento como protótipo. Os restantes módulos não foram integrados com este sistema, devido a não haver uma decisão por parte da empresa sobre se o sistema de versionamento será uma funcionalidade da *API*. Nesta fase o foco da empresa está na disponibilização dos serviços. No futuro se for pretendido pela empresa, este sistema poderá ser facilmente utilizado nos restantes módulos, passando os serviços existentes, a ser considerados versão 1.0.0.

Nesta secção apresenta-se como os restantes módulos estão ligados aos serviços em comum, e como o módulo *Lists* foi ligado ao sistema de versionamento. Este processo pode ser replicado para os restantes módulos se a empresa o desejar.

A Figura 29 apresenta para o módulo exemplificativo *Contacts* se liga no protocolo *SOAP* e no estilo arquitetural *REST* aos serviços comuns. Os restantes módulos com exceção do módulo *Lists* funcionam da mesma forma. As classes representadas a amarelo são pertences à *Zend Framework 2*, a *framework* utilizada neste projeto, por restrição de implementação, como foi indicado na secção 4.2.2.

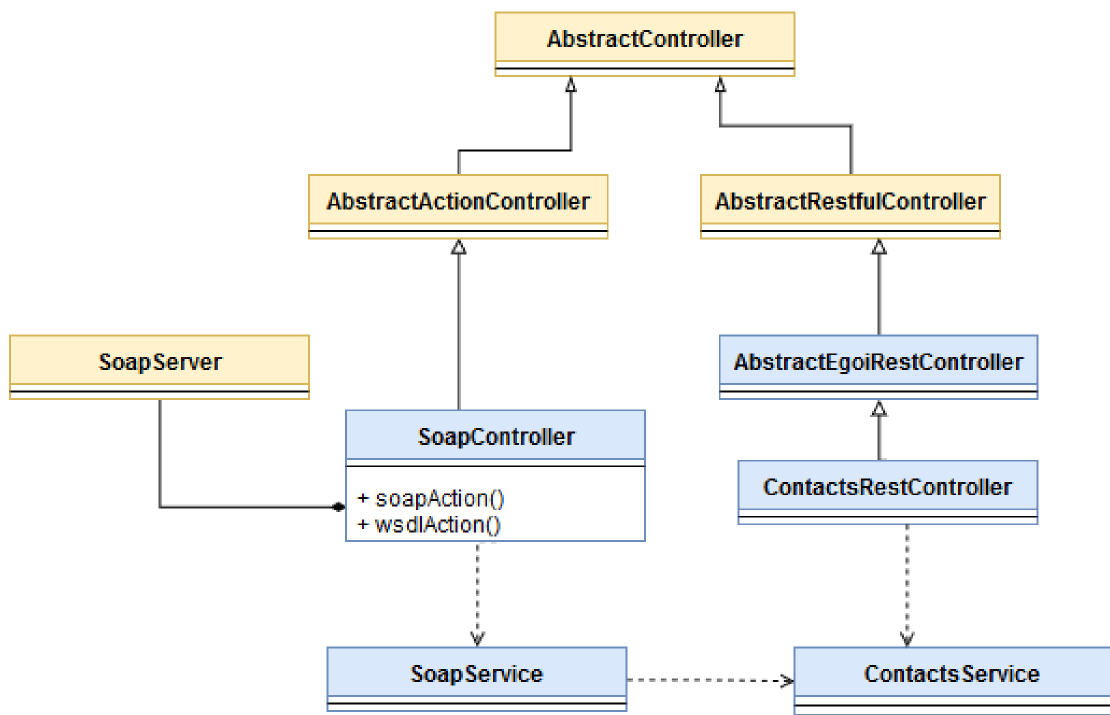


Figura 29 - Diagrama de classes da relação entre *REST* e *SOAP* com os serviços comuns

As seguintes classes são importantes para que se possa compreender como os sistemas foram implementados:

- ***AbstractRestfulController:*** possui os métodos necessários para direcionar os pedidos *REST*, com base no verbo *HTTP* utilizado. Os controladores *REST* de cada módulo devem estender essa classe, para poderem utilizar essa lógica, rescrevendo os métodos dos serviços pretendidos;
- ***AbstractEgoiRestController:*** esta classe é intermédia, entre o controlador da *Zend Framework* 2 e os controladores de cada módulo, foi criada para rescrever a mensagem de erro *Method Not Allowed* que ocorre quando um determinado método pretendido não existe. Para tal foi necessário rescrever todos os métodos referentes aos verbos *HTTP* e para serviços comuns;
- ***ContactRestController*****:** representa o controlador *REST* do recurso *Contacts* e estende a classe *AbstractEgoiRestController*. Para disponibilizar o serviço de criar um contacto (*POST: Contact*), foi necessário reimplementar o método *create* na subclasse. Os métodos reimplementados utilizam a classe *ContactsService* para fornecer os serviços;
- ***AbstractActionController:*** esta classe da *Zend Framework 2*, permite que as suas subclasses implementem métodos com o sufixo *action*, e que estes sejam chamados com base na ação enviada no *URL*;
- ***SoapController:*** é apenas implementada uma vez no projeto e estende a classe *AbstactActionController*. Implementa as funções *soapAction* e *wsdlAction* que permitem respetivamente utilizar as funcionalidades *SOAP* e retornar o ficheiro *WSDL* com a especificação. O método *soapAction* instância a classe *SOAPServer* com os serviços que estiverem disponíveis na classe *SoapService*;
- ***SoapServer:*** classe *PHP* que representa um servidor *SOAP* nos protocolos *SOAP* 1.1 ou *SOAP* 1.2 e suporta o uso de *WSDL* [51];
- ***SoapService:*** esta classe contém a lista de todos os serviços disponíveis na *API SOAP* e é usada para instanciar o *SoapServer*. O funcionamento desta classe é semelhante ao do controlador *REST* e é aqui que o serviço pretendido é chamado na classe ContactsServices;
- ***ContactsService:*** possui todos os serviços do módulo *Contacts*. Nos restantes módulos o funcionamento é semelhante.

5.1.2.1 Alternativa de Interligação com o sistema de versionamento

A Figura 30 apresenta as classes necessárias para entender como o estilo arquitetural *REST* e o protocolo *SOAP* foram integrados com o sistema de versionamento no módulo *Lists*. Dado que ambos estão implementados no mesmo projeto, é objetivo desta implementação reaproveitar as partes comuns.

A principal diferença para os módulos que não utilizam o sistema de versionamento, é que o controlador *REST* e o *SoapService* utilizam a classe Lists*VersionManager* em vez de usarem diretamente o serviço.

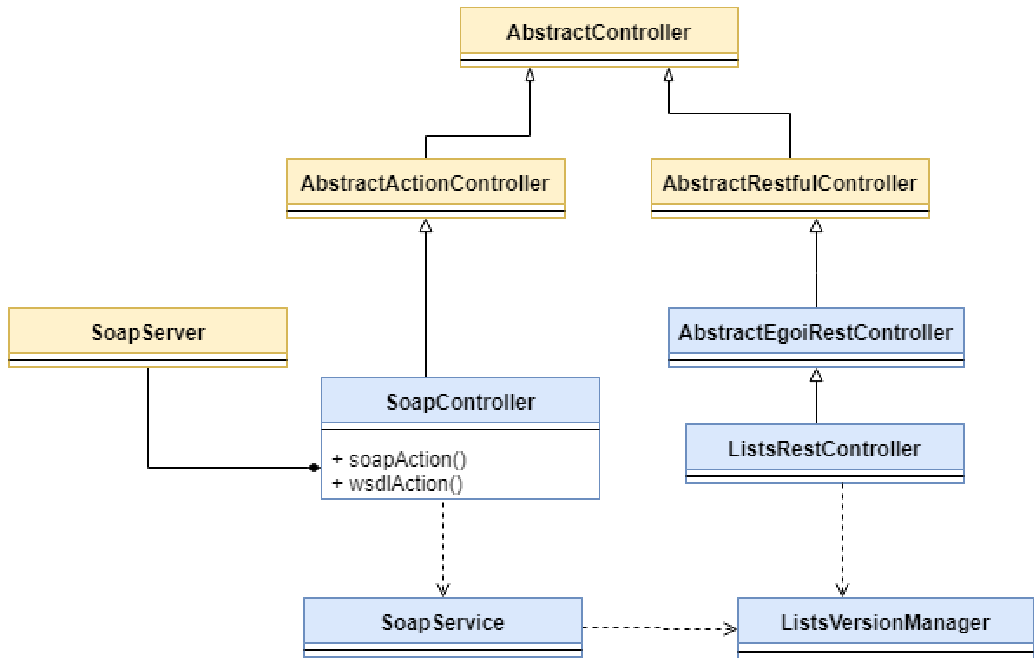


Figura 30 - Diagrama de classes da relação entre *REST* e *SOAP* com o sistema de versionamento

As classes são semelhantes ao que aconteceu no design sem sistema de versionamento apresentado na Figura 29. No entanto existem as seguintes diferenças:

- ***ListsRestController***: os métodos reimplementados nesta classe utilizam a classe *ListsVersionManager* para obter a *ListsOperation* correta, como foi explicado na secção 5.1.1, em vez de ligar diretamente ao serviço;
- ***SoapService:*** de forma similar esta classe utiliza os serviços da operação obtida a partir do *ListsVersionManager*;
- ***ListsVersionManager:*** possui a lista de serviços e versões do seu módulo assim como a responsabilidade de retornar a operação correta com base nos argumentos passados, tal como foi explicado na secção 5.1.1.

5.1.3 Sistema de erros

Tal como indicado na secção 1.3 e estudado na secção 2.2.1, um dos objetivos deste projeto é apresentar mensagens de erro com informações úteis para o utilizador. Em específico na *API REST,* devem ser utilizados os códigos *HTTP* apropriados.

Para criar este sistema foram identificados os erros que poderiam ocorrer e para cada um deles foi criado uma exceção personalizada. O sistema de erros apresentado utiliza as exceções personalizadas apresentadas na Figura 31, que estendem a classe *Exception* do *PHP*.

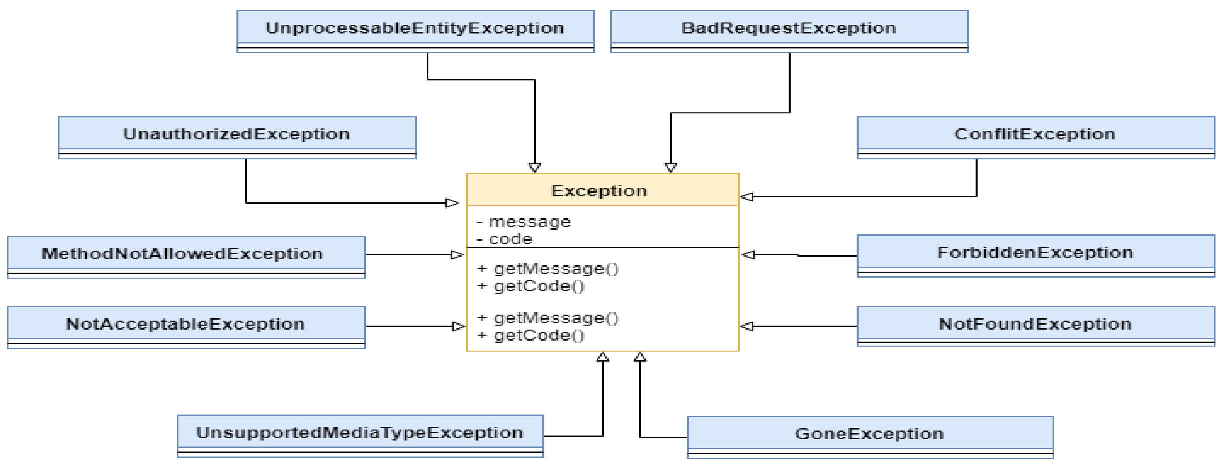


Figura 31 - Exceções personalizadas

Os erros detetados pela *API* são:

- *Unauthorized;*
- *Forbidden;*
- *Not Found;*
- *Method Not Allowed;*
- *Not Acceptable;*
- *Conflict;*
- *Unsupported Media Type;*
- *Unprocessable Entity*.

O erro *Internal Server Error* não necessita de exceção própria, uma vez que é tratado pela própria *framework*. As exceções possuem o argumento *message* onde são colocados os dados necessários para a criação da mensagem de erro personalizada e o atributo *code* que pode ser usado como um identificador do tipo de erro que ocorreu.

Cada tipo de erro tem o seu próprio comportamento e pode ter uma mensagem associada. Esta mensagem pode ser personalizada com as informações especificas do erro ocorrido ou genérica, dependendo do tipo de erro que ocorreu. A Figura 32 apresenta as classes *model* de cada um dos erros identificados. As classes estendem *APIErrorModel* de forma a garantir que possuem os seguintes atributos:

- ***Error:*** o nome do erro que ocorreu (*e.g. Forbidden*);
- ***Reason:*** o motivo de ocorrência do erro. que pode ser genérico ou personalizado, dependendo do tipo de erro que ocorreu. Por exemplo, no caso de num pedido os dados enviados serem inválidos, que corresponde ao erro *Unprocessable Entity*, a mensagem deve explicar em específico quais dados eram inválidos;
- ***Solution:*** uma mensagem a explicar o utilizador o que precisa de fazer para que o problema não ocorra. Esta mensagem pode conter um *URL* para a documentação;
- ***HTTPCode:*** representa o código *HTTP* do erro ocorrido. Este pode ser um dos seguintes (motivos de ocorrência estão explicados na secção 2.2.1):
    - **400:** *Bad Request*;
    - **401:** *Unauthorized;*
    - **403:** *Forbidden;*
    - **404:** *Not Found;*
    - **405:** *Method Not Allowed;*
    - **406:** *Not Acceptable;*
    - **409:** *Conflict;*
    - **410:** *Gone*;
    - ***415:*** *Unsupported Media Type;*
    - **422:** *Unprocessable Entity.*
- ***SOAPFaultType:*** indicar o tipo de *SOAP Fault*. Tipicamente será *SOAP-ENV:Client* para os erros em que a culpa é do utilizador.

Para criar as classes *Model* das mensagens de erros, utilizou-se o padrão *Factory* com a fábrica *APIErrorFactory*. Para cada uma das exceções apresentadas na Figura 31, foram criadas as classes modelo correspondentes, apresentadas na Figura 32.

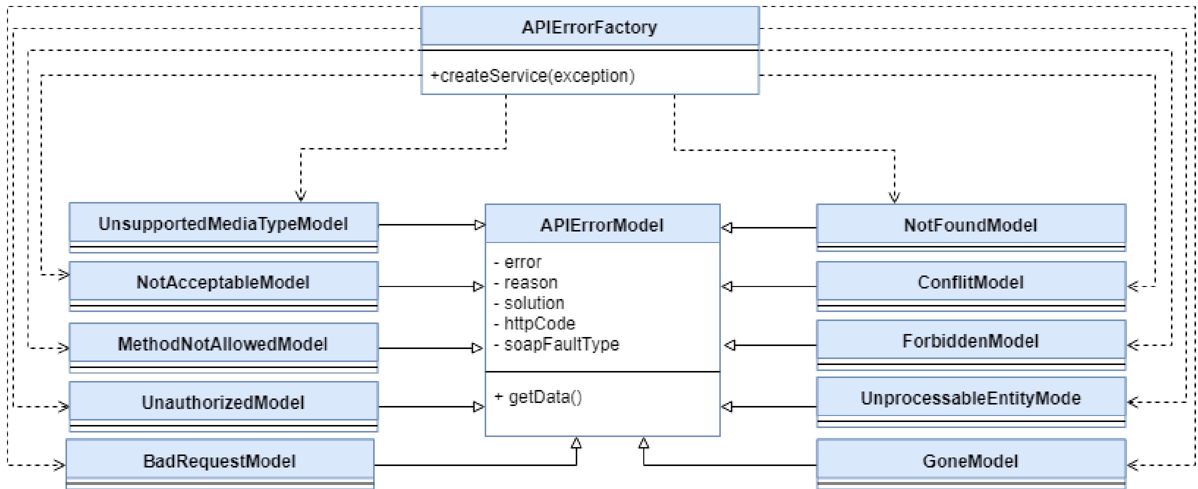


Figura 32 - Modelos de erros

A classe *"APIErrorFactory"* recebe através de uma injeção de dependência uma exceção e com base no seu tipo, instancia o modelo de erro correspondente. Cada modelo preenche os atributos *error, reason, solution* e *httpCode*, utilizando os dados contidos no atributo *message* da exceção recebida. As classes *model* de cada erro estendem a classe *APIErrorModel*. Desta forma há a garantia que possuem os argumentos e as funções necessárias ao seu correto funcionamento. O sistema apresentado pode ser usado pela *API REST* e pela *API SOAP* pois possibilita o retorno das respostas em diversos formatos. A função *getData* permite obter os dados do modelo num *array*.

5.1.4 Seleção do formato da resposta

O sistema deste projeto permite que os utilizadores utilizem os serviços através de uma *API REST* ou *SOAP*. No caso da primeira, o sistema pode possibilitar que a resposta seja enviada em diferentes formatos como *JSON* ou *XML*. No caso de *SOAP*, caso o pedido resulte num erro, é necessário retornar um *SOAP Fault* e no caso de sucesso deve ser retornado um *SOAP* Envelop. Estas diferenças criam a necessidade de adaptar a resposta consoante a situação. A Figura 33 apresenta um sistema que permite adaptar a resposta consoante o contexto.

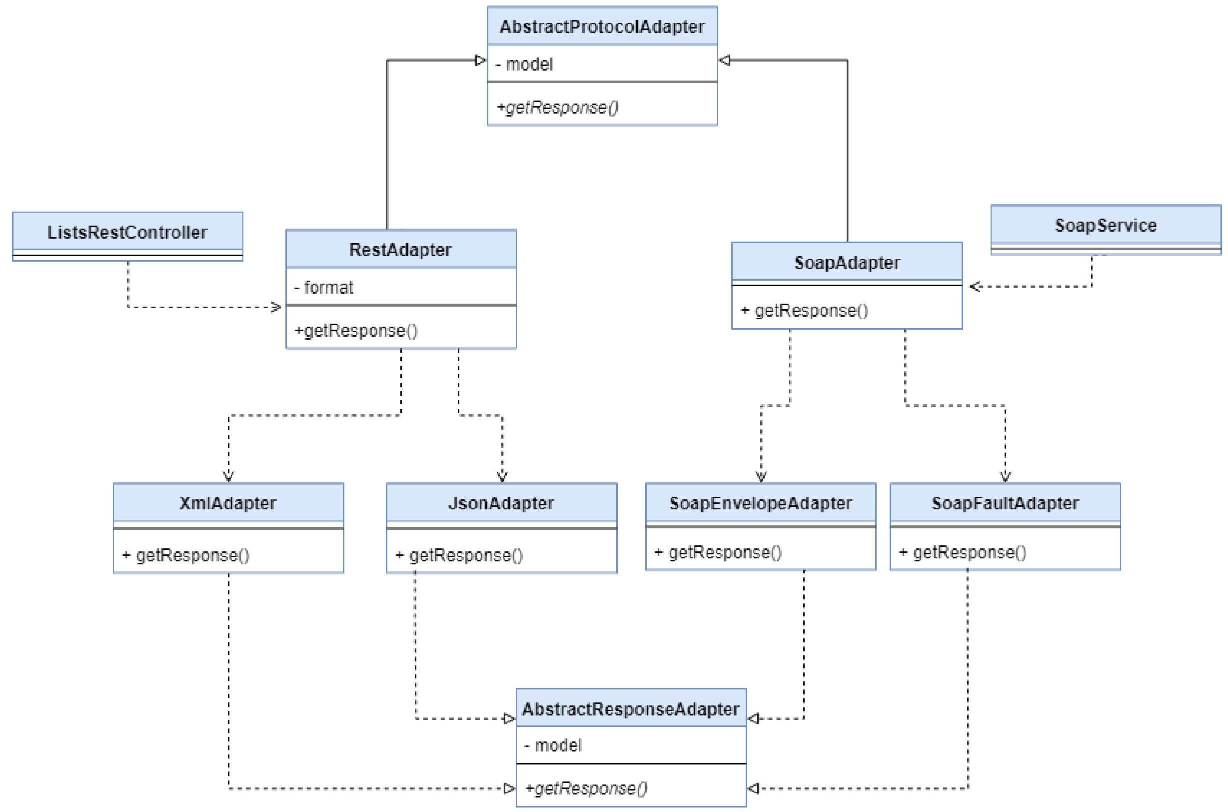


Figura 33 – Diagrama de classe do sistema de seleção do formato de resposta

As classes do sistema apresentado na Figura 33 são descritas seguidamente:

- ***AbstractProtocolAdapter:*** classe abstrata que permite a receção do *model* da resposta e possui o método abstrato *getResponse*;
- ***AbstractResponseAdapter:*** esta classe é abstrata e recebe permite a receção de um *model* de resposta assim como garante que as classes filho disponibilizem o método *getResponse*;
- ***SoapAdapter:*** esta classe é instanciada pela classe *SoapService* e recebe o *model* da resposta por *dependency injection*. Se o tipo do *model* recebido corresponder a um erro, utiliza a classe *SoapFaultAdapter*, caso contrário utilizada a classe *SoapEnvelopeAdapter*. Estas classes possuem a seguinte funcionalidade:
    - ***SoapEnvelopeAdapter:*** cria um *SOAP Envelope* com base no *model* recebido;
    - ***SoapFaultAdapter:*** cria uma *SoapFault* com as informações do erro e retorna-a no método *getResponse*.
- ***RestAdapter:*** possibilita que a *API REST* retorne as respostas em vários formatos. No exemplo apresentado na figura, é possível retornar em JSON ou XML. Esta classe instancia o adaptador correspondente e envia o modelo.

5.2 Implementação

Nesta secção são apresentados os detalhes e decisões de maior relevo, relativos à implementação da solução. Em específico a forma como a disponibilização de *REST* e *SOAP* foi executada nos módulos da *API*. O sistema de versionamento apenas foi implementado no módulo Lists, uma vez ainda foi decidido se a nova *API* vai disponibilizar oficialmente essa funcionalidade, como foi indicado na secção 5.1.2. Nesta secção apresentam-se mais detalhes sobre o sistema de versionamento e o fluxo ocorrido desde a realização do pedido até à resposta final para o módulo implementado.

A implementação utilizou a *Zend Framework 2* como *framework* de desenvolvimento, uma vez que esta era uma restrição da empresa. A *Zend Framework* é uma coleção de packages profissionais em *PHP* que providencia código orientado a objetos. Esta tecnologia utiliza o *Composer* como gestor de dependência de módulos *PHP* e o *PHPUnit* como *framework* de testes aos módulos implementados. A tecnologia foi instalada por mais de 200 milhões de vezes e é utilizada para o desenvolvimento de aplicações *web* e de *API* [52].

5.2.1 Serviços implementados

A Figura 34 lista as funcionalidades implementadas que foram apresentadas na secção 4.2.1. Estas funcionalidades foram previamente especificadas no formato *Open API* por outros elementos, como indicado na secção 1.3 e a implementação seguiu a estrutura especificada.

Figura 34 - Lista de serviços da *API*

5.2.2 Disponibilização de *REST* e *SOAP*

O desenvolvimento da solução seguiu o *design* apresentado na secção 5.1.2 e primeiramente foi implementado o sistema *REST*. Para disponibilizar este estilo arquitetural é necessário que os controladores herdem algumas funções da classe *AbstractRestfulController* disponibilizada pela *framework*. Dos métodos disponibilizados pela classe foram utlizadas as funções *create*, *delete*, *get* e *update* correspondentes aos verbos *HTTP POST*, *DELETE*, *GET* e *PUT*.

Este sistema necessita que pelo menos um controlador seja criado por cada recurso e dependendo do módulo em questão, rescrever os métodos a disponibilizar. A Figura 35 apresenta de forma exemplificativa como o sistema *REST* se interliga com os serviços partilhados com *SOAP*. Por exemplo, se o utilizador fizer um *HTTP GET* para o recurso *Contacts*, é executado o serviço *getContacts* da classe *ContactsServices*.

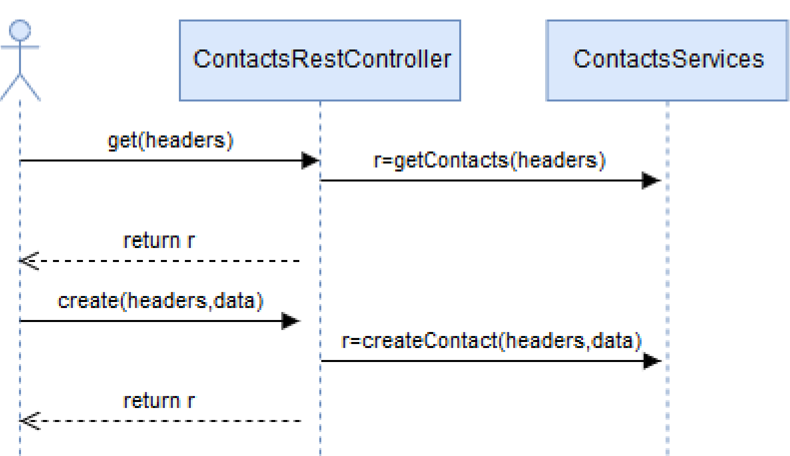


Figura 35 - Diagrama de sequência do sistema *REST* interligado aos serviços

A disponibilização do protocolo *SOAP* foi realizada utilizando a classe *SoapServer* da *Zend Framework*. Esta classe disponibiliza a opção de ser utilizada em modo *WSDL* ou ser preenchida com uma classe que contenha os serviços a disponibilizar. A *Zend Framework* também disponibiliza a classe *AutoDiscovery* para a geração automática do *WSDL*, a partir do *SoapServer* ou a classe *Wsdl* para a construção manual [53].

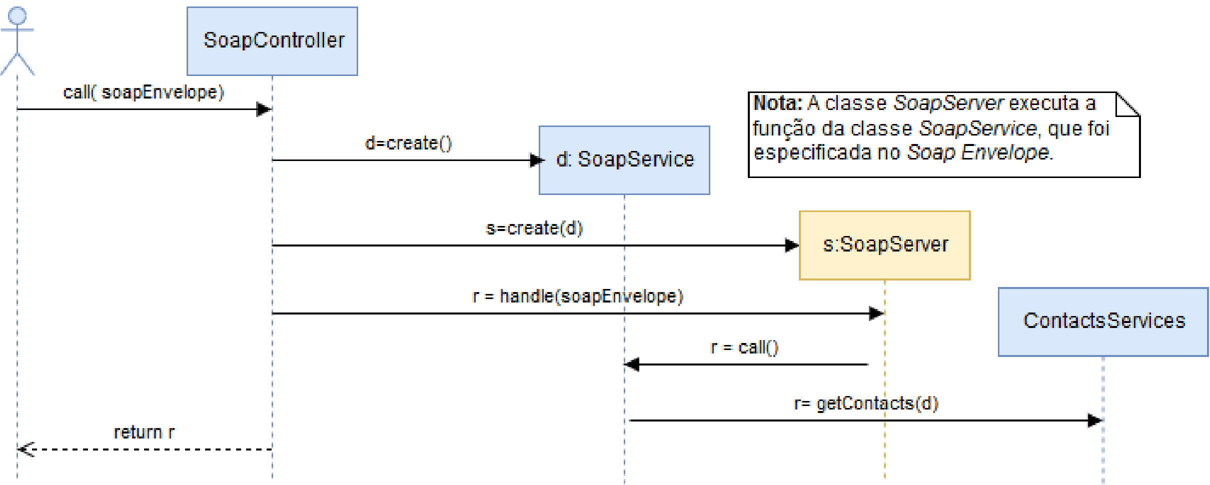


Figura 36 - Diagrama de sequência do sistema *SOAP* interligado aos serviços

Contrariamente ao que acontece na implementação do estilo arquitetural *REST*, foi tomada a decisão após dialogar com responsáveis da empresa, que apenas seria disponibilizada uma rota *SOAP* através da qual todos os serviços disponíveis poderiam ser chamados. Para esse fim foi criada a classe *SoapService* onde se encontram a lista de serviços disponíveis e é este objeto que é injetado no *SoapServer* para o preencher.

5.2.2.1 Disponibilização de REST e SOAP com o sistema de versionamento

Como explicado na secção 5.1.2 o sistema de versionamento só foi implementado como protótipo, para o módulo Lists, uma vez que ainda não foi decidido se a *API* vai disponibilizar este sistema. No entanto a implementação neste módulo serviu para validar se o sistema funciona.

Cada método implementado nos controladores utiliza o *VersionManager* do seu módulo para obter a versão do serviço pretendida pelo utilizador. A integração com este sistema será explicada em mais detalhe na secção 5.2.4 onde será apresentado um diagrama de como este sistema se integra com o sistema de versionamento por serviço.

A Figura 37 apresenta um diagrama de sequência exemplificativo da implementação dos serviços que permitem obter todas as listas de contactos ou criar uma lista de contacto, integrado com o sistema de versionamento.

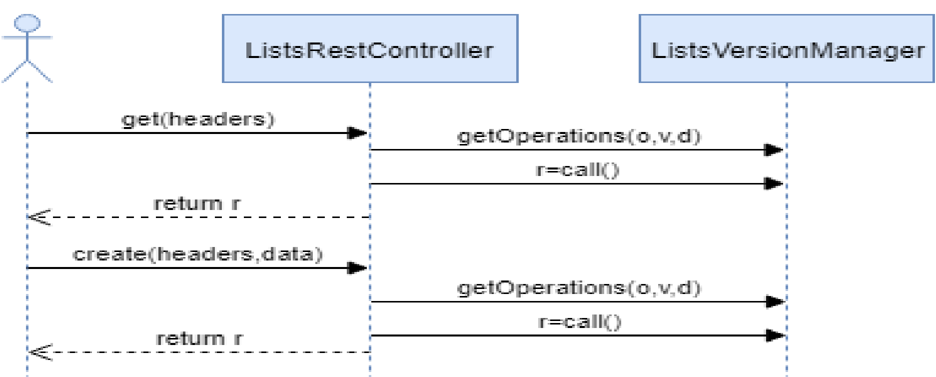


Figura 37 - Diagrama de sequência de interligação do sistema *REST* com o sistema de versionamento

Na Figura 38 exibe-se como o sistema SOAP ser interliga com o sistema de versionamento. Os serviços da classe *SoapService* chamam o *Version Manager* apropriado, tal como ocorre em *REST*. O fluxo a partir dessa classe para a frente é comum em ambos, tal como será explicado em mais detalhe na secção 5.2.4.

Com esta implementação não é possível gerar o *WSDL* automaticamente em *SOAP,* uma vez que os serviços não podem ser descritos para todas as versões automaticamente. No entanto, devido a uma das características principais deste projeto ser a disponibilização dos mesmos serviços em *SOAP* ou *REST*, é possível gerar o *WSDL* a partir da especificação *OpenAPI*.

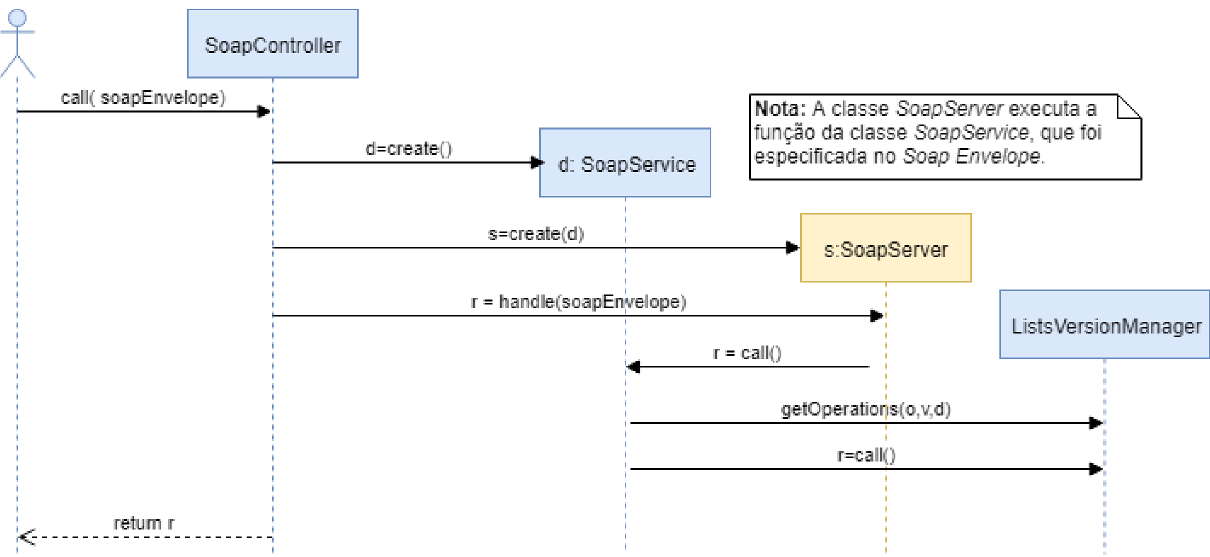


Figura 38 - Diagrama de sequência de interligação do sistema *SOAP* com o sistema de versionamento

5.2.3 Sistema de versionamento

Durante o desenvolvimento do sistema de versionamento exclusivamente para o módulo *Lists*, como indicado na secção 5.1.2, foi necessário tomar algumas decisões mais detalhadas sobre como este iria funcionar. Foi essencial decidir de que forma seriam passados os argumentos da versão pretendida e como funcionariam os *wild-cards*. De seguida foi decidido de que forma a lista de versões disponíveis é atualizada nas classes *VersionManager*.

5.2.3.1 Seleção da versão pretendida

No contexto deste projeto, dado que o sistema de versionamento deve funcionar em *REST* e *SOAP* e que ambos possuem *headers* (*HTTP* headers no caso de *REST* e *SOAP headers* no caso de *SOAP*), os dados do versionamento serão passados nos *headers*. Outro motivo para esta decisão é que a versão pretendida, é um argumento é opcional e se o utilizador não a passar, assume-se que pretende a última versão.

Esta opção permite também a disponibilização de várias estratégias de seleção com a utilização de *wild-cards*. Por exemplo, se o utilizador passar o argumento "1.*", poderá indicar que quer a versão mais recente dentro da "*major* 1". Por estes motivos a utilização de *headers* personalizados é a solução preferida nesta situação. O envio desta informação é realizado através do *header ServiceVersion,* criado de forma personalizada para este propósito.

O sistema de versionamento possibilita o uso de wild-cards para selecionar as versões com base nos argumentos enviados. Por exemplo, se o utilizador desejar a última versão, pode enviar apenas o símbolo "*". Se pretender a última versão da *major* 3, pode enviar "3.*". Caso o *header* não seja enviado, o sistema assume que o utilizador pretende a última versão disponível do serviço. Por último, se o valor enviado for sintaticamente incorreto ou referente a uma versão que não existe, é retornada uma mensagem de erro. O sistema assume que à direita de um *wild-card* nunca pode estar um número.

De forma a facilitar o entendimento do comportamento do sistema com base no *header enviado, a* lista apresenta lista como o sistema responde aos pedidos, com base em diversos exemplos de valores para o *header ServiceVersion*, para um serviço com as seguintes características:

- **Versão atual:** 3.1.0.
- **Versões anteriores:** 1.0.0; 1.1.0; 1.1.1; 1.1.2; 2.0.0; 2.1.0; 3.0.0.

A Tabela 14 apresenta o comportamento do sistema do versionamento, com base em exemplos de valores do *header ServiceVersion*, tendo em conta as versões disponíveis.

Tabela 14 - Comportamento do sistema de versionamento com base no valor do *header*

Valor do *header*	**Retorno**
Não envia o *header*	Versão 3.1.0
*	Versão 3.1.0
1.1.1	Versão 1.1.1
1.1.*	Versão 1.1.2
1.2.*	Erro porque a versão não existe
2.*	Versão 2.1.0
2.*.*	Versão 2.1.0
2.*.*.*	Versão 2.1.0
*.1	Erro porque o formato do *header* é invalido

5.2.3.2 Version Manager

O sistema de versionamento é responsável por garantir que a versão do serviço pretendida pelo utilizador é executada. Essa responsabilidade pertence mais especificamente às classes *VersionManager* de cada módulo e requerem alguma atenção especial dos desenvolvedores para que a lista de versões e serviços esteja atualizada.

O extrato de código 6 apresenta de forma exemplificativa como esta atualização ocorre no módulo *Lists*. A variável *versions* é um *map* cuja *key* é o nome do serviço e o *value* é um vetor com a lista de versões disponíveis. De forma similar a variável *deprecatedVersions* guarda a lista de versões descontinuadas. Estas variáveis são atualizadas pelos desenvolvedores, nos *VersionManagers* apropriados.

```
$versions =[
     "CreateList" => ["2.0.0" ],
     "DeleteList" => [ "2.0.0" , "2.1.0" , "3.0.0" ]
 ];

$deprecatedVersions=[
     "CreateList" => [ "1.0.0" , "1.0.1" ],
     "DeleteList" => [ "1.0.0" , "1.0.1" , "1.1.0" , "1.1.1" ]
 ];
```

Código 6 - Exemplo da especificação da lista de versões de cada serviço

O método *getOperations* implementado na classe *AbstractVersionManager*, como indicado na secção 5.1.1, possui a responsabilidade de escolher a versão. Os argumentos que recebe são o nome do serviço, o *header ServiceVersion*, caso tenha sido enviado e os dados do serviço, caso seja preciso. No caso de *REST*, o nome do serviço está especificado nas funções dos *controllers* e no caso de *SOAP* nas

funções da classe *SoapService*. Com base nos dados recebidos, a função *getOperations* analisa o *header ServiceVersion* de forma a verificar se é um *wild-card* ou uma versão específica. Caso seja um *wild-card* verifica se existe alguma versão correspondente ou se este é invalido e caso seja uma versão particular verifica se ela existe. Se ocorrerem erros são lançadas exceções com a informação relevante que são capturadas pelos controladores. Caso o *wild-card* seja inválido é lançada uma exceção *BadRequestException*, se a versão pretendida já tenha sido descontinuada é lançada uma *GoneException* e se não existir *NotFoundException*.

5.2.4 Visão geral dos sistemas

A Figura 39 apresenta um diagrama de sequência exemplificativo do fluxo de execução do sistema de versionamento na *API REST*. As validações dos dados nas classes modelo, em caso de erro lançam uma exceção personalizada (5.1.3) que é utilizada pelo controlador. As classes a azul são especificas do estilo *REST*.

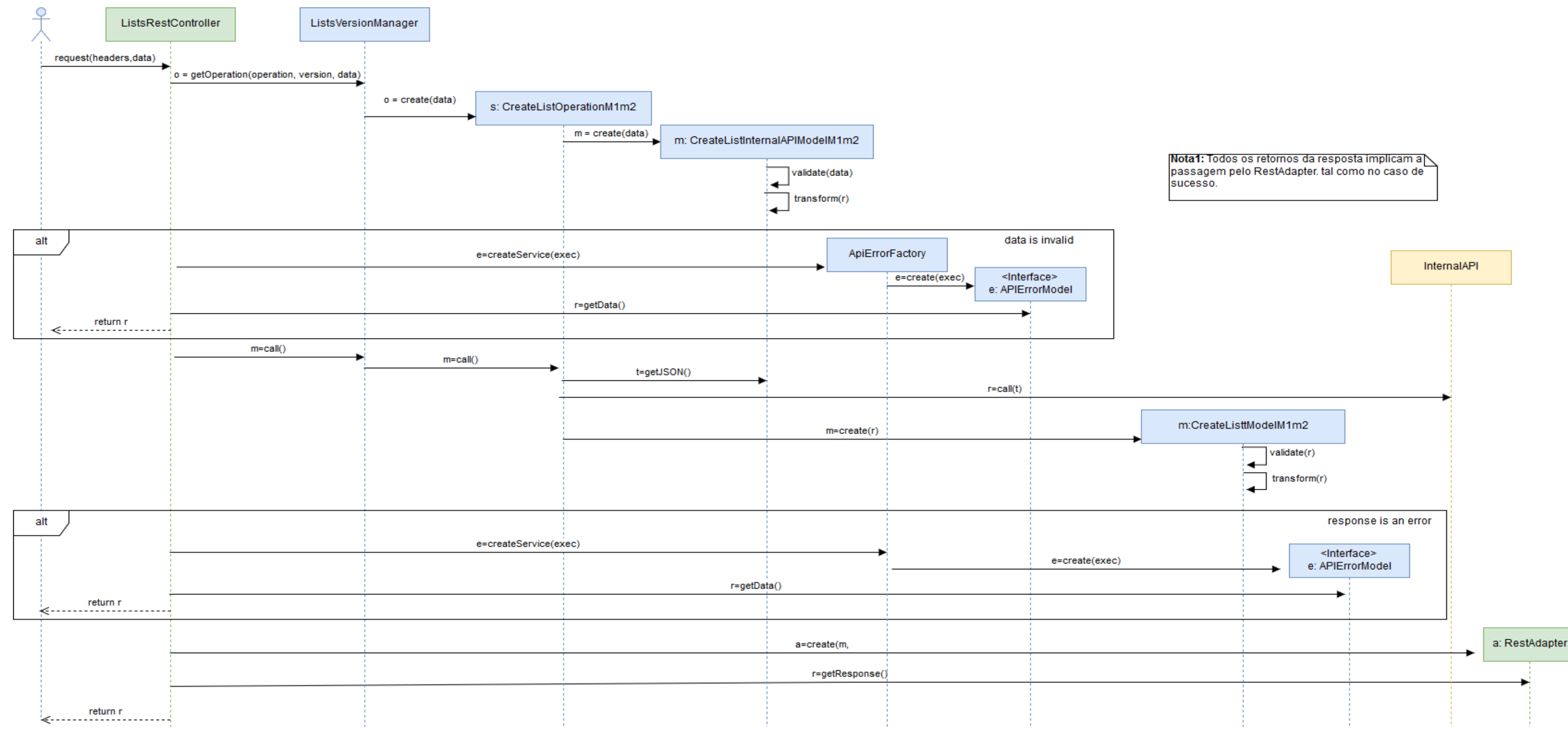


Figura 39 - Diagrama de sequência da utilização do sistema de versionamento em *REST*

De forma semelhante, a Figura 40 apresenta como *SOAP* utiliza o sistema de versionamento, no entanto para simplificar o diagrama, tal só foi explicado a partir do momento em que o pedido chega à classe *SoapService*, como apresentado na secção 5.2.2. As classes a verde são específicas do protocolo *SOAP*.

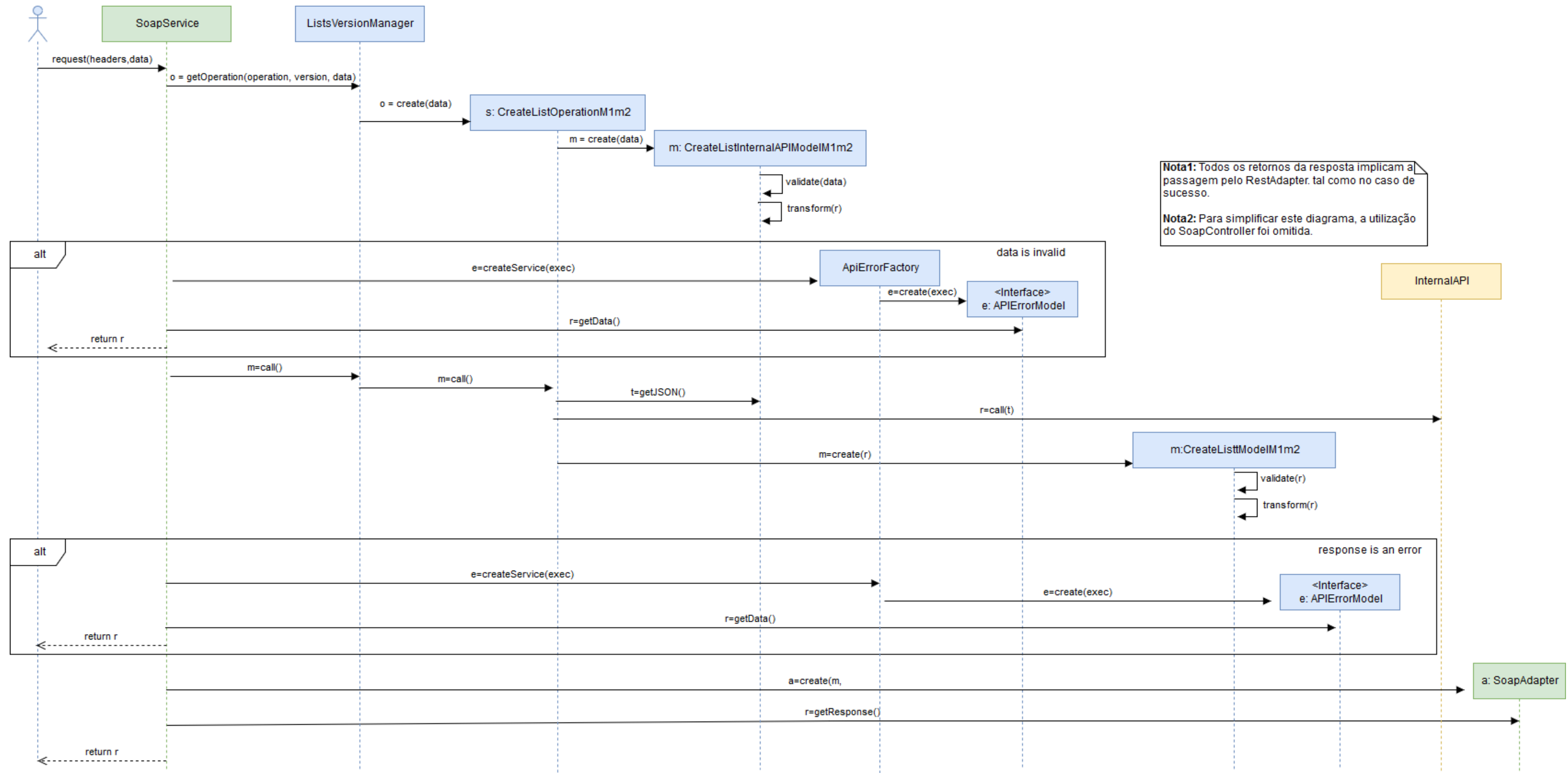


Figura 40 - Diagrama de sequência da utilização do sistema de versionamento em *SOAP*

Independentemente se o utilizador está a utilizar a *API REST* ou a *SOAP*, o sistema de versionamento é o mesmo para ambos. No caso de *REST* o fluxo é iniciado quando é solicitado o *controller REST* do módulo que representa o recurso pretendido. Nesta fase pode ocorrer o lançamento de uma exceção correspondente ao erro *Unsupported Media Type,* após análise dos *headers* enviados pelo utilizador. Em *SOAP* o fluxo de execução até ao serviço pretendido na classe *SOAP Service* foi apresentado na secção 5.2.2. O fluxo de execução seguinte é comum em ambos até ser necessário retornar a resposta para o utilizador.

Seguidamente o método *getOperation* é chamado na classe *VersionManager* apropriada. O método recebe o nome do serviço, a versão pretendida, caso seja enviada e os dados necessários, se existirem. Se a versão enviada não existir podem ocorrer os erros indicados na secção 5.2.3.2. Caso exista a *Operation* instancia o *InternalAPIModel* adequado e os dados são validados e transformados. Esta validação pode resultar no lançamento de uma exceção indicativa do erro *Unprocessable Entity,* caso os dados sejam inválidos.

Seguidamente a *Operation* é chamada e o *JSON,* necessário para realizar o pedido à *API* interna da E-goi, é obtido do *InternalAPIModel*. Um *model* de resposta é instanciado utilizando a resposta da *API* privada e estes dados são validados e transformados. Nesta fase, alguns erros que não podem ser verificados previamente, sem realizar um pedido à *API* privada, podem ser detetados e as exceções correspondentes podem ser lançadas. Os erros que podem ocorrer são o *Forbidden*, *Not Found*, *Conflit* e *Internal Server Error* caso este seja o retorno da *API* privada.

Por fim o fluxo diverge entre *REST* e *SOAP* e em caso de sucesso será instanciado respetivamente um *RestAdapter* ou um *SoapAdapter,* de forma a obter a resposta final no formato pretendido. Se o formato pretendido em REST não estiver disponível poderá ocorrer o lançamento da exceção referente ao erro *Not Acceptable*.

As exceções que foram lançadas previamente são capturadas no controlador *REST* ou na classe *SoapService* e são utilizadas para instanciar um modelo de resposta de erro, através da utilização da classe *APIErrorFactory* como explicado na secção 5.1.3.

Para os serviços que não estão a utilizar o sistema de versionamento, em vez de o pedido ser encaminhado para o *VersionManager* para obter a *Operation* correta, é diretamente chamado o serviço da classe Service do módulo e processo restante é comum aos diagramas apresentados.

5.2.5 Resultados dos testes de software

Nesta secção apresentam-se detalhes de como foram realizados os testes de aceitação e do tempo de resposta da *API*, assim como os resultados obtidos.

5.2.5.1 Testes de aceitação

A especificação de uma *API* define os argumentos de entrada e de resposta de cada serviço e as respostas em caso de sucesso ou de erros. Os testes devem validar as várias respostas da *API* em diversos cenários com argumentos válidos ou inválidos [11] e devem permitir responder a questões como as seguintes [11]:

- Qual é o comportamento por defeito da *API,* quando nenhum argumento de *query* é passado?
- Qual é o comportamento da *API* quando os parâmetros de *query* são passados com os valores corretos?
- Qual é comportamento da *API* quando um argumento é passado sem nenhum valor?
- Qual é comportamento da *API* quando um argumento é passado com um valor incorreto?
- Qual é o formato de dados por defeito quando nenhum formato é especificado no pedido?
- Quais são os códigos *HTTP* para diferentes casos de sucesso ou insucesso?
- Qual o comportamento quando um utilizador sem permissões tenta efetuar o pedido?

Na nova *API* quando nenhum argumento de *query* é transmitido, é assumido um valor por defeito e é esperado que a resposta seja de sucesso. Se for passado um argumento inválido ou um argumento obrigatório não for transmitido, é retornado *Unprocessable Entity ou uma SOAP Fault* com a mesma informação. Dependendo do serviço os códigos *HTTP* para o estilo arquitetural *REST*, no caso de sucesso ou no caso de erros variam.

Os serviços *REST* foram testados utilizando os testes de aceitação, gerados de forma automática a partir da especificação *Open API*, como explicado na secção 1.3. Os testes *SOAP* foram gerados de forma automática a partir dos testes *REST*, uma vez que os serviços são comuns e a única diferença é o formato dos dados em que são transmitidos. Estes testes, utilizam o *PHPUnit* como *framework* de testes e verificam a coerência entre os serviços *SOAP* e *REST* e a coerência da implementação do estilo arquitetural *REST* com a especificação *Open API.* No futuro, quando o ficheiro *WSDL* for gerado automaticamente, os testes SOAP poderão ser gerados diretamente utilizando o ficheiro.

Por exemplo, para verificar o serviço *get automations*, é necessário realizar testes com os argumentos de *query offset*, *limit*, *orderBy* e *sort.* Os testes a este serviço cobrem os casos em que podem ocorrer os códigos de erro 401, 403, 406 e 422 assim como o caso de sucesso. Para este serviço são realizados os seguintes testes:

- Os argumentos de *query* não são passados e é esperado que assumam um valor por defeito e a resposta seja retornada com sucesso;
- Os parâmetros de *query* são passados corretamente e é esperada uma resposta de sucesso coerente com a especificação;
- Os argumentos de *query* são passados com valores inválidos e é esperado que ocorra o erro *Unprocessable Entity;*
- O pedido é realizado com uma *apikey* inválida e é esperado que ocorra o erro *Unauthorized;*
- Utilizando uma *apikey* de um utilizador sem permissões é esperado que ocorra o erro *Forbidden;*
- Se transmitido o *header Accept* com um valor inválido, é esperado que ocorra o erro *Not Acceptable.*

Se todos estes testes tiverem o resultado esperado, é possível concluir que o serviço *get automations* está implementado de forma coerente com a especificação. Estes testes foram realizados sempre que um *commit* era feito para o repositório, dado que são executados num *stage* de uma pipeline *Jenkins* criada para este projeto.

Após concluída a implementação inicial, foram realizados testes a todos os serviços tanto para o estilo arquitetural *REST*, como para o protocolo *SOAP*. Todos os testes passaram com sucesso pelo que se pode concluir que a implementação está coerente com a especificação.

Para validar o sistema de versionamento foram criados e realizados testes específicos para o módulo *Lists*, uma vez que apenas esse módulo possui esse sistema implementado. Para tal foram criadas as versões 1.0.1 e 2.0.0 de cada um dos serviços do módulo. A versão 1.0.0 foi colocada como *deprecated.* Os testes validaram a resposta quando o *header* da versão é passado num formato incorreto, e que quando é pedido uma versão *deprecated* é devolvido o erro correspondente. Foi ainda validado que era possível pedir a versão 1.0.1 passando "1.*" e 1.0.1. Por fim testou-se que se a versão 2.0.0 era executada quando o *header* não era passado.

Estes resultados validam a hipótese 1, devido a todos os testes terem passado e a hipótese 2, apresentadas na secção 6.2 ,uma vez que todos os serviços estão cobertos por testes, que são gerados automaticamente. Devido a estes testes estarem a ser executados num *stage* de uma *pipeline* do Jenkins sempre que um serviço é alterado e a especificação reflete esta mudança, esse serviço é automaticamente testado através de um novo teste.

5.2.5.2 Testes ao tempo de resposta dos serviços

De forma a validar a hipótese 3, apresentada na secção 6.2 ,é necessário realizar pedidos a todos os serviços disponibilizados na nova *API* e aos serviços correspondentes na *API* privada da E-goi. Para executar estes foi utilizada a ferramenta JMeter e criado um plano de testes para cada uma das *API*.

Para cada um dos planos, primeiramente foi configurada uma *thread group* com um *loop count* de 30, para realizar cada um dos pedidos 30 vezes, e *number of thread*s como valor 1 para simular 1 utilizador. De seguida foi configurado um *HTTP Header Manager* onde foi especificado o *header* da *apikey*.

Seguidamente, para cada um dos serviços a testar foi adicionado e configurado um *HTTP R*equest e colocadas as rotas e parâmetros de *query* corretos. Por fim foi adicionado um *listener View Result Tree* para visualizar os valores de respostas de forma a encontrar potenciais erros.

Foram realizados pedidos a cada um dos serviços disponibilizados, primeiramente na *API* pública e de seguida correndo o plano de testes da privada. Este plano chama os serviços da privada que internamente são chamados pela pública. Os parâmetros de *query* enviados foram os mesmos para ambas as *API*.

Os dados obtidos foram utilizados para verificar estatisticamente se existem diferenças entre as médias e qual o valor. Uma vez que ambas as amostras possuem pelo menos 30 serviços, é possível concluir que as amostras se aproximam de uma distribuição normal.

Para verificar se as variâncias são homogéneas, foi realizado um teste de Fisher utilizando a ferramenta R Studio, que facilita a utilização da linguagem R, que permite a realização de testes estatísticos e geração de gráficos. De seguida apresenta-se o resultado do teste e uma vez que o valor p é inferior a 0.05, não é possível realizar um teste t para duas amostras independentes, dado que esse é um dos requisitos [54]:

```
            F test to compare two variances

data: RServices and RV3
F = 0.73505, num df = 1559, denom df = 1559, p-value = 1.327e-09
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
  0.6655559 0.8117986
sample estimates:
ratio of variances
          0.7350492
```

Com os resultados obtidos é possível realizar teste t de Welch para duas amostras, uma vez que não possui o requisito da homogeneidade de variâncias [54]. As hipóteses testadas são as seguintes:

$$h0: \mu 1 = \mu 2$$

$$h1: \mu 1 \neq \mu 2$$

De seguida apresenta-se o resultado do teste realizado, onde x são os dados da resposta direta à *API* privada e y os dados temporais dos pedidos da *API* pública:

```
            Welch Two Sample t-test

data: RServices and RV3
t = -25.202, df = 3046.9, p-value < 2.2e-16
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -37.82320 -32.36269
sample estimates:
mean of x mean of y
 291.7218  326.8147
```

Uma vez que o valor de p é inferior a 0.05, a hipótese h0 é rejeitada. Assim verifica-se que as existem diferenças temporais entre as médias dois grupos, pelo que é possível calcular o aumento de tempo, com base no intervalo de confiança, que varia entre -37.82320 e -32.36268.

Utilizando uma regra de 3 simples conclui-se com 95% de confiança que o aumento temporal, causado pelo uso da *API* pública em relação a um pedido direto à *API* interna, está entre 9.9% e 11.57%, pelo que a hipótese 3 foi validada.

6 Avaliação da Solução

Neste capítulo são apresentadas e explicadas as grandezas avaliadas, as hipóteses testadas, as metodologias utilizadas e os respetivos testes estatísticos realizados.

6.1 Grandezas

Identificados os problemas e objetivos é possível identificar as grandezas que serão analisadas para testar a solução. Identifica-se as seguintes grandezas:

- **Grau de cobertura do código por testes:** a percentagem de serviços coberta por testes. Esta grandeza envolve o problema da versão 2 se encontrar mal estruturada;
- **Testes bem-sucedidos:** a percentagem de testes bem-sucedidos;
- **Satisfação dos utilizadores:** a satisfação dos utilizadores com a nova *API* e com a anterior. Esta será analisada avaliando seguintes métricas:
  - **Tempo de resposta:** o tempo de resposta da *API* é uma das métricas que pode afetar a satisfação de um cliente. Nessa medida faz sentido analisar o tempo de resposta à *API* pública em relação a um pedido direto às *API* internas o E-goi;
  - **Satisfação com a facilidade de aprendizagem:** opinião sobre o quão fácil é aprender a utilizar a *API*. Se a *API* estiver bem documentada e seguir as especificações dos estilos arquiteturais utilizados, a aprendizagem deve ser facilitada;

- **Satisfação com a facilidade de utilização:** apreciação sobre o quão simples é utilizar a *API* e se as mensagens de erro são percetíveis;
- **Satisfação com a quantidade de funcionalidades:** opinião sobre a quantidade das funcionalidades do E-goi que foram disponibilizadas na *API* pública versão 3;
- **Satisfação com a utilidade das funcionalidades:** apreciação sobre o quão bem as funcionalidades disponibilizadas cumprem as necessidades dos clientes;

6.2 Hipóteses

Com base nas métricas referidas, pretende suportar os resultados do trabalho realizado com base nas hipóteses apresentadas na Tabela 15.

Tabela 15 - Hipóteses

Identificação	**Hipótese**
Hipótese 1	Pelo menos 95% dos serviços está coberto por testes
Hipótese 2	100% dos serviços testados funcionam corretamente
Hipótese 3	O valor médio da diferença temporal causada pelo uso da *API* púbica em *REST* em relação à utilização direta da *API* interna do E-goi é menor do que 20%
Hipótese 4	A satisfação geral dos utilizadores é superior na versão 3 da *API* em relação à 2. A satisfação será medida tendo em conta a opinião dos utilizadores sobre os tempos de resposta, facilidade de aprendizagem, facilidade de utilização, quantidade e qualidade das funcionalidades disponibilizadas

As hipóteses 1 e 2 cobrem as métricas grau de cobertura do código por testes e Testes bem-sucedidos. Os testes vão ser realizados em todos os serviços e também ao sistema de versionamento. A hipótese 3 envolve a métrica tempo de resposta, que influencia a satisfação dos utilizadores. Por fim a hipótese 4 envolve todas as métricas apresentadas na secção 6.2, que influenciam a satisfação dos utilizadores.

6.3 Metodologia

Os dados para cada uma das hipóteses foi obtidos no último mês após o desenvolvimento ter sido concluído. A metodologia utilizada de modo a obter os dados necessários para testar as hipóteses 1 e 2 (especificados na secção 6.2), passou por utilizar o *PHPUnit* para gerar um relatório com a execução de testes de aceitação.

Os testes de aceitação que foram executados, foram gerados automaticamente a partir da especificação *Open API* e foram criados por outros elementos. Estes testes são executados num *stage* de uma pipeline do *Jenkins* sempre que um *commit* é feito para o repositório. No entanto os resultados apresentados serão da execução manual destes testes, após a conclusão do desenvolvimento.

Os testes ao sistema de versionamento foram criados manualmente à parte para esse propósito, e executados apenas no módulo *Lists*, uma vez que este sistema só foi implementado neste módulo como explicado na secção 5.1.2. No entanto este sistema pode ser utilizado nos outros módulos se a E-goi pretender no futuro.

Relativamente à hipótese 3, a abordem foi utilizar as médias obtidas através da realização de pedidos, tanto à nova *API* pública, como ao serviço ou serviços diretamente correspondentes nas *API* internas do E-goi. Uma vez que este teste não se trata de um teste de carga, realizou-se este processo apenas 30 vezes por serviço, de forma a obter o valor médio do tempo de resposta da *API* pública e da privada.

De seguida foi realizado um teste de Fisher de forma a verificar se as variâncias são homogéneas. Caso fossem seria realizado um teste t para duas amostras não emparelhadas e caso não fossem seria efetuado um teste t de Welch para duas amostras. Os resultados permitiram concluir se existem diferenças entre as médias e os valores do intervalo de confiança, e foram utilizados para descobrir um intervalo do aumento que ocorreu entre as médias do tempo de resposta e validar ou rejeitar a hipótese 3. Os resultados dos testes relativos ás primeiras três hipóteses foram apresentados na secção 5.2.5.

Os dados para testar da hipótese 4 foram obtidos através da realização de um questionário de 10 perguntas, comparativo entre a versão 2 e a versão 3, a membros das equipas da E-goi que utilizam internamente a *API* pública. O questionário continha perguntas relativas às seguintes grandezas: tempo de resposta; facilidade de aprendizagem; facilidade de utilização; quantidade de funcionalidades; usabilidade das funcionalidades.

Este questionário permitiu compreender o quão satisfeitos os clientes se encontram com a nova versão. Os dados das respostas foram utilizados para gerar gráficos de barras com as classificações de satisfação média de cada pergunta. Seguidamente foram calculadas as médias de valores para cada uma das 5 métricas e por fim será calculada a média de satisfação geral para validar se a satisfação é superior na nova *API*.

6.4 *Resultados do questionário*

De forma a avaliar que a satisfação dos utilizadores da versão 3 da *API* é superior à da versão 2, realizou-se o questionário disponibilizado no Anexo B. O questionário possui 10 perguntas, que devem ser respondidas com base na satisfação em cada *API*, para as seguintes métricas:

- **Tempo de resposta:** pergunta 1;
- **Facilidade de aprendizagem:** pergunta 2; pergunta 3; pergunta 4; pergunta 5;
- **Facilidade de utilização:** pergunta 6; pergunta 7; pergunta 8;
- **Quantidade de funcionalidades:** pergunta 9;
- **Utilidade das funcionalidades:** pergunta 10.

No momento da realização dos questionários, a *API* ainda não foi lançada para os clientes, no entanto é utilizada por funcionários da E-goi. O questionário foi distribuído por 11 colaboradores, pertencentes a 2 departamentos que utilizam a *API* internamente. Estes departamentos utilizam a *API* pública, uma vez que a sua função é a integração de serviços externos com a E-goi e por vezes a sua utilização.

Apesar de não haver uma população superior a 30 para realizar o estudo, 80% dos problemas de usabilidade são detetados com 4 ou 5 indivíduos. Sujeitos adicionais possuem uma menor probabilidade de revelar nova informação e os problemas mais graves, provavelmente foram detetados nos primeiros elementos [55].

A Figura 41 revela que a satisfação com a velocidade de resposta é em média superior na nova *API* em relação à anterior. Este resultado está provavelmente relacionado coma versão *do PHP* utilizada no desenvolvimento de cada *API* e com um maior cuidado com a eficiência dos serviços.

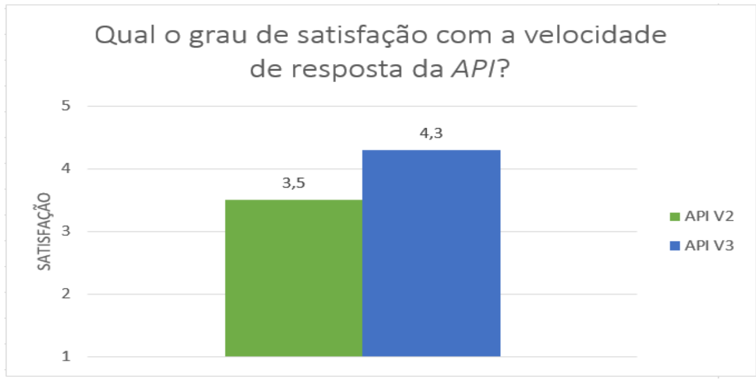


Figura 41 - Gráfico de barras dos resultados da pergunta sobre o tempo de resposta

Na Figura 42 mostram-se os resultados das médias das respostas às perguntas relacionadas com a facilidade de aprendizagem. A população questionada considerar que a nova *API* é mais fácil de aprender inicialmente em relação à *API* anterior.

Estes resultados estão relacionados com a documentação ser superior e com o facto do estilo arquitetural *REST* se encontrar implementado seguindo convenções típicas deste estilo. Por esse motivo as perguntas relacionadas com esses temas também apresentaram resultados semelhantes.

Os resultados revelam ainda que os elementos que responderam preferem a implementação do protocolo *SOAP* na nova *API*. Este resultado poderá está relacionado com o facto de *SOAP* poder ser utilizado com *JSON* na *API* anterior, em vez de *XML* como indicado na especificação.

Calculando a média de respostas sobre satisfação com a facilidade de aprendizagem, determina-se o valor de 3.0 para a *API* anterior e 4.4 para a nova *API.*

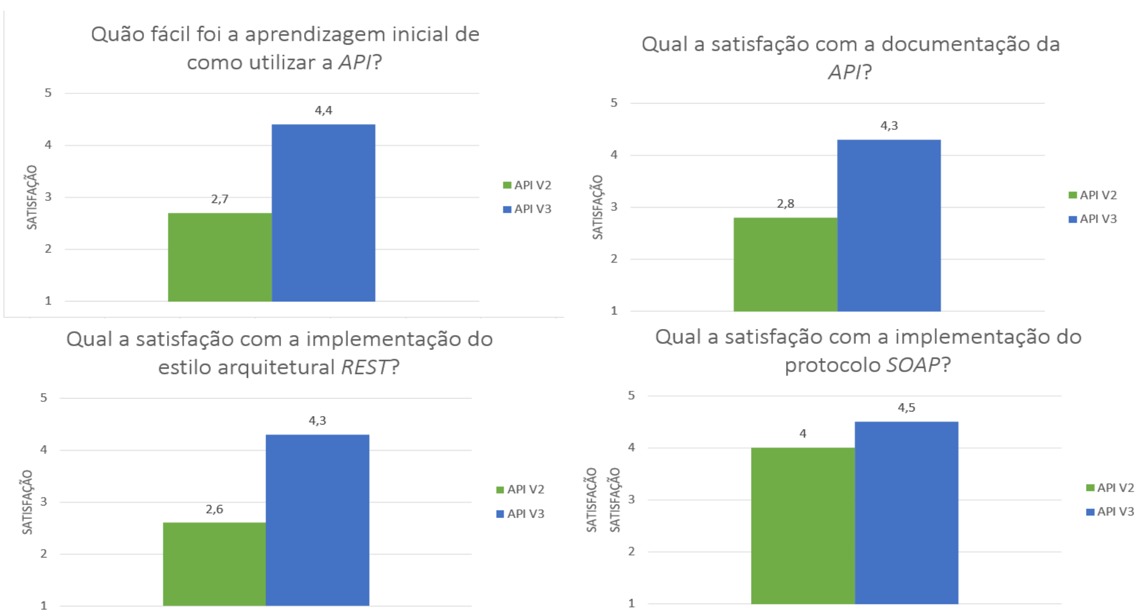


Figura 42 - gráficos de barras das perguntas sobre a facilidade de aprendizagem

A Figura 43 disponibiliza os resultados das questões relacionadas com a satisfação com a facilidade de utilização das *API*. A população estudada considera que a nova *API* é mais fácil de utilizar que a anterior. Este resultado está provavelmente relacionado com uma superior organização das rotas e recursos na nova *API*.

Os utilizadores consideraram também que as mensagens de erro apresentadas na nova *API* e os códigos *HTTP* utilizados são mais úteis e fazem mais sentido na nova *API*. Tal como indicado na

secção 1.3, um dos objetivos deste projeto era ter um cuidado especial com as mensagens de erro e códigos *HTTP* utilizados, para que estes fossem úteis ao utilizador e lhe permitissem resolver o problema. Estes resultados revelam que esse objetivo foi comprido.

Em média a satisfação com a facilidade de utilização é 2.9 na *API* anterior e 4.3 na nova, resultado das médias das respostas 3 perguntas realizadas.

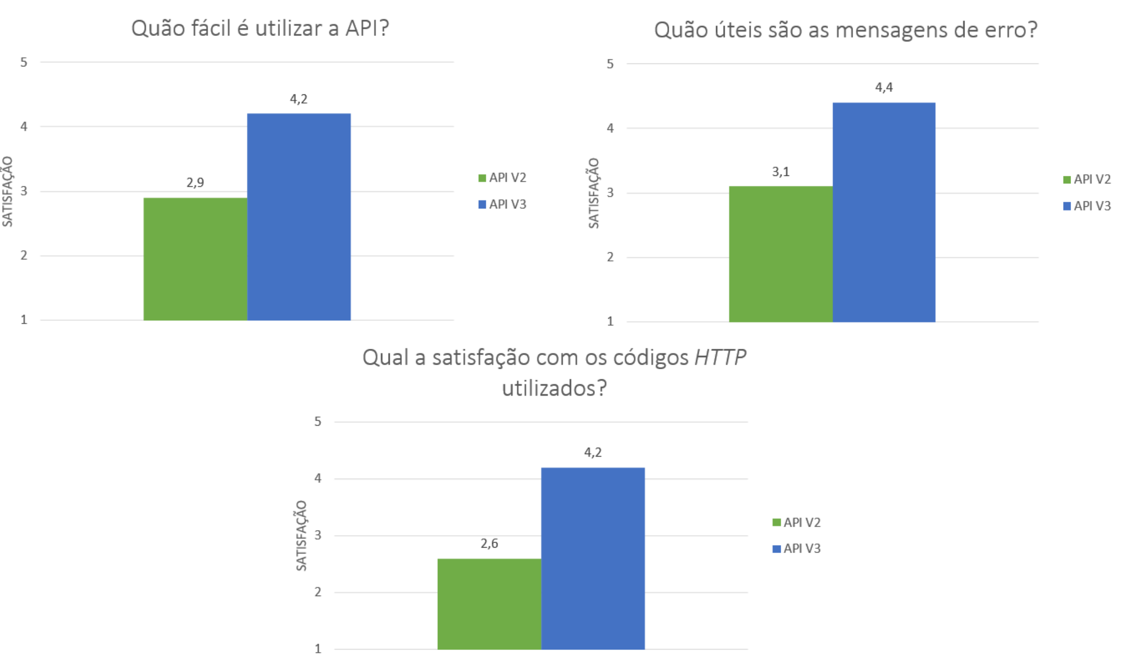


Figura 43 - gráficos de barras das perguntas sobre a facilidade de utilização

A Figura 44 mostra que existe uma ligeira diferença na satisfação na quantidade de funcionalidades disponíveis nas *API*, com preferência para a anterior.

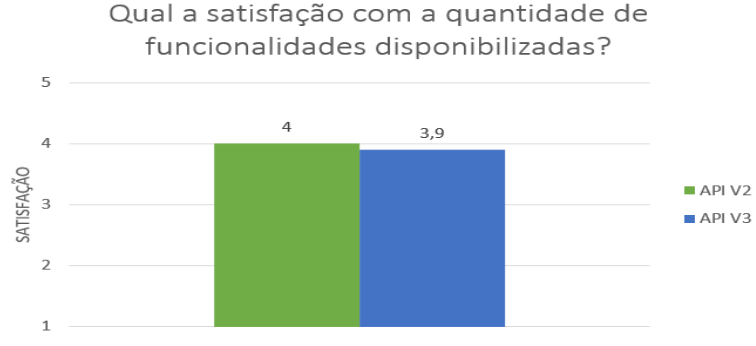


Figura 44 - Gráfico de barras dos resultados da pergunta sobre a quantidade de funcionalidades

Na Figura 45 mostram-se os resultados sobre a opinião média da população estudada, em relação à utilidade das funcionalidades disponibilizadas. Existe uma ligeira preferência pelos serviços da nova *API*. Com a evolução desta *API* e a disponibilização de novos serviços, esta diferença deverá de crescer.

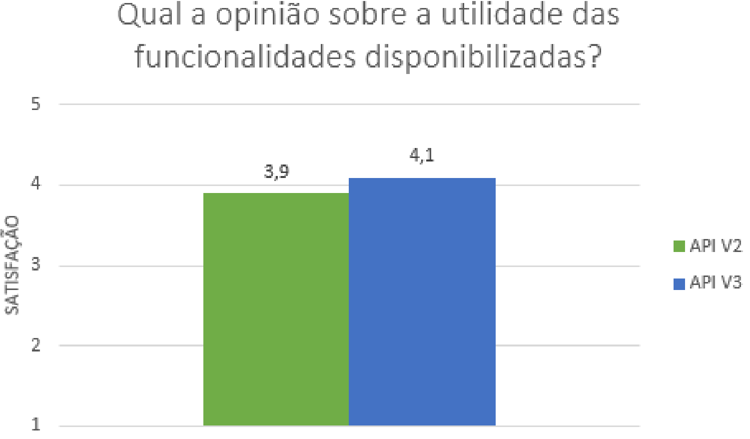


Figura 45 - Gráfico de barras dos resultados - Gráfico de barras dos resultados da pergunta sobre a utilidade das funcionalidades

Com estes resultados, calculando a média geral com base na satisfação média em cada métrica e dando um peso semelhante a cada uma, determina-se que a satisfação média geral é 3.5 na API anterior e 4.2 na nova *API*. Estes resultados indicam que os utilizadores se sentem mais satisfeitos com a nova *API*, o que era um dos objetivos deste trabalho.

7 Conclusão

Neste capítulo apresenta-se os objetivos atingidos, limitações do trabalho desenvolvido e o trabalho futuro.

7.1 Objetivos atingidos

Como indicado na secção 1.3, o objetivo deste trabalho era o desenvolvimento de uma nova *API* pública da E-goi. Esta *API* foi desenvolvida por mais elementos e neste trabalho em especifico, foram enumerados 4 objetivos, sendo que todos foram atingidos. A nova *API* foi desenvolvida por mais elementos e a responsabilidade deste projeto em especifico, era garantir o suporte do estilo arquitetural *REST* e do protocolo *SOAP* e facilitar a utilização da *API,* fazendo uso de vários códigos *HTTP* e garantindo que as mensagens de erro contêm toda a informação necessária. Pretendia-se ainda analisar formas de evoluir esta *API* para que a sua longevidade fosse aumentada, e desenvolver um sistema de versionamento por serviço que funcionasse tanto em *REST* como em *SOAP*.

Para atingir este objetivo realizou-se um estudo onde se analisou o conceito e a história das *API*, o protocolo *HTTP, o* mecanismo de *caching* e como pode ser realizado o controlo de tráfego de utilização da *API*. De seguida foi efetuado um estudo ao estilo arquitetural *REST* para se compreender as suas regras e um estudo semelhante do protocolo *SOAP.* Foi ainda estudada a tecnologia emergente *GraphQL* para se entender como funciona, como poderia ser integrada neste projeto e se fazia sentido nesta fase, tendo sido concluído que por enquanto a E-goi não pretende disponibilizar a tecnologia.

Nesta fase foi também realizado um estudo sobre evolução das *API* onde se estudou as principais alterações que ocorrem nas *API* e porque são um problema para os clientes. Foram identificadas formas de como pode ser realizado o versionamento e estudadas duas técnicas aplicadas por outros grupos. Por fim realizou-se um estudo da *API* anterior da E-goi de forma a documentar os principais problemas e evitar que ocorressem na nova *API*.

Seguidamente realizou-se uma análise de valor onde se analisaram as oportunidades e a proposta de valor de projeto foi descrita. Apresentou-se o modelo Canvas para dar uma visão mais abrangente do projeto e a cadeia de valor de Porter para possibilitar a visualização das atividades da E-goi. Por fim realizou-se um processo de decisão utilizando o modelo *AHP* para escolher se seria melhor fazer um único projeto que suporte *REST* e *SOAP*, um projeto diferente para cada um ou um projeto que apenas suporte *REST*. Conclui-se que a melhor opção era um único projeto que suporte ambos.

Após a análise de valor, iniciou-se a análise e design arquitetural onde se desenvolveu alguns artefactos. Apresentou-se o domínio da solução através do modelo de domínio e de um glossário. Nesta fase também foi efetuada a análise de requisitos e o design arquitetural, onde se desenvolveu o diagrama de componentes, diagrama de pacotes, diagrama de implantação. Considerou-se ainda como ficariam as alternativas arquiteturais caso se tivesse tomado outra decisão no modelo *AHP*.

Em seguida desenhou-se a arquitetura da solução onde foram apresentados diversos diagramas de classes relativos ao sistema de versionamento, como *REST* e *SOAP* se interligam na *API,* tanto utilizando o sistema de versionamento, como diretamente aos serviços partilhados. Foi ainda desenhado o sistema de erros e escolhidos os códigos *HTTP* a utilizar. Foi também desenhado o sistema utilizado para escolher para permitir respostas em vários formatos como *JSON* ou *XML*.

Na implementação foram implementados os serviços seguindo a especificação desenvolvida por outras pessoas e implementada a disponibilização em *REST* e *SOAP*, como desenhado na fase de *design*. Como a empresa não decidiu se pretende disponibilizar a funcionalidade de versionamento por serviço, esse sistema apenas foi implementado para o recurso *Lists*, para servir como protótipo. No futuro se for pretendido disponibilizar a funcionalidade, os restantes recursos podem ser facilmente integrados, da mesma forma que o recurso *Lists.* Neste documento foram apresentados diversos diagramas de sequência sobre a implementação e descritas algumas decisões tomadas.

Após a implementação ter sido concluída, avaliou-se a solução cumpria os objetivos propostos, tendo em conta as grandezas e hipóteses a verificar. Realizaram-se testes de aceitação aos serviços, utilizando testes gerados automaticamente por outros elementos que verificam se os serviços estão coerentes com a especificação, e foram realizados testes personalizado ao sistema de versionamento no módulo *Lists*, tendo obtido 100% de sucesso e cobertura

Utilizando a ferramenta *JMeter* foram medidos os tempos de resposta da *API* pública e da *API* privada, 30 vezes por serviço e em todos os serviços disponibilizados. No caso da *API* privada os tempos medidos foram dos serviços utilizados pela *API* pública. Através de um teste estatístico verificou-se que a utilização da *API* pública representa em média um aumento médio do tempo de resposta, em relação à utilização direta da privada, é inferior a 20%, o que era um dos objetivos de avaliação.

Foi também realizado um questionário de satisfação internamente a 11 colaboradores da E-goi que utilizam internamente a *API*, de forma a medir a satisfação com a nova *API* pública e com a anterior. O questionário avaliou a satisfação com o tempo de resposta, com a facilidade de aprendizagem e utilização, com a quantidade e a utilidade das funcionalidades disponibilizas. Verificou-se que a satisfação em geral é maior com a nova *API.*

De momento a nova *API* ainda se encontra em desenvolvimento e só está a ser utilizada internamente. Estão ainda a ser incluídas novas funcionalidades na *API* pelo que este resultado provavelmente vai melhorar no futuro. Com os resultados obtidos na avaliação e com o trabalho realizado conclui-se que os objetivos propostos na secção 1.3 foram todos atingidos e que os problemas foram resolvidos.

7.2 Limitações e trabalho futuro

No momento da escrita deste documento a nova *API* ainda não foi lançada, dado que ainda serão adicionados novos serviços e os que foram disponibilizados estão sujeitos a alterações. Será realizado trabalho no sentido de apresentar novas funcionalidades aos clientes da E-goi assim como de melhorar a sua satisfação. Os serviços *REST* serão também disponibilizados em *XML,* sendo que o sistema atual já está pensado e preparado para tal.

O ficheiro *WSDL* poderá no futuro ser gerado através da especificação da *Open API* em vez de utilizar a funcionalidade *AutoDiscovery* da *Zend Framework*. Tal é possível porque tanto *REST* como *SOAP* disponibilizam exatamente os mesmos serviços que já se encontram especificados

em *Open API*. Caso o sistema de versionamento seja utilizado, esta é a melhor forma de gerar o *WSDL*, devido a depender da especificação em vez da estrutura do método correspondente a um serviço *SOAP* e outra forma de garantir que tanto *SOAP* como *REST* disponibilizam os mesmos serviços.

O sistema de versionamento foi desenhado para suportar todos os serviços tanto em *SOAP* como em *REST*, no entanto uma vez que a E-goi ainda não decidiu se quer disponibilizar o sistema, só foi implementado no recurso *Lists* como protótipo. Se a E-goi decidir que quer disponibilizar esta funcionalidade, os restantes serviços terão de ser ligados da mesma forma que os serviços do módulo *List* foram. Este processo é relativamente rápido sendo que só é necessário colocar os serviços numa *Operation* própria, criar o *VersionManager* do recurso e especificar as operações como versão 1.0.0. A partir desse momento os serviços estão associados a uma versão, o que permite aumentar a satisfação e longevidade da *API* dado que, se um serviço for alterado de uma forma que possa quebrar as aplicações dos clientes, poderá ser mantida a versão anterior.

Por fim se for no futuro a empresa desejar disponibilizar uma *API* pública de *GraphQL*, tal pode ser feita como referido nas alternativas arquiteturais. Poderá ser criada uma *API* que depende da desenvolvida neste projeto, da *API* privada ou de ambas em simultâneo, o que permite que possa ter mais funcionalidades. Poderá também ser disponibilizado como uma componente deste projeto, tal como os serviços *REST* e *SOAP* foram implementados.

Referências

[1] A. Abelló, C. Ayala, C. Farré, C. Gómez, M. Oriol e O. Romero, “A Data-Driven Approach to Improve the Process of Data-Intensive API Creation and Evolution,” em *29th International Conference on Advanced Information*, Essen, Germany, 2017.

[2] W. Santos, “ProgrammableWeb - Witch API types and architectural styles are most used,” ProgrammableWeb, 26 11 2017. [Online]. Available: https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26].. [Acedido em 18 12 2017].

[3] M. Fowler, “MartinFlower.com - Richardson Maturity Model,” MartinFlower, 18 03 2010. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Acedido em 30 01 2018].

[4] M. Massé, REST API - Design Rulebook, Sebastapol: O’Reilly Media, Inc, 2012.

[5] W3C, “W3C - Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content,” W3C, 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231. [Acedido em 06 11 2017].

[6] W3C, “W3C - SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),” W3C, 27 04 2007. [Online]. Available: https://www.w3.org/TR/soap12/#soapenv. [Acedido em 19 11 2017].

[7] P. Koen, G. Ajamian, R. Burkart, A. Clamen, J. Davidson, R. D’Amore, C. Elkins, K. Herald, M. Incorvia, A. Johnson, R. Karol, R. Seibert, A. Slavejkov e K. Wagner, “Providing clarity and a common language to the “fuzzy front end”,” *Research Technology Management,* pp. 46-55, 2001.

[8] A. Osterwalder e Y. Pigneur, Business Model Generation: Inovação em Modelo de Negócios, 1ª ed., Rio de Janeiro: Alta Books, 2011.

[9] D. Barnes, Understanding Business: Processes, New York: Routledge, 2001.

[10] T. L. Saaty, “Decision making with the analytic hierarchy process,” *International journal of services sciencies,* vol. 1, pp. 83-98, 2008.

[11] B. De, API Management, Springer, 2017.

[12] M. Biehl, RESTful API Design -Best Practices in API Design with REST, API-University Press, 2016.

[13] K. Lane, "API Evangelist - The History of APIs," API Evangelist, [Online]. Available: https://history.apievangelist.com/#The History of APIs. [Acedido em 11 11 2017].

[14] W3C, "W3 - Web API Design Coockbook," W3C, 02 10 2012. [Online]. Available: https://www.w3.org/TR/api-design/. [Acedido em 7 11 2017].

[15] K. Sandoval, "Nordic APIs - Stemming the Flood – How to Rate Limit an API," 16 2 2016. [Online]. Available: https://nordicapis.com/stemming-the-flood-how-to-rate-limit-an-api/. [Acedido em 5 12 2017].

[16] A. R. Fielding, E. M. Nottingham, Akamai, E. J. Reschke e greenbytes, 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7234#section-5.2. [Acedido em 28 05 2018].

[17] S. Sohan, C. Anslow e F. Maurer, "A case study of web API evolution," em *Services (SERVICES), 2015 IEEE World Congress on*, 2015.

[18] T. Espinha, A. Zaidman e H.-G. Gross, "Web API growing pains: Stories from client developers and their code," em *Proc. of Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, 2014.

[19] S. Wang, I. Keivanloo e Y. Zou, "How do developers react to restful api evolution?," em *International Conference on Service-Oriented Computing*, 2014.

[20] D. Romano, "Analyzing the Change-Proneness of APIs and web APIs," 2015.

[21] P. Leitner, A. Michlmayr, F. Rosenberg e S. Dustdar, "End-to-End Versioning Support for Web Services," *Services Computing, 2008. SCC'08. IEEE International Conference on,* vol. 1, pp. 59--66, 2008.

[22] S. Raemaekers, A. v. Deursen e J. Visser, "Semantic versioning versus breaking changes: A study of the maven repository," em *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference*, 2014.

[23] SemVer, "Semantic Versioning 2.0.0," SemVer, [Online]. Available: Semantic Versioning 2.0.0. [Acedido em 15 04 2018].

[24] P. Kaminski, H. Müller e M. Litoiu, "A design for adaptive web service evolution," em *Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, 2006.

[25] W3C, "Hypertext Transfer Protocol (HTTP/1.1): Section 3 Message Format," 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7230#section-3. [Acedido em 04 02 2018].

[26] W3C, "Hypertext Transfer Protocol (HTTP/1.1): Section 6 Response Status Codes," HTTP Status Codes, 06 2014. [Online]. Available: https://tools.ietf.org/html/rfc7231#section-6. [Acedido em 01 02 2018].

[27] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Doctoral Dissertation: University of California, Irvine, 2000.

[28] M. W. Khan e E. Abbasi, "Differentiating parameters for Selecting Simple Object Access Protocol (SOAP) vs. Representational State Transfer (REST) based architecture," *Journal of Advances in Computer Network,* pp. 63-6, 11 2015.

[29] W3C, "W3C - SOAP Version 1.2 Part 0: Primer (Second Edition)," W3C, 27 04 2007. [Online]. Available: https://www.w3.org/TR/soap12-part0/. [Acedido em 29 01 2018].

[30] W3C, "W3C - Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language," W3C, 26 6 2007. [Online]. Available: https://www.w3.org/TR/2007/REC-wsdl20-20070626/#meaning. [Acedido em 17 12 2017].

[31] w3schools, "W3 Schools - XML WSDL," w3schools, [Online]. Available: https://www.w3schools.com/xml/xml_wsdl.asp. [Acedido em 17 12 2017].

[32] Facebook, "GraphQL - Working Draft," Facebook, 10 2016. [Online]. Available: http://facebook.github.io/graphql/. [Acedido em 07 02 2018].

[33] J. Helfer, "Apollodata Dev-Blog: GraphQL explained," Apollo, 23 05 2016. [Online]. Available: https://dev-blog.apollodata.com/graphql-explained-5844742f195e. [Acedido em 08 02 2018].

[34] GraphQL, "GraphQL - Queries and Mutations," GraphQL, [Online]. Available: http://graphql.org/learn/queries/. [Acedido em 07 02 2018].

[35] howtographql, "howtographql - Big Picture (Architecture)," howtographql, [Online]. Available: https://www.howtographql.com/basics/3-big-picture/. [Acedido em 22 02 2018].

[36] GraphQL, "GraphQL - Caching," GraphQL, [Online]. Available: http://graphql.org/learn/caching/. [Acedido em 22 02 2018].

[37] W3c, "Simple Object Acess Protocolo (SOAP) - The SOAPAction HTTP Header Field," W3C, 08 05 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/#_Toc478383528. [Acedido em 20 05 2018].

[38] GraphQL, "GraphQL - Best Practices - Versioning," GraphQL, [Online]. Available: http://graphql.org/learn/best-practices/. [Acedido em 22 02 2018].

[39] E-goi, “E-goi API Docs,” E-goi, [Online]. Available: https://api-docs.e-goi.com. [Acedido em 11 11 2017].

[40] P. Koen, G. Ajamian, S. Boyce, A. Clamen, E. Fisher, S. Fountoulakis, A. Johnson, P. Puri e R. Seibert, Fuzzy front end: effective methods, tools, and techniques, New York: Wiley, 2002.

[41] A. Hassan, “The Value Proposition Concept in Marketing: How Customer Perceive The Value Delivery by Firms- A Study of Custommers Perspectives on Supermarkets in Southampton in the United Kingdom,” *International journal of marketing studies,* p. 68, 1 6 2012.

[42] P. Fifield, Marketing Strategy Masterclass, Routledge, 2008.

[43] S. Nicola, E. Ferreira e J. Ferreira, “A novel framework formodeling value for the customer, an essay on negotiation,” *International Journal of Information Technology & Decision Making,* 12 2012.

[44] R. Woodruff, “Customer value The next source for competitive advantage,” *Journal of academy of marketing science,* p. 139, 1997.

[45] A. Lindgreen e F. Wynstra, “Value in business markets: What do we know? Where are we going?,” *Industrial Marketing Management,* pp. 732-748, 2005.

[46] A. Osterwalder, Y. Pigneur, G. Bernarda, A. Smith e T. Papadakos, Value Proposition Design: Como construir propostas de valor inovadoras, 1ª ed., São Paulo: HSM Editora, 2014.

[47] T. Woodall, “Conceptualising 'Value for the Customer': An Attributional, Structural and Disposional Analysis,” *Academy of marketing science review,* 1 2003.

[48] C. S. Marins, D. d. O. Souza e M. d. S. Barros, “O uso do método de Análise Hierárquica (AHP) na tomada de decisão,” *XLI SBPO,* 2009.

[49] C. Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, 3ª ed., New Jersey: Prentice Hall, 2004.

[50] D. M. Stephen White, BPMN Modeling And Reference guide:: understanding and using BPMN, Future Strategies Inc, 2008.

[51] PHP, “PHP Documentation - The SoapServer class,” [Online]. Available: http://php.net/manual/en/class.soapserver.php. [Acedido em 06 05 2018].

[52] Zend Framework, “Zend Framework - About,” [Online]. Available: https://framework.zend.com/about. [Acedido em 22 05 2018].

[53] Zend Framework, “Zend Framework Documentation - Zend\Soap\Server,” 2018 05 23. [Online]. Available: https://zendframework.github.io/zend-soap/server/.

[54] G. D. Ruxton, “The unequal variance t-test is an underused alternative to Student's t-test and the Mann--Whitney U test,” *Behavioral Ecology,* vol. 17, pp. 688--690, 2006.

[55] R. A. Virzi, “Refining the test phase of usability evaluation: How many subjects is enough?,” *Human factors,* vol. 34, pp. 457--468, 1992.

Anexo A - Questionário do modelo AHP

As perguntas seguintes são comparações, entre critérios ou soluções relativamente a um critério, que devem ser respondidos colocando um (x) no número que representa a melhor opção. Para tal, deve ser utilizada a seguinte escala:

- 1: ambos os elementos têm igual importância;
- 3: um determinado elemento tem uma moderada importância em relação ao outro;
- 5: um determinado elemento tem uma forte importância em relação ao outro;
- 7: um determinado elemento tem uma importância muito forte em relação ao outro;
- 9: um determinado elemento tem uma extrema importância em relação ao outro;
- 2,4,6,8: Valores intermédios.

Os critérios em análise são os seguintes:

- Suporta os 2 estilos arquiteturais: indica se as soluções respeitam as regras dos 2 estilos arquiteturais *REST* e *SOAP*;
- Escalabilidade: indica o quão escalável é a solução;
- Reaproveitamento de código: indica quanto código pode ser reaproveitado;

As soluções em análise são as seguintes:

- Fazer 2 projetos: fazer 2 projetos, um para cada API de cada estilo arquitetural;
- Fazer 1 projeto: fazer 1 projeto que suporte os 2 estilos arquiteturais;
- Fazer só a API REST: fazer apenas um projeto que suporte apenas *REST*, uma vez que este estilo arquitetural é o mais popular.

A. Comparação entre critérios (Preencher com um único x na melhor opção) :

1. Suporta os 2 estilos arquiteturais ou escalabilidade

Suporta os 3 estilos arquiteturais																Escalabilidade
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

2. Suporta os 2 estilos arquiteturais ou reaproveitamento de código

Suporta os 3 estilos arquiteturais																Reaproveitamento de código
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

3. Escalabilidade ou reaproveitamento de código

Escalabilidade																Reaproveitamento de código
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

B. Comparação de soluções para o critério "Suporta os 2 estilos arquiteturais" (Preencher com um único x na melhor opção):

1. Fazer 2 projetos ou fazer 1 projeto

Fazer 2 projetos																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

2. Fazer 2 projetos ou fazer só a *API REST*

Fazer 2 projetos																Fazer só a *API REST*
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

3. Fazer só a *API REST* ou fazer 1 projeto

Fazer só a *API REST*																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

C. Comparação de soluções para o critério "Escalabilidade" (Preencher com um único x na melhor opção):

1. Fazer 3 projetos ou fazer 1 projeto

Fazer 3 projetos																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

2. Fazer 2 projetos ou fazer só a *API REST*

Fazer 2 projetos																Fazer só a *API REST*
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

3. Fazer só a *API REST* ou fazer 1 projeto

Fazer só a *API REST*																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

D. Comparação de soluções para o critério "Reaproveitamento de código" (Preencher com um único x na melhor opção):

1. Fazer 3 projetos ou fazer 1 projeto

Fazer 2 projetos																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

2. Fazer 3 projetos ou fazer só a *API REST*

Fazer 2 projetos																Fazer só a *API REST*
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

3. Fazer só a *API REST* ou fazer 1 projeto

Fazer só a *API REST*																Fazer 1 projeto
9	8	7	6	5	4	3	2	1	2	3	4	5	6	7	8	9

Anexo B - Questionário de satisfação

As perguntas seguintes pretendem avaliar a satisfação dos utilizadores da nova *API* pública do E-goi, comparando-a com a anterior sempre que possível. Em cada questão, para cada uma das *API*, deve ser selecionado (x) o número que representa a satisfação do utilizador. Para tal, deve ser utilizada a seguinte escala:

- **1:** muito insatisfeito;
- **2:** insatisfeito;
- **3:** neutro;
- **4:** satisfeito;
- **5:** muito satisfeito;

1. Qual o grau de satisfação com a velocidade de resposta da *API*?

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

2. Quão fácil foi a aprendizagem inicial de como utilizar a *API*?

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

3. Qual a satisfação com a documentação da *API*?

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

4. **Qual a satisfação com a implementação do estilo arquitetural *REST*?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

5. **Qual a satisfação com a implementação do protocolo *SOAP*?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

6. **Quão fácil é utilizar a *API*?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

7. **Quão úteis são as mensagens de erro?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

8. **Qual a satisfação com os códigos *HTTP* utilizados?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

9. **Qual a satisfação com a quantidade de funcionalidades disponibilizadas?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

10. **Qual a opinião sobre a utilidade das funcionalidades disponibilizadas?**

API	**1**	**2**	**3**	**4**	**5**
Anterior (versão 2)					
Nova (versão 3)					

Anexo C - Suporte de *GraphQL*

A Figura 46 apresenta uma solução de como poderia ser disponibilizado futuramente o estilo arquitetural GraphQL aproveitando o projeto desenvolvido neste trabalho. A API de GraphQL poderia ser um projeto separado que comunicava com a API pública e com as API internas do E-goi de forma a poder ter acesso aos recursos necessários para disponibilizar uma rota GraphQL.

A possibilidade de comunicar com as *API* internas possibilita que a API *GraphQL* disponibilize serviços não existentes na *API* pública. Tal como foi explicado na secção 2.3.4 do capítulo Estado da Arte, não é desejável para a E-goi disponibilizar no contexto deste projeto essas funcionalidades, no entanto no futuro com o aumento do uso desta tecnologia, esta solução poderá ser aplicada, caso a quantidade de utilizadores de *GraphQL* o justifique .

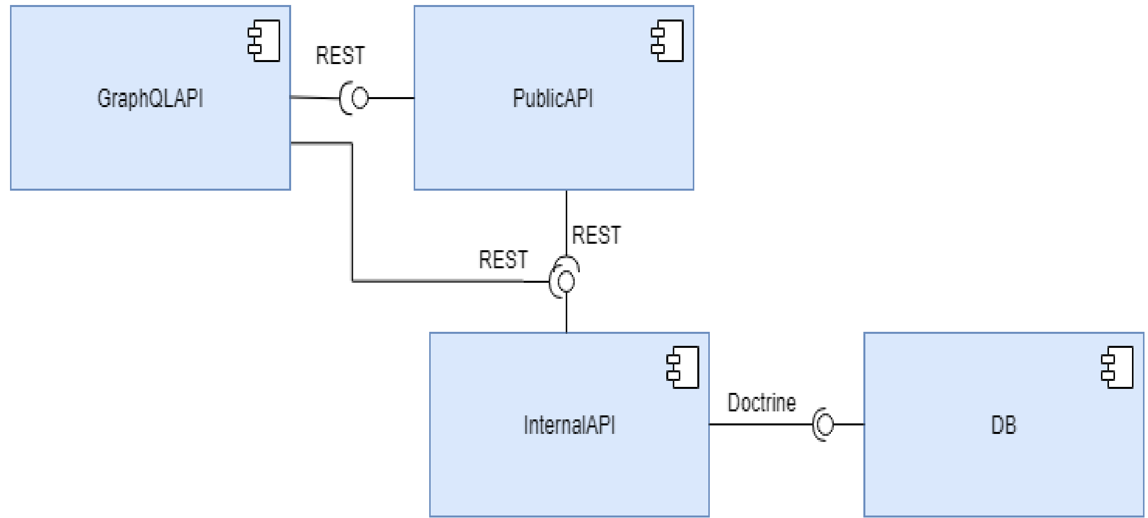


Figura 46 - Diagrama de componentes alternativo 3

Outra solução seria a alternativa arquitetural apresentada na Figura 47. Em vez de ser criado um novo projeto para disponibilizar para a *API* de *GraphQL*, esta poderia ser uma componente do projeto da *API* pública. Esta solução permite um maior reaproveitamento de código, no entanto não é tão escalável como a anterior.

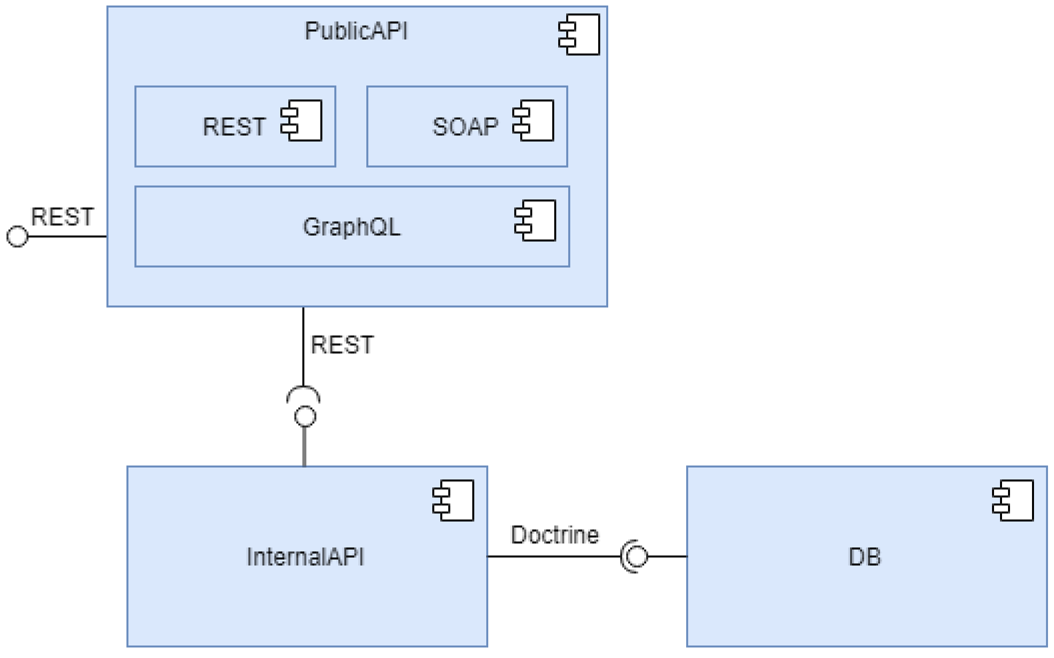


Figura 47 - Diagrama de componentes alternativo 4