INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Suporte à Evolução de API

RICARDO SOARES DOS SANTOS
Julho de 2018

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Suporte à Evolução de API

Ricardo Soares dos Santos

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

Orientadora: Isabel de Fátima Silva Azevedo

Supervisor: Ivo Pereira

Júri:

Presidente:

[Nome do Presidente, Categoria, Escola]

Vogais:

[Nome do Vogal1, Categoria, Escola]

[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, julho 2018

Resumo

A evolução de uma *Application Programming Interface (API)* é o conjunto das mudanças que ocorrem nessa *API* ao longo do tempo. Portanto, sempre que ocorre uma mudança evolutiva, existem componentes relacionadas com a *API* que sofrem alterações, tais como a documentação e as bibliotecas de código cliente.

Assim sendo, ao desenhar uma *API*, especialmente seguindo uma abordagem *design-first*, é relevante definir um contrato formal, que deve conter toda a informação necessária sobre o comportamento da *API*, sendo conhecido como especificação de *API*. Desta forma, todas as componentes relacionadas podem depender do mesmo contrato, o que implica que, caso ocorra uma evolução na *API*, o contrato é alterado e todas essas componentes podem ser atualizadas dinamicamente com a nova informação. Portanto, este contrato pode ser utilizado para gerar componentes como documentação, código cliente e testes de *software*, reduzindo custos de desenvolvimento e manutenção.

Neste documento, são apresentadas soluções para a *API* pública do E-goi. O E-goi é uma plataforma de *marketing* digital multicanal e para alcançar uma solução que resolva os problemas encontrados nesta *API*, maioritariamente relacionados com evolução e altos custos de manutenção do *software*, é necessário estudar o estilo arquitetural *Representational State Transfer (REST)* e linguagens para especificação de *API REST.* Também devem ser estudadas outras *API* públicas na área de *marketing*, diferentes tipos de versionamento e registos de utilização da *API* que sejam relevantes, uma vez que é através destes registos que se podem encontrar potenciais falhas e melhorias que podem resultar em mudanças evolutivas na *API*.

A arquitetura das soluções desenvolvidas e outras alternativas arquiteturais para geração de documentação, código cliente e testes estão descritas e apresentadas utilizando *Unified Modelling Language (UML)*. Seguidamente é apresentado o *design* a um nível mais baixo, conjuntamente com alguns aspetos de implementação importantes. Por fim é feita a experimentação e avaliação para verificar se esta cumpre todos os objetivos e resolve os problemas identificados.

Palavras-chave: Evolução de *API*, Especificação de *API*, Documentação, Testes de *Software*, Geração de Código, *Design-first*, *REST*, Manutenção de *Software*, *Marketing* Digital

Abstract

Evolution can be understood as the changes that occur in an Application Programming Interface (API) over time. Therefore, every time an evolutive change occurs, there are components related to the API that also change, such as documentation and client code libraries.

Thereby, when designing an API, especially when following a design-first approach, it is relevant to define a formal contract, known as API specification, which must contain every information regarding the API behavior. Being so, every related component can depend on the same contract, meaning that when an evolution on the API occurs, the contract is altered, and every component may be updated dynamically with the new information. Thus, this contract may be used to generate components like documentation, client code and software tests, reducing development and maintenance costs.

In this document, it is presented a set of solutions for E-goi's public API. Egoi is a multichannel digital marketing platform and, to achieve a solution to solve these API problems, majorly related to evolution and high software maintenance costs, it is necessary to study Representational State Transfer (REST) architectural style and languages for specification of REST API. Also, it must be studied other public API in marketing area, different sorts of versioning and important records of API usage, once these records are used to find potential flaws and improvements that may result in evolutive changes in the API.

The architecture of the developed solutions and other architectural alternatives to the generation of documentation, client code and tests are descripted and presented using Unified Modelling Language (UML). Then, it is presented the design in a lower level, together with some important implementation aspects. Finally, it is done the experimentation and evaluation to verify if the solution matches all the required objectives and solves the identified problems.

Keywords: API evolution, API Specification, Documentation, Software Tests, Code Generation, Design-first, REST, Software Maintenance, Digital Marketing

Índice

Lista de Figuras

Lista de Tabelas

Lista de Excertos de Código

Lista de Acrónimos e Siglas

AES	*Advanced Encryption Standard*
AHP	*Analytic Hierarchy Process*
API	*Application Programming Interface*
CRUD	*Create, Read, Update, Delete*
FFE	*Fuzzy Front End*
FURPS+	*Functionality, Usability, Reliability, Performance and Supportability Plus*
HATEOAS	*Hypermedia As The Engine Of Application State*
HTML	*HyperText Markup Language*
HTTP	*HyperText Transfer Protocol*
IDE	*Integrated Development Environment*
IP	*Internet Protocol*
JSON	*JavaScript Object Notation*
MSON	*Markdown Syntax for Object Notation*
NPD	*New Product Development*
OAI	*Open API Initiative*
OAS	*OpenAPI Specification*
PHP	*PHP: Hypertext Preprocessor*
RAML	*RESTful API Modelling Language*
REST	*Representational State Transfer*
RPC	*Remote Procedure Call*
SDK	*Software Development Kit*
SHA2	*Secure Hash Algorithm 2*
SOAP	*Simple Object Access Protocol*
UML	*Unified Modelling Language*
URI	*Uniform Resource Identifier*
URL	*Uniform Resource Locator*

XML *eXtensible Markup Language*

XML-RPC *eXtensible Markup Language – Remote Procedure Call*

YAML *YAML Ain't Markup Language*

1 Introdução

Neste capítulo são identificados os problemas que levaram ao desenvolvimento do trabalho reportado neste documento, bem como o seu contexto, os objetivos a serem cumpridos para solucionar esses problemas e a abordagem e processo de desenvolvimento adotados para obtenção da solução final. Por fim é apresentada a estrutura deste documento.

1.1 Contexto

O E-goi[1], uma plataforma de automação de *marketing* multicanal com mais de trezentos mil clientes em mais de cinquenta países (apesar do foco estar no mercado português, brasileiro, espanhol e colombiano), possui diversas ferramentas para facilitar a gestão do *marketing* digital de empresas. Sendo uma plataforma de *marketing* digital, este *software* permite a criação de campanhas de *marketing* para diferentes canais e enviá-las para uma lista de contactos (ou segmento dessa lista).

Atualmente existe uma grande necessidade de automatizar processos e diversas empresas disponibilizam parte dos seus serviços internos aos seus clientes através de *Application Programming Interfaces (API)* [1]. Esta automação de processos é essencial para muitos negócios, como *marketing* digital. Assim, a empresa E-goi disponibiliza aos seus clientes uma *API* pública, para que este possam integrar as suas aplicações com o E-goi, de forma a automatizar processos de *marketing*.

A evolução de uma *web API* ocorre quando um serviço é alterado, adaptado ou ajustado [2]. De forma a manter a compatibilidade com o que já existe, é frequente deixar disponível, não só a

[1] https://www.e-goi.com/

última versão do serviço, mas também versões anteriores [2]. De forma a disponibilizar diferentes versões do mesmo serviço da *API* é possível utilizar *proxies* de *API.* Algumas *API* disponibilizam ainda *Software Development Kits (SDK)*, que são bibliotecas que abstraem os utilizadores da lógica relacionada com pedidos *web*, tornando assim a comunicação entre a *API* e o cliente mais transparente.

As *API* públicas são, muitas vezes, caixas negras para os seus utilizadores, visto que o seu código não está publicamente acessível [3]. Desenvolvedores que trabalham nas suas aplicações rapidamente integram *software* para desenvolver sistemas distribuídos. No entanto aprender a utilizar um *API* não é uma tarefa simples [4].

Assim sendo, uma das características mais importantes em relação à evolução de uma *API* pública é a sua documentação, que existe para tornar mais fácil a sua utilização por parte de um utilizador, devendo ser detalhada, objetiva e manter-se atualizada, ou podem ser criadas ainda mais dificuldades ao utilizador com um impacto negativo [5]. Portanto, para que uma *API* seja fácil de utilizar*,* é essencial que a sua documentação esteja sempre atualizada e forneça ao utilizador exemplos da sua utilização [5, 6, 7]. De facto, a documentação pode ser bastante importante para perceber como utilizar bibliotecas e ferramentas desenvolvidas por outras equipas, cujas funcionalidades são facilmente acessíveis através de *API* [8]. Contudo, a documentação de *API* é frequentemente criada de forma manual e requer que esta seja atualizada à medida que a respetiva *API* evolui, para que não se torne obsoleta [1], implicando elevados custos de manutenção.

Como para qualquer outra aplicação, testes de *software* são uma componente essencial de qualquer *API* para aumentar a qualidade do *software* e reduzir os seus custos de manutenção [9]. Contudo, por vezes, desenvolver testes pode ser um processo ainda mais árduo que produzir a aplicação [9].

Assim sendo, manter uma *API*, incluindo documentação, *SDK* e testes, pode tornar-se bastante difícil e custoso. No entanto, estes artefactos podem ser gerados dinamicamente, reduzindo assim os custos de manutenção. Tal pode ser alcançado com base na especificação de *API*. A especificação de uma *API* estabelece um contrato e descreve a *API* através de um documento facilmente legível por humanos e máquinas [10], permitindo a geração de documentação, código e testes de *software* dinamicamente a partir desse documento.

1.2 Problema

Com o crescimento da plataforma E-goi, surgiu a necessidade de criar uma nova *API* pública. Esta *API* atua como um *proxy* entre as aplicações dos clientes e as *API* internas do E-goi, como é visível na Figura 1.

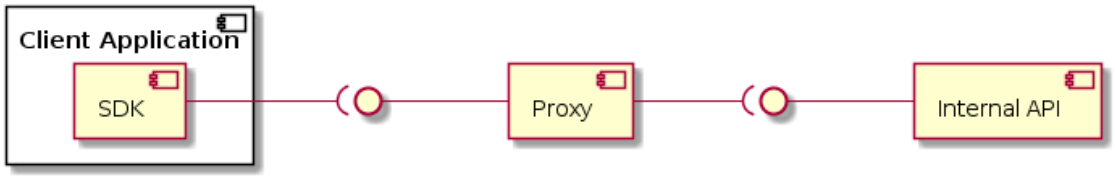

Figura 1 - Vista lógica da nova API pública do E-goi

Contudo, a *API* anterior possuía diversos problemas de escalabilidade e, portanto, surgiu a necessidade de desenvolver uma nova *API*, que se encontra a ser desenvolvida paralelamente a este trabalho.

Houve ainda outros problemas, na sua maioria relacionados com custos de manutenção após ocorrer uma mudança evolutiva na *API* e são estes problemas que se desejam ver resolvidos com esta dissertação. Um dos principais problemas enfrentados foi o facto de a *API* possuir uma documentação insuficiente e que se tornou rapidamente obsoleta. A documentação e os *SDK* foram criados manualmente, pelo que é necessário que estes sejam alterados sempre que a *API* evolui. Além disso, não existem testes de *software* nem qualquer tipo de registos da utilização da *API* para posterior monitorização da mesma, o que dificulta a identificação de falhas e possíveis melhorias.

1.3 Objetivos

A nova *API* pública suporta o protocolo *Simple Object Access Protocol (SOAP)* e o estilo arquitetural *Representational State Transfer (REST).*

Com esta dissertação, face aos problemas identificados na secção 1.2, pretende-se resolver, para a componente *REST*:

- Especificação da *API*;
- Geração dinâmica de documentação, com base na especificação da *API*, inclusive exemplos de utilização;

- Geração dinâmica de um *SDK* na linguagem *PHP Hypertext Preprocessor (PHP)*, com base na especificação da *API*;
- Geração dinâmica de testes de *software*, com base na especificação da *API*;

Também será necessário desenvolver uma solução para gerar registos da utilização da *API* pública partilhada pela componente *REST* e pela componente *SOAP*.

1.4 Abordagem e Processo de Desenvolvimento

A especificação da *API*, bem como a geração dinâmica de documentação, SDK e testes, implica a efetuar um estudo de linguagens para especificação de *API.* Como a *API* a ser descrita segue o estilo arquitetural *REST*, é necessário efetuar um estudo deste mesmo estilo arquitetural. Para que a documentação tenha detalhe e qualidade suficiente para facilitar a utilização da *API* por parte dos utilizadores, é necessário analisar a *API* anterior do E-goi e as *API* dos seus principais concorrentes, de forma a encontrar características fundamentais a incluir na documentação.

Deve ser ainda estudado alguns conceitos de evolução e versionamento e o sistema de versionamento existente na nova *API*. Também deve ser analisada a relevância da existência de registos de utilização da *API*.

Para entender melhor o projeto e a área de negócio em que este se encontra inserido será efetuada uma análise de valor. Será feita também uma análise técnica, dividida em análise de requisitos e análise de negócio.

Depois de analisar o problema é necessária definir a arquitetura do sistema. Seguidamente é elaborado o *design* de cada uma das componentes e a respetiva implementação.

Por fim é necessária testar e avaliar o *software* produzido, assegurando assim a sua qualidade.

1.5 Estrutura do Documento

O presente documento está estruturado em diversos capítulos:

- **Introdução**: Neste capítulo introdutório é apresentado o contexto desta dissertação, os problemas que levaram à sua criação, os objetivos propostos e a abordagem e processo de desenvolvimento adotado;
- **Estado da Arte**: Neste capítulo são estudados temas relacionados com *API*, nomeadamente o estilo arquitetural *REST*, evolução e versionamento, linguagens

para especificação de *API REST* e registos de utilização da *API.* É também feita uma análise e comparação de diversas *API* na área de *marketing* digital da *API* anterior do E-goi.

- **Análise de Valor:** Neste capítulo é feita uma análise de valor, constituída pelo processo de negócio e inovação, valor para o cliente, as propostas de valor para cada um dos segmentos, o modelo de negócio *Canvas*, a cadeia de valor de *Porter* e o método *Analytic Hierarchy Process (AHP)*;
- **Análise de Negócio e Requisitos:** Neste capítulo é feita a análise de requisitos (funcionais e não funcionais) e também a análise de negócio;
- ***Design* Arquitetural:** Neste capítulo é apresentado o *design* arquitetural das soluções escolhidas e também o *design* de alternativas de solução encontradas;
- ***Design* e implementação:** Neste capítulo é dado e explicado um exemplo de especificação *OpenAPI* para um serviço da *API* e é apresentado o *design* de soluções que utilizam essa especificação. É também mostrado o *design* referente à aplicação para criação de registos de utilização da *API*;
- **Testes e Validação da Solução:** Neste capítulo são identificadas as grandezas, hipóteses, metodologias e testes estatísticos que podem ser utilizados para avaliar e testar as soluções desenvolvidas de forma a verificar se estas cumprem os objetivos propostos e solucionam os problemas identificados;
- **Conclusões:** Neste capítulo são apresentadas as conclusões obtidas com o trabalho efetuado.

2 Estado da Arte

Neste capítulo está descrita a informação obtida relativamente ao estado da arte que foi efetuado. O capítulo encontra-se dividido no estudo de conceitos e tecnologias relevantes para o desenvolvimento deste projeto (como o estilo arquitetural *REST*, evolução e versionamento de *API*, linguagens para especificação de *API REST* e identificação de possíveis registos de utilização da *API*) e no estudo de algumas *API*, incluindo a *API* anterior do E-goi e sua concorrência.

2.1 API

Uma *API* é um conjunto de funcionalidades utilizadas por programadores, atuando como uma interface entre sistemas e onde os desenvolvedores têm de seguir as suas restrições arquiteturais [11]. Devido à popularidade dos serviços *web*, as empresas disponibilizam frequentemente os seus serviços através de *web API* [12]. Sendo necessário um modelo de como interagir com aplicações *web*, surgiu, proposto por Fielding, o estilo arquitetural *Representational State Transfer (REST)* [13], que é analisado na secção 2.1.1.

Tal como *API* locais, *web API* têm necessidade de evoluir trazendo novas funcionalidades, mas também incompatibilidades [12]. Por isso, nesta secção é feito ainda um estudo relativo a evolução e versionamento, é feita uma análise de registos da utilização da *API* que passam ser importantes e linguagens de especificação para *API REST*.

2.1.1 REST

De forma a aumentar a escalabilidade de aplicações *web*, surgiu o estilo arquitetural *REST*, ao qual estão inerentes seis restrições [14]:

- **Cliente-servidor:** Ambas as aplicações, cliente e servidor, são implementadas e implantadas separadamente, desde que ambas estejam de acordo com a restrição "interface uniforme".
- **Interface uniforme:** Todos os componentes que interagem numa aplicação *web* devem estar de acordo com a mesma interface, o que implica a identificação de recursos, manipulação de recursos através de representações, mensagens auto descritivas e a utilização de *Hypermedia As The Engine Of Application State (HATEOAS)*;
- **Sistema de camadas:** Aplicações intermédias, como *proxies*, podem existir entre o cliente e servidor, desde que estejam de acordo com a interface uniforme;
- ***Cache*:** Atua nas respostas, reduzindo a latência e melhorando a performance geral de uma aplicação *web*;
- **Sem estado:** A aplicação servidora não tem conhecimento acerca do estado das aplicações clientes, obrigando assim os clientes a enviar informação contextual quando comunicam com o servidor;
- **Código-sob-demanda:** Temporariamente, os servidores podem transferir programas executáveis para os seus clientes para que estes possam entender e executar o código. Esta é a única restrição considerada opcional.

De acordo com o modelo de maturidade de Richardson, as aplicações *web* têm diferentes níveis de maturidade [15, 16, 14]:

- **Nível 0:** Nível mais básico de um serviço *web*, onde este possui apenas um *Uniform Resource Identifier (URI)* e um método *Hypertext Transfer Protocol (HTTP)* [16];
- **Nível 1:** Este é o nível do *URI*, onde vários *URI* são utilizados, existindo recursos individuais em vez de uma única rota, mas utilizando apenas um método *HTTP* [16];
- **Nível 2:** Neste nível são utilizados múltiplos *URI* e os métodos *HTTP* [16];
- **Nível 3:** O nível 3 é o nível de hipermédia, e pode ser alcançado ao implementar as restrições presentes no nível anterior e adicionando *links* para outros recursos, permitindo assim a transição entre os estados da aplicação [16].

Após alcançar o nível de hipermédia, é possível caracterizar uma *API* como sendo *RESTful* e o modelo de maturidade de Richardson é útil para declarar o que isso significa. Quando apenas o nível 1 ou 2 é alcançado, é utilizado o termo "*REST* pragmático" [17], em oposição ao termo "REST puro" [18].

Assim sendo, apesar de a maioria das *API REST* não se encontrarem desenhadas de forma uniforme, existem várias regras que podem ser seguidas para desenhar uma *API REST* de forma consistente [14]. Relativamente ao *URI* [14]:

- Devem ser usadas letras minúsculas e as palavras devem ser separadas por hífens e não por sublinhas;
- As extensões dos ficheiros não devem ser incluídas no *Uniform Resource Locator (URL)*;
- Devem ser usados subdomínios consistentes (por exemplo, no caso de *API* deve ser usado o subdomínio *"api"* e no caso de uma página de documentação relacionada com a *API* deve ser usado, por convenção, o subdomínio *"developer"*;
- Recursos separados por uma barra devem corresponder a diferentes níveis hierárquicos. Isto sugere que todos os níveis da hierarquia num *URL* podem ser acedidos individualmente;
- É possível a existência de identificadores numéricos ou alfanuméricos entre os níveis hierárquicos;
- Cada um dos níveis hierárquicos corresponde a um modelo de recurso, que podem ser entidade únicas, coleções ou ações. Para entidades únicas o recurso deve encontrar-se no singular, enquanto para coleções este deve encontrar-se no plural. Já para as ações devem ser utilizados verbos;
- Ações *Create, Read, Update, Delete (CRUD)* não devem ser usadas no *URL*. Ao invés, devem ser usados os métodos *HTTP*;

Outra característica importante relativamente a *REST* é o facto de utilizar os métodos e códigos de estado *HTTP*, sendo que *REST* é um estilo arquitetural que utiliza o protocolo *HTTP* como base. Os métodos *HTTP* possuem objetivos distintos e também possuem diferentes características, que são elas segurança e idempotência. Um método é considerado seguro quando não efetua nenhuma ação além de obter dados [19]. Idempotência significa que o resultado da operação é sempre o mesmo, seja executado uma ou mais vezes (casos de erro não são considerados). A Tabela 1 mostra as características segurança e idempotência dos métodos *"GET", "POST", "PUT" e "DELETE"* [19].

Tabela 1 - Segurança e idempotência dos métodos *"GET", "POST", "PUT"* e *"DELETE"*

Método HTTP	**Seguro**	**Idempotente**
GET	Sim	Sim
POST	Não	Não
PUT	Não	Sim
DELETE	Não	Sim

Também é importante considerar a importância dos *headers HTTP* para meta dados. Por exemplo, o *header "Content-Type"* deve ser utilizado para indicar o formato dos dados e o *header "Content-Length"* é útil para informar o cliente acerca do tamanho da resposta. Isto pode ser útil sobretudo quando o cliente necessita de saber o tamanho do conteúdo da mensagem sem ter de o transferir [14]. A utilização de *cache* também é encorajada, visto que permite diminuir a latência, aumentar confiança e diminuir a carga do servidor da *API* [14].

A utilização de *HATEOAS*, o último nível de maturidade no modelo de Richardson, implica que toda a informação necessária relativa a um estado da aplicação seja exposta a partir de *links* [14]. O Código 1 mostra um exemplo de utilização de *HATEOAS* na operação "GET: /lists/1/contacts/1" da nova *API*, no formato *JavaScript Object Notation (JSON)*.

```
{
  "name": "E-goi Dev",
  "email": "dev_teste@e-goi.com",
  "links": [
    {
      "rel": "self",
      "href": "api.e-goi.com/lists/1/contacts/1"
    },
    {
      "rel": "list",
      "href": "api.e-goi.com/lists/1"
    }
  ]
}
```

Código 1 - Utilização de *HATEOAS* na nova *API* do E-goi

2.1.2 Evolução e Versionamento

A evolução de uma *API* existe sempre que é feita uma mudança na mesma. Essas alterações podem ser compatíveis ou incompatíveis com os serviços já existente. Por exemplo, se for adicionado um parâmetro opcional a um serviço, este evolui de uma forma compatível com o serviço anterior, visto que não obriga o cliente a modificar o seu código. No entanto, se for adicionado um parâmetro obrigatório ao serviço, este evolui para uma versão incompatível com a versão anterior, sendo que código do cliente tem de ser modificado.

O versionamento de uma *API* ocorre à medida que esta evolui e existem vários padrões de alterações que devem ser considerados [6]. Podemos verificar que diferentes *API* utilizam diferentes técnicas de versionamento [6]. Algumas optam por ter apenas uma versão disponível (removendo as versões desatualizadas ao fim de algum tempo), enquanto outras mantêm disponíveis múltiplas versões da mesma *API* [6]. Ambas as abordagens possuem vantagens e desvantagens. A primeira é menos flexível pois obriga os utilizadores a migrarem sempre que é lançada uma nova versão [6]. A segunda abordagem oferece mais flexibilidade, no entanto exige um esforço de manutenção maior por parte de quem publica a *API*. Existem diversas maneiras de versionar uma *API*, entre elas [20]:

- Através do *header "Accept" (e. g. Accept: application/vnd.api.v1)*;
- No *URL (e. g. /api/contacts/v1)*;
- Utilizando um *header* personalizado *(e. g. Version: 1)*;
- Através de um parâmetro de consulta *(e. g. /api/contacts?version=1)*;
- Através da criação de um novo subdomínio.

Outro conceito de relevo relativamente a versionamento é o versionamento semântico. Versionamento semântico é uma nomenclatura de versionamento através da qual é possível verificar se uma versão é compatível com outra [6]. De acordo com o versionamento semântico, existem três tipos de alterações [21]:

- ***Major*:** Quando uma nova funcionalidade torna a nova versão criada é incompatível com as anteriores;
- ***Minor*:** Quando uma nova funcionalidade é adicionada sem causar incompatibilidade com as versões anteriores;
- ***Patch*:** Quando é efetuada a correção de um comportamento incorreto que não afeta a compatibilidade com versões anteriores.

Seguindo as regras do versionamento semântico, sempre que uma versão de um serviço é lançada, esta não deve ser alterada, devendo ser criada uma versão sempre que surge uma alteração [21]. Assim sendo, a primeira versão estável da *API* deve ser a versão "1.0.0". Por exemplo, uma alteração do tipo *Major* alteraria a versão para "2.0.0". Seguidamente, se fosse efetuada uma alteração do tipo *Patch*, a versão passaria a ser "2.0.1". Por fim, se fosse aplicada uma alteração *Minor*, a versão alteraria para "2.1.0".

A nova *API* do E-goi utiliza versionamento através de um *header* personalizado. Assim sendo, como é possível existir múltiplas versões do mesmo serviço, podem ser usadas diferentes estratégias de seleção [2]. Uma estratégia de seleção é responsável por selecionar a versão do serviço mais apropriada para o utilizador, com base nas suas preferências. Recorrendo ao versionamento semântico, além de ser possível selecionar uma versão específica de um serviço, é possível também definir diferentes estratégias de seleção de uma versão através de símbolos genéricos, onde o caracter "*" é referente à última versão. Por exemplo, ao utilizar um serviço com o *header* de versionamento com o valor "2.*", obtém-se a última versão *"minor"* cuja versão *major* é a versão 2. Com este *header*, o utilizador terá sempre a última versão compatível com a versão 2 do serviço correspondente. Por exemplo, assumindo a existência das seguintes versões para um serviço: 1.0.0; 1.1.0; 1.1.1; 1.1.2; 2.0.0; 2.1.0; 2.1.1; 3.0.0, a Tabela 2 mostra a versão executada pela *API* para diferentes *headers*.

Tabela 2 - Relação entre o *header* de versionamento e a versão executada pela *API*

Header	**Versão Executada**
Não envia header	3.0.0 (última)
*	3.0.0 (última)
1.1.1	1.1.1
1.1.*	1.1.2
2.*	2.1.1
2.*.*	2.1.1
1.2.*	Erro sendo que a versão não existe
*.1	Erro porque o formato do *header* é inválido

2.1.3 Linguagens para Especificação de API

No enquadramento tecnológico são analisadas e comparadas diversas tecnologias para especificação de *API REST*.

2.1.3.1 OpenAPI

A *Open API Initiative* (OAI) é uma iniciativa criada com o objetivo de criar uma linguagem para especificação de *API REST* independente de plataformas ou organizações. A versão 3.0 da *OpenAPI* foi a primeira disponibilizada pela OAI desde a sua doação, quando a *Swagger Specification* foi renomeada para *OpenAPI Specification* (OAS) [22].

A *OAS* permite especificar e descrever a lista de recursos e respetivas operações disponibilizadas por uma *API REST*, bem como a lista de parâmetros. Em cada parâmetro é possível indicar o seu nome, descrição, tipo de dados, valores que esse parâmetro pode assumir e se é obrigatório ou opcional. Também é possível definir tipos de autenticação e estruturas de dados que podem ser reutilizadas em diferentes pedidos ou respostas. Um documento baseado na *OAS* pode ser escrito em *JSON* ou em *YAML Ain't Markup Language (YAML)* e não requer um processo de desenvolvimento específico, podendo ser utilizado tanto uma abordagem *design-first* (criar a especificação e de seguida implementar a *API* com base no contrato criado) como *code-first* (implementar primeiro os serviços da *API* e em seguida gerar a especificação com base em anotações no código).

O Código 2, escrito em *YAML*, apresenta um exemplo de especificação de uma *API* com uma operação para obter informação de um contacto e a definição da estrutura de dados referente ao contacto. O termo *"openapi"* visa definir que a versão da especificação *OpenAPI* utilizada é "1.0.0", *"servers"* indica que o *URL* do servidor disponível para responder aos pedidos e na secção *"info"* é indicado que o título da *API* é *"Contacts API"* e a respetiva versão é "1.0.0". A instrução "*paths"* é utilizada para definir a rota *"/contacts/{id}"*. Dentro da rota é existe a instrução *"get"*, que corresponde à operação *HTTP*. A operação tem um identificador próprio (*"operationId"*), uma descrição (*"summary"*), o grupo ao qual pertence (*"tags"*) e os parâmetros de pedido e de resposta. Em cada parâmetro do pedido é indicado o seu nome, o local onde é enviado (neste caso, o *"id"* do contacto é enviado na rota) se é obrigatório ou opcional e a sua estrutura (neste caso o *"id"* é um número inteiro). Nos parâmetros da resposta é apresentado o código de resposta "200", a descrição e o conteúdo. No conteúdo é descriminado que o formato dos dados retornados é *JSON* e é referenciada uma estrutura *"Contacts".* Essa estrutura de dados é definida na secção *"schemas"* (subsecção de *"components"*). Na estrutura dos contactos são indicados os parâmetros obrigatórios e as propriedades de cada um.

```yaml
openapi: 3.0.0
servers:
  - url: 'http://localhost:8080/v1'
info:
  version: 1.0.0
  title: Contacts API
paths:
  '/contacts/{id}':
    get:
      summary: Info for a specific contact
      operationId: getContactById
      tags:
        - Contacts
      parameters:
        - name: id
          in: path
          required: true
          description: The id of the contact to retrieve
          schema:
            type: integer
      responses:
        '200':
          description: Successful request
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/Contact'
components:
  schemas:
    Contact:
      required:
        - id
        - name
        - phone
      properties:
        id:
          type: integer
        name:
          type: string
        phone:
          type: integer
          pattern: '^[2-9]\d{2}-\d{3}-\d{4}$'
```

Código 2 - Exemplo de especificação *OpenAPI 3.0*

Após a *Swagger Specification* ter sido renomeada, *Swagger* passou a ser considerado um conjunto de ferramentas utilizadas para implementar a *OAS* [22]. Essas ferramentas incluem:

- *Swagger Editor*, que permite ao utilizador construir uma especificação *OpenAPI* diretamente no navegador *web* e verificar os resultados em tempo real,
- *Swagger UI*, que é uma biblioteca utilizada para construir documentação interativa (no entanto não permite dividir a documentação em diversas páginas),
- *Swagger Codegen*, que permite a geração de código para diversas linguagens e *frameworks* (alguns exemplos são *Clojure*, *Go*, *JavaScript*, *Java*, *.NET*, *NodeJS*, *PHP*, *Python*, *Ruby*, *Scala*) com base na especificação e que é utilizada por várias empresas tais como Autodesk, Cisco, IBM, Red Hat e Upwork [23].

2.1.3.2 RESTful API Modeling Language

RESTful API Modeling Language (RAML) é a linguagem de especificação para *API REST* criada pela *MuleSoft*. Um documento *RAML* é escrito na linguagem *YAML* e permite descrever a estrutura de uma *API*, incluindo: recursos, operações permitidas, parâmetros, tipos de autenticação, formato de entrada e saída dos dados. O Código 3 apresenta a estrutura de um documento *RAML* com uma operação para obter informação de um contacto e a definição da estrutura do contacto.

No exemplo apresentado, as instruções *"title"*, *"version"* e *"baseUri"* indicam respetivamente que o título da *API* é *"Contacts API"*, a sua versão é "1.0.0" e o *URI* base é *"http://localhost:8080/v1*". De seguida é definido o tipo de dados dos contactos, com a instrução *"types"*. Como os contactos não são um tipo de dados primitivo, é possível utilizar a instrução *"properties"* para indicar os atributos do contacto. Após a definição dos contactos são indicadas as rotas disponíveis (neste caso *"/contacts/{id}"*). O parâmetro *"id"* é definido na instrução *"uriParameters"* e seguidamente é descrita a operação *HTTP* utilizada (*"get"*). A operação tem um nome e uma descrição e uma resposta, sendo que essa resposta tem, neste exemplo, o código de resposta "200", o formato de dados *JSON* e é associado com o tipo de dados *"Contact"* através da instrução *"type"*.

```
#%RAML 1.0
title: Contacts API
version: 1.0.0
baseUri: http://localhost:8080/v1
types:
  Contact:
    displayName: Contact
    type: object
    properties:
      id:
        required: true
        displayName: id
        type: integer
        format: int32
      name:
        required: true
        displayName: name
        type: string
      phone:
        required: true
        displayName: phone
        type: integer
        format: int32
/contacts/{id}:
  uriParameters:
    id:
      required: true
      displayName: id
      description: The id of the contact to retrieve
      type: integer
      format: int32
  get:
    displayName: Info for a specific contact
    description: Info for a specific contact
    responses:
      200:
        description: Successful request
        body:
          application/json:
            displayName: response
            description: Successful request
            type: Contact
```

Código 3 - Exemplo de especificação *RAML 1.0*

RAML incentiva a abordagem *design-first* e permite a reutilização de código através de definições (por exemplo é possível definir tipos, que atuam de maneira semelhante às estruturas de dados da *OAS*). Também fornecem extensões para vários *Integrated Development*

Environments (*IDEs*), ferramentas como *API Workbench,* um *IDE* criado especificamente para o *design e* especificação de *API*, *API Designer*, para descrever a *API* e visualizar os resultados em tempo real. Também disponibilizam a *API Modeling Framework*, uma ferramenta para utilização de *RAML* juntamente com *OAS* e geradores de código para algumas linguagens e *frameworks*: *JavaScript*, *Java*, *NodeJS*, *PHP*, *Python* e *Ruby*.

2.1.3.3 API Blueprint

API Blueprint [24] é uma linguagem para especificação de *API REST*, criada e suportada pela *Apiary*. A *Apiary* fornece ferramentas que, apesar de também suportarem *OAS*, utilizam a *API Blueprint* como especificação recomendada. São disponibilizadas ferramentas de geração de código, mas apenas para *.NET* e *NodeJS*.

Um documento da especificação *API Blueprint* é escrito em *Markdown Syntax for Object Notation (MSON)* e está subdividido em várias secções, sendo elas meta dados, nome e descrição, recursos, grupos de recursos e estruturas de dados. A secção dos recursos tem uma ação associada a cada recurso e a secção das estruturas de dados contém os parâmetros do pedido e da resposta da operação *REST*.

Um exemplo de especificação *API Blueprint* da definição de uma estrutura de dados e de um serviço para visualizar informação correspondente ao contacto está demonstrado no Código 4. A especificação apresentada utiliza a versão 1A, definida através da instrução *"FORMAT"* e tem como URI base *"http://localhost:8080/v1",* como demonstrado na instrução *"HOST"*. No bloco de código seguinte é possível verificar que o nome da *API* é *"Contacts API"* e que pertence ao grupo "*Contacts"*. Após a definição da estrutura base da *API* é definida a rota *"[/contacts/{id}"*, os parâmetros (incluindo o seu tipo de dados, descrição e se são obrigatórios ou opcionais) e a operação *HTTP* (*"get"*). Por fim é apresentada a resposta, indicando o formato dos dados como sendo *JSON* e o código de resposta "200". Ainda relativamente à resposta, é possível adicionar atributos, que podem ser tipos de dados primitivos ou complexos. Neste caso, existe o atributo *"Contact"*, um atributo complexo que está descrito na secção *"Data Structures"*.

```
FORMAT: 1A
HOST: 'http://localhost:8080/v1'

# Contacts API
# Group Contacts

## Contacts By Id [/contacts/{id}]
+ Parameters
   + id (number, required)
      The id of the contact to retrieve

### Info for a specific contact [GET]
Info for a specific contact

+ Response 200 (application/json)
      Successful request
   + Attributes (Contact)

# Data Structures
## Contact (object)
### Properties
+ `id` (number, required)
+ `name` (string, required)
+ `phone` (number, required)
```

Código 4 - Exemplo da especificação *API Blueprint 1A*

2.1.3.4 Comparação das Linguagens para Especificação de API REST

Relativamente às linguagens para especificação de *API REST*, todas as analisadas são bastante completas e têm funcionalidades semelhantes, apesar de a *API Blueprint* possuir menos funcionalidades relacionadas com reutilização de código (por exemplo, não é possível reutilizar o mesmo conjunto de *headers* em operações diferentes).

RAML e *API Blueprint* são usadas em abordagens *design-first*, enquanto a especificação *OpenAPI* não obriga à utilização de uma metodologia específica, podendo ser usada *design-first* ou *code-first*, ficando ao critério de quem especifica a *API* decidir a metodologia mais adequada.

Adicionalmente, no momento da escrita deste documento, a especificação *OpenAPI* é a especificação com a maior comunidade e maior número de contribuidores no *github* [10, 25, 26] e é a especificação utilizada pelas ferramentas do *Swagger*, logo é a que possui ferramentas de geração de código em mais linguagens.

2.1.4 Registos de Utilização de API

A determinação de quando e como uma *API* deve evoluir é uma das principais dificuldades relacionadas com a sua evolução [1]. De forma a tomar decisões que possam enriquecer o valor de negócio ao alterar a *API*, é necessária informação adequada, sendo assim importante perceber que utiliza a *API* e de que forma e qual o valor gerado a partir dela [7]. Portanto, uma *API* deve possuir a capacidade de medir, monitorar e reportar a utilização da *API* [7]. Assim, é importante gerar registos da sua utilização para posterior análise com a produção de relatórios que ajudem a detetar problemas e possíveis melhorias [1].

O registo da utilização da *API* deve oferecer registos relacionados com o acesso, consumo, performance e eventuais exceções [7]. Para gerar registos de utilização é importante definir um conjunto de dados que possam quantificar uma tendência ou comportamento. Alguns desses dados podem ser o tipo de dispositivo utilizado por quem utiliza a *API*, as aplicações a partir das quais a *API* é invocada e a região geográfica na origem do tráfego [7]. Também deve ser registado o endereço de *Internet Protocol (IP)* dos clients e a data e hora de quando o pedido foi recebido e a respetiva resposta enviada [7]. Vários tipos de meta informação tais como o *URI*, método *HTTP*, ou outra informação presente em *headers*, deve ser devidamente registada em cada chamada à *API*, visto que essa informação pode ser processada e mais dar originar relatórios para análise da *API* [7]. Alguns dos dados extraídos desta meta informação podem ser:

- Método do pedido *HTTP*;
- Tamanho do pedido;
- Tamanho da resposta;
- Recurso ou ação;
- Formato de dados do pedido;
- Formato de dados da resposta.

Métricas de performance da *API* e casos de erro e códigos de estado também devem ser registados, como parte do registo de utilização de uma *API* [7].

2.2 API no Mercado

De seguida, está sintetizada a análise da *API* anterior do E-goi e a análise das *API* dos principais concorrentes do E-goi. Por fim, é feita uma comparação entre as *API* analisadas com base em

critérios como as características da documentação, existência de *SDK* e apresentação de relatórios da utilização da *API*.

2.2.1 E-goi

A *API* anterior do E-goi oferece serviços em vários protocolos, entre eles *SOAP* [27]e *eXtensible Markup Language – Remote Procedure Call (XML-RPC)* [28]. Também oferece serviços *REST*, mas a *API REST* não segue diversas convenções deste estilo arquitetural. São utilizados os métodos *HTTP*, contudo são utilizadas ações no *URL* em vez de recursos.

A sua documentação está bastante incompleta. Não existem exemplos de pedidos nem de possíveis respostas. Devido ao facto de possuir exemplos de código em várias linguagens, torna-se um pouco mais fácil utilizar a *API.* No entanto, para algumas funcionalidades que surgiram à medida que a *API* evoluiu, não existem exemplos de código. A Figura 2 mostra a ausência de exemplos de código para o serviço de importar um conjunto de subscritores para uma lista de contactos.

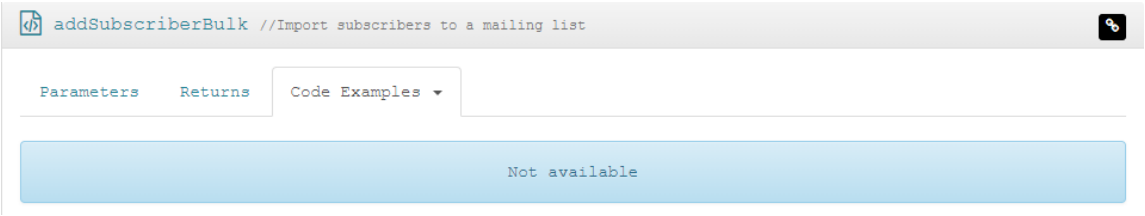

Figura 2 - Ausência de exemplos de código no serviço *addSubscriberBulk* da E-goi API V2

Os pontos positivos desta API são o facto de possuir uma lista de erros que podem ocorrer e os *SDK* para diversas linguagens.

2.2.2 MailChimp

MailChimp é uma plataforma de *marketing* digital multicanal que fornece serviços aos seus clientes através de uma *API REST*, que se encontra neste momento na versão 3. A sua *API* está bastante bem documentada, possuindo exemplos de pedidos e de resposta e a lista de erros possíveis. Além da documentação estática, oferecem ainda uma documentação interativa (ao qual denominam de *Playground*, que, apesar de não estar tão detalhada, permite ao utilizador interagir e experimentar a *API* através do navegador *web*).

As respostas da *API* do *MailChimp* utilizam *HATEOAS*. O Código 5 apresenta um exemplo de resposta, em *JSON*, de um serviço para listar os navegadores mais utilizados pelos subscritores

de uma determinada lista. Essa resposta contem meta dados que relacionam os dados do pedido efetuado com a respetiva lista de contactos.


```
{
 "clients": [
  {
   "client": "Gmail",
   "members": 50,
   "list_id": "57afe96172"
  }
 ],
 "list_id": "57afe96172",
 "total_items": 1,
 "_links": [
  {
   "rel": "self",
   "href": "https://usX.api.mailchimp.com/3.0/lists/57afe96172/clients",
   "method": "GET",
   "targetSchema": "https://api.mailchimp.com/schema/3.0/Lists/Clients/Collection.json"
  },
  {
   "rel": "parent",
   "href": "https://usX.api.mailchimp.com/3.0/lists/57afe96172/",
   "method": "GET",
   "targetSchema": "https://api.mailchimp.com/schema/3.0/Lists/Instance.json"
  }
 ]
}
```


Código 5 - Exemplo de resposta do serviço para obter navegadores *web* mais utilizados de uma lista de contactos da *API* do *MailChimp*

2.2.3 ActiveCampaign

ActiveCampaign possui uma versão 3 da *API*, que segue o estilo arquitetural *REST*. No entanto, esta não foi analisada porque ainda se encontra na versão *Beta*. Assim sendo, foi analisada a versão 2 da *API*, que utiliza como protocolo de transporte o *HTTP* e segue um estilo arquitetural semelhante ao utilizado no protocolo *RPC*. Esta versão da *API* mostra diversos exemplos da sua utilização, desde exemplos de pedido e resposta, a exemplos da sua utilização na linguagem de programação *PHP.* Contudo não apresenta ao utilizador quais são os possíveis erros que podem surgir.

Também disponibilizam *SDK*, denominando estes de *API Wrappers*, para *PHP*, *NodeJS*, *Ruby* e *Python*. Os *SDK* são a abordagem recomendada para a utilização da *API* do *ActiveCampaign*, visto

que facilitam a utilização da *API* por parte do utilizador e são totalmente suportados pela *ActiveCampaign*, estando sempre atualizados à medida que a *API* evolui.

```php
<?php
require_once("includes/ActiveCampaign.class.php");
$ac = new ActiveCampaign("API_URL", "API_KEY");
$account = $ac->api("account/view");
print_r($account);
```

Código 6 - Exemplo da utilização do *SDK* de *PHP* do *ActiveCampaign* para a visualização da informação da conta

O Código 6 mostra a simplicidade de uma chamada à *API* para visualizar a informação da conta através do *SDK* de *PHP*, onde o utilizador do *SDK* necessita apenas de o instanciar com os seus dados (*URL* e chave da *API* fornecidos após a criação de uma conta) e invocar a função correspondente à operação que pretende efetuar.

2.2.4 Infusionsoft

Infusionsoft oferece os seus serviços aos utilizadores através de uma *API REST* e de uma *API XML-RPC*. Ao contrário da *API* do E-goi em que as diferentes *API* (*SOAP*, *REST* e *XML-RPC*) possuem uma página de documentação comum entre elas, a *Infusionsoft* separou totalmente as *API*, possuindo uma documentação diferente para cada. De facto, as *API* da *Infusionsoft* possuem algumas diferenças entre elas além do protocolo/estilo arquitetural. A documentação da *API XML-RPC*, ao contrário da *API REST*, não apresenta os erros que podem ocorrer quando um pedido é efetuado, enquanto a *API REST* não possui exemplos de código como os demonstrados na documentação da *API XML-RPC.*

Além disso, a documentação da *API REST* é interativa, já a de *XML-RPC* é estática, mas possui uma página de documentação interativa separada da documentação original (à semelhança do *Playground* da *API* do *MailChimp*). Ambas possuem exemplos de pedido e de resposta e são suportadas pelos *SDK* (existem *SDK* diferentes para cada uma das *API*, apesar de conterem as mesmas funcionalidades).

2.2.5 GetResponse

A *GetResponse API* segue o estilo arquitetural *REST* e encontra-se na versão 3. Possui uma documentação completa, na medida em que mostra exemplos de pedidos, exemplos de

respostas e os possíveis códigos de resposta de sucesso, bem como possíveis erros, que podem ser retornados numa determinada operação. Disponibilizam apenas um *SDK*, para a linguagem *PHP*.

Quantos aos exemplos de resposta demonstrados na documentação desta *API*, é possível verificar que aplicam *HATEOAS,* sendo que retornam meta dados (nomeadamente *links*). O Código 7, que mostra um exemplo de resposta, em *JSON*, de um pedido para obter um formulário, é possível verificar o retorno de *links* para facilitar a navegação.


```
{
   "webformId": "Fur",
   "formId": "Fur",
   "name": "Webform - 07.08.2015",
   "href": "https://api.getresponse.com/v3/forms/Fur",
   "campaign": {
      "campaignId": "o5lx",
      "href": "https://api.getresponse.com/v3/campaigns/o5lx",
      "name": "campaign_name55f69a1b0ff01"
   },
   "hasVariants": "true",
   "scriptUrl":
"https://app.getresponse.com/view_webform_v2.js?u=ycng&webforms_id=113103",
   "status": "published",
   "createdOn": "2015-08-07T07:37:57+0000",
   "statistics": {
      "visitors": 0,
      "uniqueVisitors": 0,
      "subscribed": 0
   }
}
```


Código 7 - Exemplo de resposta da *GetResponse API* ao visualizar informação de um formulário

2.2.6 Comparação da API do E-goi com API Concorrentes

De seguida é feita uma comparação entre as *API* analisadas anteriormente. A Tabela 3 mostra essa comparação, estando assinaladas com um "X" as características que a *API* possui e com um "-" as características que não possui. Essas características estão identificadas com números de 1 a 6 e são as seguintes:

1. Documentação possui exemplo de pedido;
2. Documentação possui exemplo de resposta;
3. Documentação possui lista de erros possíveis;

4. Apresenta exemplos de código para suporte à documentação;
5. Possui *SDK*;
6. Possui uma página de documentação interativa.

Tabela 3 - Comparação da E-goi *API V2* com *API* de concorrentes

	E-goi	**Mail Chimp**	**Active Campaign**	**Infusionsoft REST**	**Infusionsoft RPC**	**Get Response**
1	-	X	X	X	X	X
2	-	X	X	X	X	X
3	X	X	-	X	-	X
4	X	-	X	-	X	-
5	X	-	X	X	X	X
6	-	X	-	X	X	-

Como podemos verificar a partir da Tabela 3, a *API* pública do E-goi é a única que não possui exemplos de pedido ou resposta, o que leva a um aumento do esforço de aprendizagem da *API* por parte do utilizador. Todas as *API,* com exceção da *ActiveCampaign API* e da *API RPC* da *Infusionsoft,* indicam os erros que podem ocorrer e a razão pela qual estes ocorrem, permitindo assim aos utilizadores corrigir mais facilmente eventuais incorreções nos pedidos.

As *API* do E-goi, *ActiveCampaign* e *Infusionsoft (RPC)* optam por mostrar exemplos de código em várias linguagens para que a *API* seja fácil de implementar. Além disso, todas as *API* analisadas (excetuando a do *MailChimp*) disponibilizam *SDK*, que são uma mais-valia para clientes que desejem integrar o *software* presente nessas *API* sem ter de lidar com lógica de pedidos *web.* As *API* do *Infusionsoft* e *MailChimp* possuem ainda páginas de documentação interativa para que os utilizadores possam testar e aprender a utilizá-las à medida que leem a documentação.

3 Análise de Valor

Neste capítulo é feita a análise de valor do trabalho proposto, onde é descrita a fase inicial do processo de negócio e inovação, valor para o cliente, a proposta de valor, o modelo de negócio *Canvas*, a cadeia de valor de *Porter* e a aplicação do método de tomada de decisão *Analytic Hierarchy Process (AHP)*.

3.1 Processo de Negócio e Inovação

O processo de inovação encontra-se dividido em três fases diferentes, sendo estas o *Fuzzy Front End (FFE)*, o *New Product Development (NPD)* e a comercialização [29], como demonstrado na Figura 3.

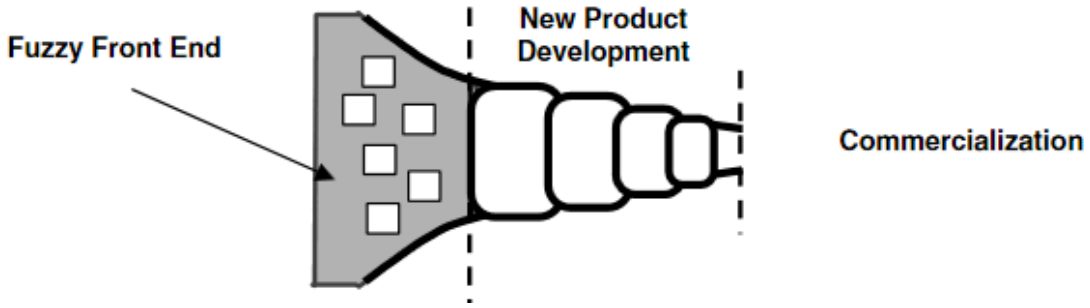

Figura 3 - Áreas do processo de inovação [29]

O *FFE* é caracterizado como sendo a fase do processo de inovação onde existem mais oportunidades de melhoramento. O *FFE* possui uma natureza de trabalho experimental e frequentemente caótica, ao contrário do *NPD*, cujo trabalho é orientado a objetivos, seguindo um plano de projeto [29]. A comercialização do *FFE* é imprevisível e incerta e o seu financiamento é variável, no entanto o *NPD* tem um certo grau de certeza relativamente à comercialização e o seu financiamento é orçamentado. Outra das características do *FFE* é que as suas receitas esperadas são bastantes incertas, enquanto no *NPD* as receitas são cada vez mais previsíveis à medida que se aproxima o seu lançamento.

A inexistência de definições comuns que permitissem a comparação do *FFE* entre empresas levou à criação do modelo *New Concept Development (NCD)* [29]. Portanto, o objetivo do *NCD* é proporcionar uma terminologia comum para o *FFE*, dividindo-se em três partes fundamentais [29], como é possível observar na Figura 4:

- O centro do modelo representa a liderança, cultura e estratégia de negócio da empresa, que conduz e controla os cinco elementos-chave do *FFE*;
- As áreas internas que rodeiam o centro do modelo definem os elementos-chave do *FFE*, sendo estes a identificação de oportunidade, análise de oportunidade, geração e enriquecimento das ideias, seleção da ideia e definição de conceito;
- A área externa é composta pelos fatores de influência da empresa que dificilmente são controlados por esta (questões políticas, legais, económicas, clientes, concorrentes, ciência e tecnologia). Estes fatores afetam todo o processo de comercialização.

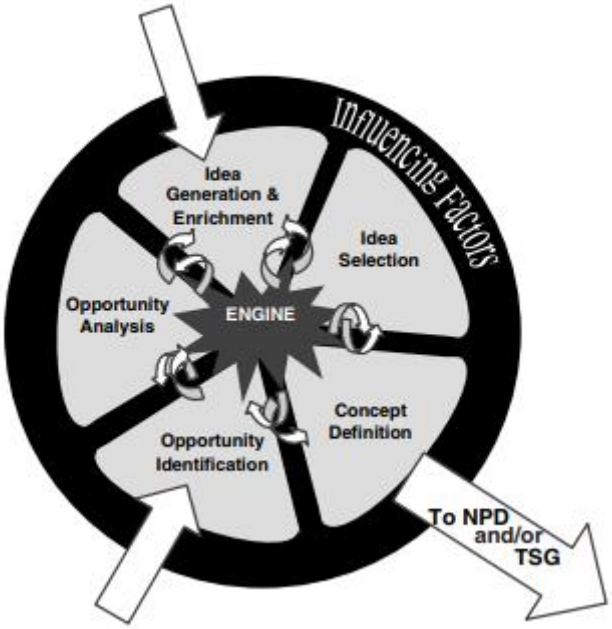

Figura 4 - Modelo *NCD* [29]

Através da Figura 4 é ainda possível verificar que as ideias passam por um processo cíclico e iterativo, fluindo pelos cinco elementos, podendo passar várias vezes pelo mesmo elemento [29]. Todo este ciclo atrasa o *FFE*, no entanto encurta o tempo e as alterações necessárias nas fases do *NPD* e da comercialização [29].

Os elementos do *NCD* para este projeto, juntamente com os métodos, técnicas e/ou ferramentas necessárias para analisar cada um dos desses elementos, são de seguida apresentados:

- **Identificação de oportunidade:** A identificação de oportunidade pode ser efetuada através de *roadmapping*, análise de tendências tecnológicas, análise das tendências dos clientes, análise de inteligência competitiva, pesquisa de mercado e planeamento do cenário [29]. Para este projeto, é possível verificar através da análise de inteligência competitiva (análise das *API* de concorrentes do E-goi), análise de mercado e tendências

dos clientes (senhas respondidas pela equipa de suporte ao cliente do E-goi) que existe uma oportunidade de melhorar significativamente a qualidade da documentação da *API REST e SOAP*. Pela análise de tendências tecnológicas é possível verificar que a documentação, *SDK* e testes podem ser gerados de forma dinâmica usando a uma tecnologia para especificação de *API.* Ainda com a análise de tendências tecnológicas, confirma-se que é necessário gerar registos de utilização da *API REST* e *API SOAP* para futura monitorização e geração de relatórios;

- **Análise de oportunidade:** A análise de oportunidade visa considerar se as oportunidades identificadas têm valor suficiente, podendo utilizar-se as mesmas ferramentas que na identificação de oportunidade, mas de forma mais detalhada [29]. Neste caso, é também através das tendências dos clientes, da análise de inteligência competitiva, da análise tecnológica e também da análise de mercado efetuada pela empresa E-goi ao *REST* e ao *SOAP* que conclui que *REST* é atualmente o estilo arquitetural mais utilizado e, portanto, o suporte para *REST* deve ser o foco para este trabalho;
- **Geração e enriquecimento de ideias:** Existem vários métodos para este elemento, entre os quais mecanismos para comunicar competências, capacidades e tecnologias partilhadas pela empresa, inclusão de pessoas de diferentes estilos cognitivos na equipa de enriquecimento da ideia e identificação de novas soluções tecnológicas, que foram os métodos utilizados para a geração da ideia. Após a aplicação destas técnicas foram geradas ideias que sugerem a utilização de *OAS*, *RAML* ou *API Blueprint* para especificação da *API REST*, permitindo a geração de documentação, *SDK* e testes;
- **Seleção da ideia:** Fazer a seleção de uma boa ideia é algo crítico para o sucesso de um negócio [29]. Os métodos existentes para a seleção da ideia são metodologias de portfólio com múltiplos fatores que utilizam medidas com indicadores numéricos, processo de seleção de ideias formais com *feedback* imediato para quem apresenta essas ideias e o uso da teoria de opções para avaliar as ideias [29], que foi utilizado neste trabalho a partir do método *AHP*, descrito na secção 3.6, concluindo que a melhor opção para especificação da *API REST* seria a *OAS*;
- **Definição do conceito:** A definição do conceito é o último elemento do *NCD* e é o único elemento que permite prosseguir para o *NPD* [29]. Das várias técnicas que podem ser utilizadas neste elemento foram utilizadas a rápida avaliação de inovações de alto potencial, a definição de objetivos (presentes na secção 1.3) e envolvimento precoce dos clientes nos testes do produto (visto que a E-goi possui equipas de desenvolvimento que

fazem projetos à medida para clientes, utilizando a *API* pública e consequentemente a sua documentação).

3.2 Valor para o Cliente

Na presente secção é feita, inicialmente, a definição de "Valor", "Valor para o Cliente" e "Valor Percebido".

- **"Valor":** Valor pode ser definido em vários contextos teóricos como a necessidade, desejo, interesse, critérios, crenças, atitudes e preferências [30];
- **"Valor para o Cliente":** Valor para o cliente por ser traduzido como o resultado da combinação dos sacrifícios e benefícios trazidos ao cliente [31];
- **"Valor Percebido":** Valor percebido pode ser entendido como a avaliação geral de um cliente acerca da utilidade de um produto ou serviço, tendo como base o que é pago e o que é recebido [32, 33].

O valor para o cliente pode ser dividido temporalmente em "Antes de compra", "Durante a compra", "Depois da compra" e "Após uso" [31]. Como analisado na secção 3.3, este projeto tem dois segmentos de clientes, sendo necessário criar duas perspetivas longitudinais de valor, uma delas para a E-goi (Tabela 4) e outra para os clientes da E-goi (Tabela 5), incluindo os benefícios ou sacrifícios associados à criação e utilização do produto.

Tabela 4 - Perspetiva longitudinal de valor para o E-goi

	Benefícios	**Sacrifícios**
Antes da compra	Existência de documentação e *SDK*.	Atualizar documentação e *SDK* manualmente.
Durante a compra	-	Colocar o projeto em produção e notificar clientes da E-goi; Aquisição de servidores e diversos *IP.*
Depois da compra	Baixos custos de manutenção da documentação, *SDK* e testes para a nova *API*.	-
Após uso	Benefícios funcionais devido à existência de testes e de registos de utilização.	Elevada quantidade de registos guardados.

Como podemos visualizar na Tabela 4, antes da compra deste projeto, a E-goi possui documentação e *SDK*, no entanto, estes precisam de ser atualizados de forma manual. Durante a compra, a E-goi tem custos relacionados com colocar o projeto em produção, tem de notificar os seus clientes para a existência de uma nova *API* e tem de adquirir servidores e endereços de *IP.* Depois da compra, a documentação, os *SDK* e os testes passam a ser gerados de forma dinâmica, reduzindo os custos de manutenção necessários. Após o uso, o E-goi passa a possuir testes e registos de utilização da *API*, que apesar de serem benéficos e essenciais para a deteção e correção de erros, implicam o sacrifício de guardar uma grande quantidade desses mesmos registos.

Tabela 5 - Perspetiva longitudinal de valor para os clientes do E-goi

	Benefícios	**Sacrifícios**
Antes da compra	Acesso à documentação da *API*; Utilização da *API* através de *SDK*.	Elevada dificuldade a entender a documentação.
Durante a compra	-	Migrar o *software* para a nova versão da *API*.
Depois da compra	Acesso à nova documentação da *API*; Utilização da *API* através de *SDK*.	-
Após uso	Documentação detalhada que permite a fácil compreensão da *API*; Correção rápida de erros.	-

A partir da Tabela 5 é possível verificar que os benefícios para os clientes da E-goi antes e depois da compra são os mesmos, porém o sacrifício relacionado com a elevada dificuldade que existia a entender a documentação (antes da compra) deixa de existir (depois da compra). Os clientes da E-goi, durante a compra, têm ainda de migrar o seu *software* para a nova versão da *API*. Por fim, após o uso, é permitido aos clientes do E-goi utilizar a *API* de forma fácil, devido à documentação detalhada. O mesmo se aplica à correção de erros (incluindo falhas da *API* e/ou erros causados pelo cliente), que se torna mais fácil devido à criação de registos de utilização da *API* e à geração dinâmica de testes.

3.3 Proposta de Valor

A proposta de valor indica o conjunto de produtos/serviços que criam valor para um segmento específico de clientes [34, 35]. O projeto desta dissertação tem dois segmentos de distintos: os

clientes do E-goi e a própria empresa E-goi. Portanto, são necessárias duas propostas de valor distintas.

3.3.1 Proposta de Valor para os Clientes do E-goi

Os clientes do E-goi vão poder usufruir das vantagens que advêm do uso de *SDK*, sendo que este abstrai o cliente de toda a lógica aplicacional relacionada com pedidos *web*.

A documentação é também uma componente essencial em qualquer *API* e descreve os serviços necessários e como utilizá-los. Para nova versão da *API REST*, a documentação será um dos focos principais de forma a facilitar o uso da *API* e aumentar a satisfação dos clientes.

Além dos *SDK* e da nova documentação, existe na nova *API* um mecanismo para gerar testes e outro para gerar registos de utilização da *API*. Isto permite que potenciais falhas, sejam estas falhas do cliente ou erros da *API*, possam ser facilmente identificadas e corrigidas.

3.3.2 Proposta de Valor para a Empresa E-goi

A proposta de valor para a empresa E-goi consiste na especificação detalhada da nova *API REST* do E-goi. A especificação da *API* vai permitir a geração dinâmica de documentação, código para os *SDK* e testes. Este processo traz valor à E-goi ao reduzir o custo de manutenção necessário para atualizar estes artefactos, uma vez que passam a ser gerados dinamicamente.

Por fim, é ainda registada a utilização da *API*. Este sistema vai permitir posteriormente que a empresa E-goi analise registos e, juntamente com o sistema de geração de testes, sejam encontrados problemas com maior facilidade.

3.4 Modelo de Negócio Canvas

A compreensão do modelo de negócio deve ser o ponto de partida para qualquer discussão, reunião ou inovação [35]. O modelo de negócio *Canvas* é uma ferramenta muito útil para definição do modelo de negócios de um qualquer projeto. Este modelo está dividido em segmentos de clientes, proposta de valor, canais, relacionamento com clientes, fontes de receita, recursos principais, atividades-chave, parcerias principais e estrutura de custo [35]. A Figura 5 ilustra o modelo de negócio *Canvas* elaborado para o projeto desta dissertação, onde estão presentes os seguintes elementos:

- **Segmentos de clientes:** Os segmentos de clientes são os clientes do E-goi que querem utilizar a *API* do E-goi e a própria empresa E-goi;
- **Proposta de valor:** Este projeto traz valor a dois segmentos distintos - a empresa E-goi e os seus clientes. A especificação da *API* permite a geração dinâmica de documentação, a geração dinâmica de *SDK* e a geração de testes, trazendo valor à E-goi na medida em que reduz os custos de manutenção compreendidos por estas componentes. Os *SDK* trazem valor aos clientes do E-goi pois abstrai-os da lógica relacionada com pedidos *web.* A criação de uma documentação detalhada também traz valor aos clientes do E-goi que querem utilizar a *API*, pois permite-lhes aprender a utilizá-la de forma fácil. A geração dinâmica de testes e o registo da utilização da *API* trazem valor a ambos, visto que permite à E-goi detetar erros e analisar e obter informação acerca de como a *API* está a ser utilizada pelos seus clientes, corrigindo assim eventuais falhas, sejam estas falhas internas ou erros causados pela inexperiência do utilizador;
- **Canais:** Os canais de distribuição do produto são a plataforma E-goi, o *blog*, a *API* anterior do E-goi (que sugere a utilização da nova *API* reencaminhando os utilizadores para a sua documentação) e também a nova *API* do E-goi, que remete os seus utilizadores para a documentação quando ocorrem erros nos pedidos;
- **Relacionamento com clientes:** As relações com os clientes deste trabalho são fundamentalmente diferentes para cada segmento. Apesar de ambos constituírem relações a longo prazo, com a empresa E-goi a relação é de assistência pessoal. Já para os clientes do E-goi a relação é essencialmente através de processos automáticos (conhecimento da *API* através da documentação e utilização dos *SDK*). No entanto, a relação com os clientes do E-goi pode tornar-se também uma relação de assistência pessoal que é estabelecida através do suporte ao cliente do E-goi e do seu *blog*, caso este tenha dúvidas na utilização dos serviços oferecidos;
- **Fontes de receita:** As fontes de receita são os lucros provenientes da utilização da *API*;
- **Recursos principais:** Os recursos principais para desenvolver este projeto são a nova *API* do E-goi, a *OpenAPI Specification (OAS)*, utilizada para especificar a *API REST*, *Zend Framework*, que é a ferramenta utilizada para desenvolver o sistema de registos da *API* e sistema de geração de testes (ver secção 4.1.2), e também o *hardware* necessário, que inclui o computador para desenvolvimento e os servidores necessários para a implantação das soluções;

- **Atividades-chave:** As atividades-chave deste projeto são investigar (*REST*, evolução e versionamento, tecnologias para especificação de *API REST*, registos de utilização da *API* que possam ser importantes e outras *API* no mercado), especificar a nova *API* do E-goi (para possibilitar a geração de documentação, *SDK* e testes dinamicamente), desenhar e implementar e testar e avaliar as soluções desenvolvidas;
- **Parcerias principais:** Os parceiros deste projeto são *Zend Technologies* (visto que os sistemas a implementar utilizam *Zend Framework*, como é possível observar na secção 4.1.2) e *Open API Initiative* (sendo que a *OAS* é a especificação de *API REST* mais adequada neste contexto, como é possível verificar através do método *AHP* na secção 3.6);
- **Estrutura de custo:** As estruturas de custos são os gastos no *hardware* necessário (computadores para desenvolvimento e servidores), gastos para alojamento da documentação, custos de desenvolvimento e custos de manutenção.

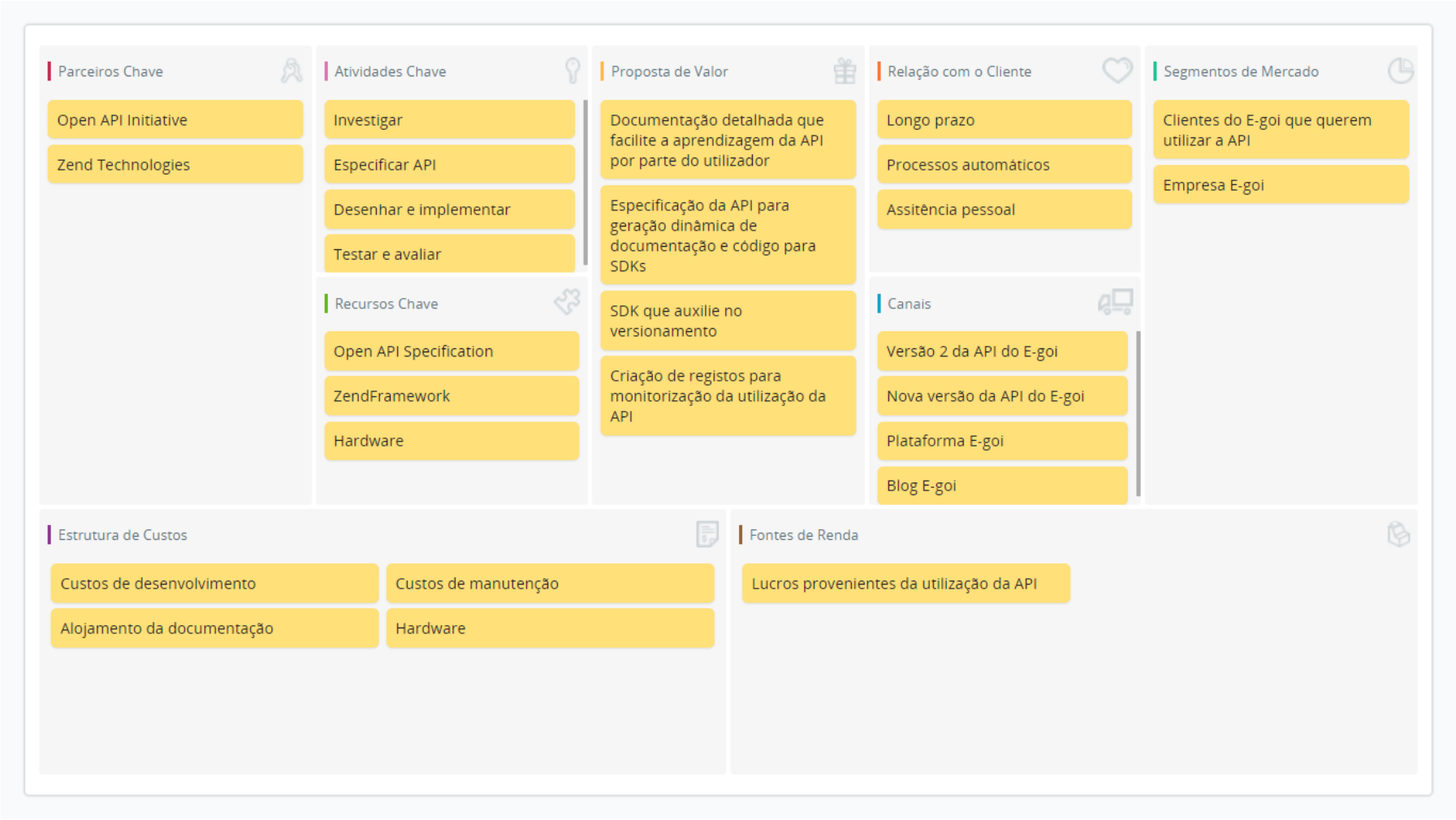

Figura 5 - Modelo de negócio *Canvas*

3.5 Cadeia de Valor de Porter

Uma empresa é um conjunto de atividades que são realizadas para desenhar, produzir, promover, entregar e suportar um produto [36]. Essas podem ser representadas usando uma cadeia de valor [36], nomeadamente a cadeia de valor de *Porter*.

A Figura 6 apresenta a cadeia de valor de *Porter* relativamente às atividades necessárias por parte da empresa E-goi para suportar este produto, onde as atividades são as seguintes:

- **Logística de entrada:** Dados para utilização dos serviços E-goi;
- **Operações:** Investigar soluções tecnológicas, desenvolver *software*, testar e avaliar *software* e documentar o *software* desenvolvido;
- **Logística de saída:** Utilização dos serviços do E-goi (através do *SDK*) e produção de relatórios;
- ***Marketing* e vendas:** Angariação de clientes e redação de artigos para o *blog* E-goi;
- **Serviço:** Correção de erros de erros detetados nos registos de utilização;
- **Infraestrutura:** Contabilidade, assuntos legais e planeamento;
- **Recursos humanos:** Recrutamento, seleção, avaliação e promoção;
- **Desenvolvimento e tecnologia:** Adição de funcionalidades ao produto, correção de erros e tratamento de problemas de segurança;
- **Obtenção de recursos:** Aquisição de computadores para desenvolvimento e aquisição de servidores.

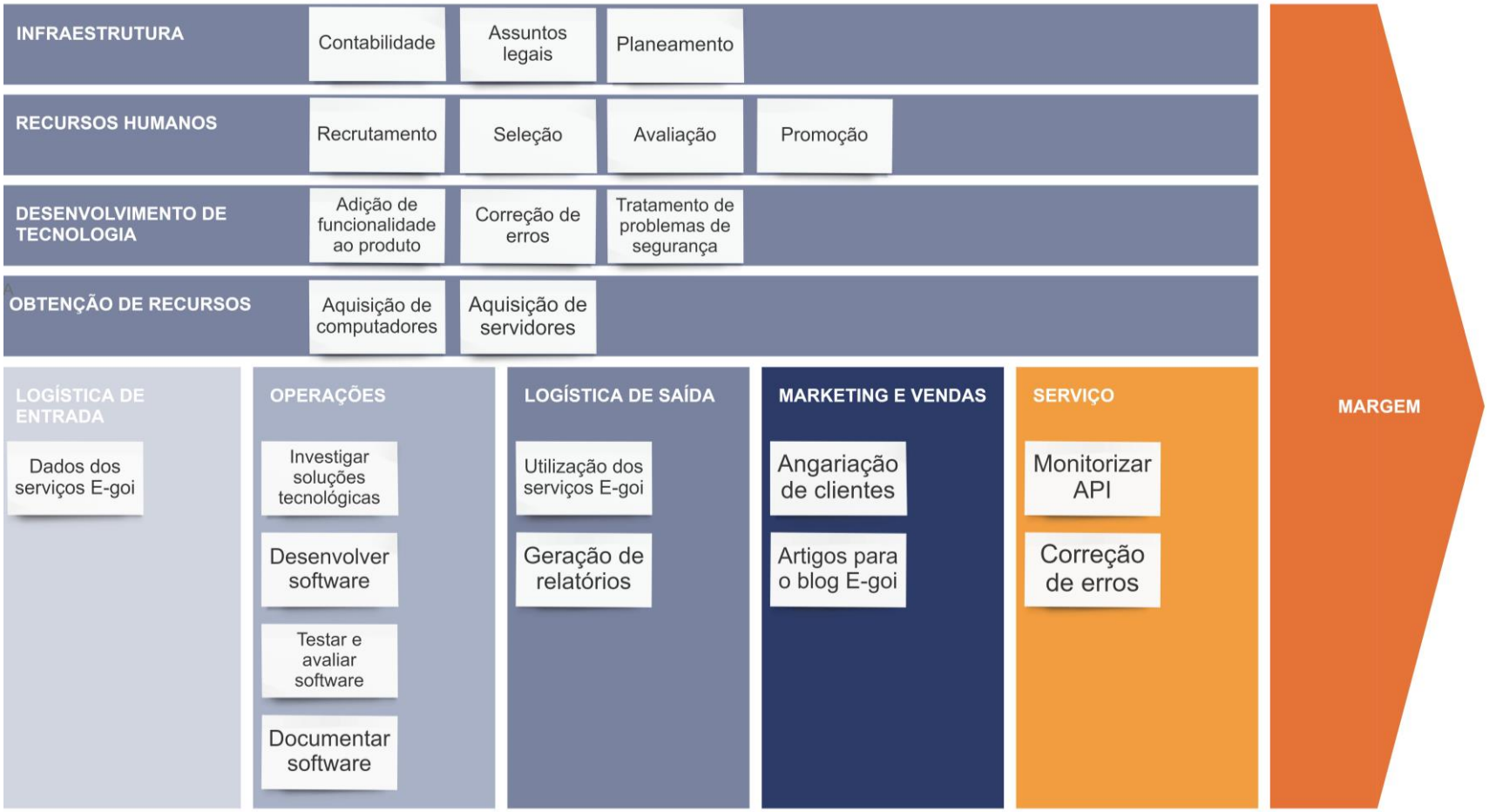

Figura 6 - Cadeia de valor de *Porter*

3.6 Método AHP

O processo de tomada de decisão é algo que está, muitas vezes, sujeito a subjetividade. Para oferecer clareza ao processo de tomada de decisão e de forma a tomar uma decisão de forma menos subjetiva pode ser utilizado o método *AHP* [37]. Para obter prioridades acerca de uma decisão, este método encontra-se dividido em vários passos [38]:

1. Definição do problema e o tipo de conhecimento procurado;
2. Estruturação da hierarquia, começando com o objetivo, seguido dos critérios e subcritérios da decisão e finalizando com o conjunto de alternativas;
3. Criação de matrizes de comparação;
4. Usar as prioridades obtidas nas comparações para medir as prioridades imediatamente abaixo e calcular a sua prioridade global.

Para fazer comparações nas matrizes de comparação é necessária uma escala que indique de forma numérica a importância ou dominância que um elemento tem sobre outro [38]. A apresenta uma versão da escala numérica utilizada no método *AHP* proposta por *Thomas L. Saaty*, onde estão presentes os valores utilizados nas matrizes de comparação, juntamente com a sua definição e explicação.

Tabela 6 - Escala fundamental de Saaty [24]

Intensidade da importância	Definição	Explicação
1	Igual importância	Os elementos contribuem igualmente para o objetivo
3	Moderada importância	A experiência e a opinião favorecem um elemento sobre outro
5	Forte importância	A experiência e a opinião favorecem fortemente um elemento sobre outro
7	Importância muito forte	Um elemento é fortemente favorecido e a sua dominância é demonstrada na prática
9	Extrema importância	Existem evidências que favorecem um elemento sobre outro no maior grau de afirmação possível
2, 4, 6, 8	Valores intermediários entre importâncias	Dúvida entre intensidades de importância

No contexto desta dissertação, o método *AHP* pode ser aplicado para decidir qual a tecnologia de especificação de *API REST* mais adequada para especificar a nova *API* do E-goi. Assim sendo, o objetivo é a especificação da *API.*

Com base na comparação feita na secção 2.1.3.4, os critérios são os seguintes:

- **Ferramentas de geração de código:** Quantidade de linguagens suportadas pelas ferramentas de geração de código existentes que utilizam a especificação e que podem ser usadas na geração de documentação e *SDK*;
- **Comunidade:** Contribuidores e tamanho da comunidade no *github*;
- **Reaproveitamento de código:** Capacidade de reutilizar código da especificação, incluindo *headers*, parâmetros e respostas.

As alternativas a considerar, tecnologias de especificação de *API REST* analisadas na secção 2.1.3.4 são as seguintes:

- *OpenAPI*;
- *RAML*;
- *API Blueprint.*

A Figura 7 sumariza a informação relativa ao objetivo, critérios e abordagens, apresentando a estrutura hierárquica, divida em objetivo, critérios e alternativas. É possível verificar que, nesta demonstração da utilização do *AHP,* todas as abordagens podem ser comparadas mediante todos os critérios e nenhum dos critérios tem subcritérios.

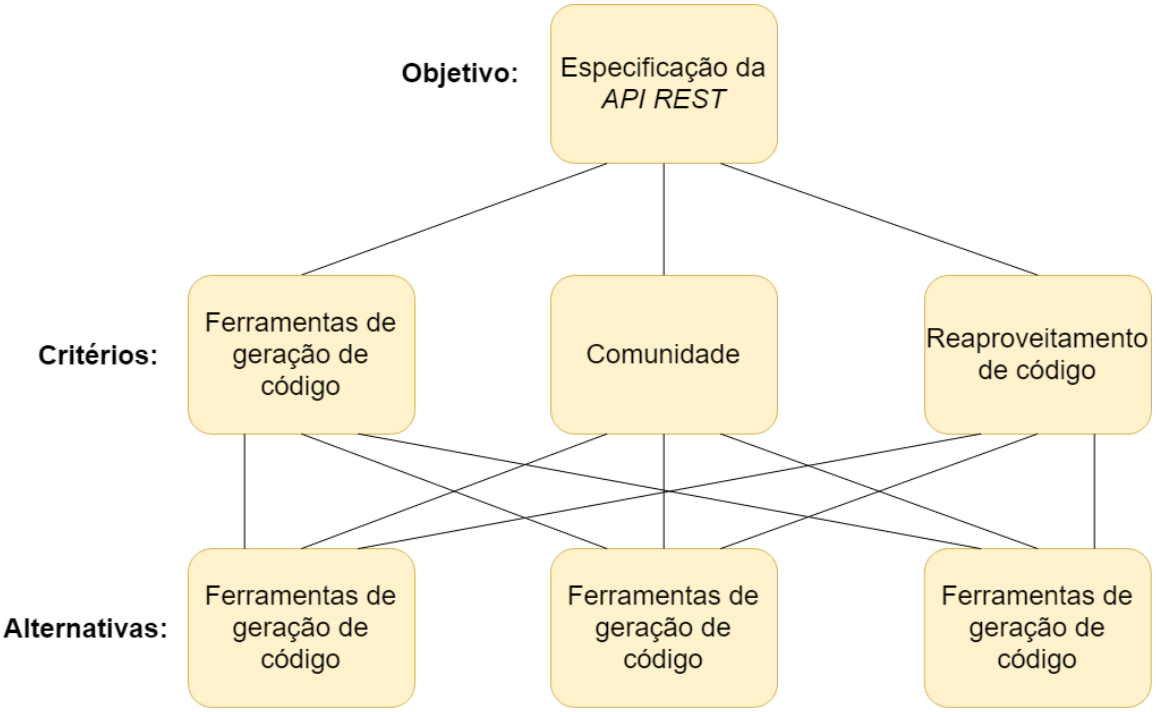

Figura 7 - Estrutura hierárquica para aplicação do método *AHP*

Depois de definir a estrutura hierárquica, é necessário criar matrizes de comparação entre critérios e alternativas (para cada critério) [38]. A Tabela 7 apresenta a tabela de comparação entre os critérios face ao objetivo.

Tabela 7 - Matriz de comparação entre os critérios

Especificação da API	**Ferramentas de geração de código**	**Comunidade**	**Reaproveitamento de código**
Ferramentas de geração de código	1	5	3
Comunidade	$\frac{1}{5}$	1	$\frac{1}{3}$
Reaproveitamento de código	$\frac{1}{3}$	3	1

De seguida é necessário calcular o vetor de prioridades[2]. Para obter o vetor de prioridades dos critérios, é necessário normalizar a matriz [37]. Para normalizar a matriz divide-se cada elemento da matriz pela soma de todos os elementos da respetiva coluna [37]. Para a matriz de comparação demonstrada na Tabela 7, a matriz normalizada é a seguinte:

$$\begin{bmatrix} 0{,}6522 & 0{,}5556 & 0{,}6923 \\ 0{,}1304 & 0{,}1111 & 0{,}0769 \\ 0{,}2174 & 0{,}3333 & 0{,}2308 \end{bmatrix}$$

Através da matriz normalizada é possível obter o vetor de prioridade calculando a média de cada linha da matriz [37]. A Tabela 8 apresenta os resultados obtidos no cálculo do vetor de prioridades e a prioridade em percentagem para cada um dos critérios. Como é possível verificar, a prioridade do critério "Ferramentas de geração de código" é de 63,33%, "Comunidade" é de 10,62% e "Reaproveitamento de código" é 26,05%.

[2] Neste exemplo demonstrativo do método *AHP*, todos os resultados são calculados com valores exatos independentemente do número de casas decimais com que os resultados são apresentados.

Tabela 8 - Prioridades dos critérios

	Prioridade	**Prioridade (percentagem)**
Ferramentas de geração de código	0,6333	63,33%
Comunidade	0,1062	10,62%
Reaproveitamento de código	0,2605	26,05%

Uma das preocupações do método *AHP* é verificar medir e melhorar a consistência das opiniões das matrizes de comparação [38]. Por essa razão, é necessário verificar se as matrizes são consistentes, e tal pode ser feito seguindo os seguintes passos:

1. **Calcular *λmax*:** pode ser obtido através da seguinte sequência de cálculos [39]:
   - Multiplicar cada prioridade por todos os elementos da coluna correspondente, obtendo assim uma nova matriz;
   - Somar cada linha da matriz anterior, obtendo um novo vetor;
   - Calcular a média da divisão de cada elemento do vetor obtido anteriormente pela prioridade correspondente.

2. **Calcular índice de consistência:** Para calcular o índice de inconsistência pode ser aplicada a fórmula $IC = \frac{\lambda max - n}{n-1}$ [37, 39], em que *n* é o número de elementos da matriz e *λmax* é o valor calculado anteriormente;
3. **Calcular relação de consistência:** O cálculo da relação de consistência é dado pela fórmula: $RC = \frac{IC}{IAM}$ [37, 39], onde *IC* é o índice de consistência previamente calculado e *IAM* é a inconsistência aleatória média, que assume diferentes valores conforme o tamanho da matriz. A relação entre a inconsistência aleatória média e o tamanho da matriz está apresentada na Tabela 9. Caso a relação de consistência seja inferior a 10%, é possível dizer que as opiniões da matriz são consistentes [37, 39].

Tabela 9 - Inconsistência aleatória [37]

Dimensão da matriz	1	2	3	4	5	6	7	8	9	10
Inconsistência aleatória média	0,00	0,00	0,58	0,90	1,12	1,24	1,32	1,41	1,45	1,49

Assim sendo, para calcular *λmax*, começa-se por multiplicar cada prioridade pela respetiva coluna, resultando na seguinte matriz:

$$\begin{bmatrix} 0{,}6333 & 0{,}5308 & 0{,}7815 \\ 0{,}1267 & 0{,}1062 & 0{,}0868 \\ 0{,}2111 & 0{,}3185 & 0{,}2605 \end{bmatrix}$$

De seguida, soma-se cada linha e obtém-se um novo vetor:

$$\begin{bmatrix} 1{,}9456 \\ 0{,}3197 \\ 0{,}7901 \end{bmatrix}$$

Por fim, para obter o *λmax*, é necessário dividir cada do vetor anterior linha pela prioridade correspondente e calcular a média desses resultados (onde $P1 \approx 0{,}6333$; $P1 \approx 0{,}1062$; $P1 \approx 0{,}2605$):

$$\frac{1{,}9456 * P1 + 0{,}3197 * P2 + 0{,}7901 * P3}{3} \approx 3{,}0385$$

Seguidamente, é calculado o índice de consistência. Sendo que *λmax* é igual a 3,0385 e a matriz tem três elementos (ou seja, *n* = 3), podemos calcular índice de consistência:

$$IC = \frac{\lambda max - n}{n - 1} \approx 0{,}0193$$

Obtendo um índice de consistência é possível prosseguir para o cálculo da relação de consistência, que é igual à divisão do índice de consistência pela inconsistência aleatória média. A inconsistência aleatória média assume o valor de "0,58", visto que a matriz de comparação das alternativas tem três elementos. Portanto a relação de consistência é obtida na seguinte expressão:

$$RC \approx \frac{0{,}0192555}{0{,}58} \approx 0{,}0332$$

Ou seja, *RC* é aproximadamente igual a 3,32%. Como *RC* é menor que 10% podemos concluir que as relações da matriz de comparação de prioridades são consistentes, logo é possível proceder à criação das matrizes de comparação entre alternativas face a cada um dos critérios.

A matrizes de comparação dos critérios "Ferramentas de geração de código", "Comunidade" e "Reaproveitamento de código" estão apresentadas na Tabela 10, Tabela 11 e Tabela 12, respetivamente (juntamente com o resultado das suas prioridades, que foram calculadas à semelhança do cálculo das prioridades dos critérios).

Tabela 10 - Matriz de comparação das alternativas face ao critério "Ferramentas de geração de código"

Ferramentas de geração de código	**OpenAPI**	**RAML**	**API Blueprint**	**Prioridade**
OpenAPI	1	7	9	0,7766
RAML	$\frac{1}{7}$	1	3	0,1549
API Blueprint	$\frac{1}{9}$	$\frac{1}{3}$	1	0,0685

Tabela 11 - Matriz de comparação das alternativas face ao critério "Comunidade"

Comunidade	**OpenAPI**	**RAML**	**API Blueprint**	**Prioridade**
OpenAPI	1	7	6	0,7545
RAML	$\frac{1}{7}$	1	$\frac{1}{2}$	0,0919
API Blueprint	$\frac{1}{6}$	2	1	0,1535

Tabela 12 - Matriz de comparação das alternativas face ao critério "Reaproveitamento de código"

Reaproveitamento de código	**OpenAPI**	**RAML**	**API Blueprint**	**Prioridade**
OpenAPI	1	1	5	0,4545
RAML	1	1	5	0,4545
API Blueprint	$\frac{1}{5}$	$\frac{1}{5}$	1	0,0909

Após definir as matrizes de comparação das alternativas e calcular a prioridade de cada uma delas é necessário verificar a consistência das opiniões, seguindo os passos inerentes ao cálculo da relação de consistência como foi demonstrado para a matriz de comparação de critérios. Na Tabela 13 é apresentado o índice de consistência e a relação de consistência das opiniões para cada um dos critérios julgados nas matrizes de comparação de alternativas. Podemos então concluir que os julgamentos avaliados nas matrizes de comparação de alternativas são consistentes, sendo que todos eles são inferiores a 10%.

Tabela 13 - Consistência das matrizes de comparação de alternativas

Critério da matriz de comparação	**Índice de consistência**	**Relação de consistência**
Ferramentas de geração de código	0,0401	6,92%
Comunidade	0,0162	2,79%
Reaproveitamento de código	0	0

Finalmente, para chegar à prioridade final de cada uma das alternativas, é necessário construir uma matriz a partir das prioridades calculadas em cada uma das matrizes de comparação de alternativas e multiplicar essa matriz pelo vetor de prioridades da matriz de critérios [38]. Esse cálculo está demonstrado na seguinte expressão (a primeira linha corresponde às prioridades da alternativa *OpenAPI*, a segunda corresponde às prioridades da alternativa *RAML* e a terceira às prioridades da alternativa *API Blueprint*):

$$\begin{bmatrix} 0{,}78 & 0{,}75 & 0{,}45 \\ 0{,}15 & 0{,}09 & 0{,}45 \\ 0{,}07 & 0{,}15 & 0{,}09 \end{bmatrix} * \begin{bmatrix} 0{,}63 \\ 0{,}11 \\ 0{,}26 \end{bmatrix} \approx \begin{bmatrix} 0{,}69 \\ 0{,}23 \\ 0{,}08 \end{bmatrix}$$

Assim, é possível concluir que a alternativa mais adequada para especificar a nova *API* do E-goi é a especificação *OpenAPI*, com uma prioridade de aproximadamente 69%, contra uma prioridade de cerca de 23% da especificação *RAML* e 8% da especificação *API Blueprint*.

É possível ainda calcular as prioridades, dividindo cada prioridade pela prioridade mais elevada [38]. Por exemplo, para a matriz anterior, as prioridades idealizadas são as seguintes:

$$\begin{bmatrix} 1 \\ 0{,}33 \\ 0{,}12 \end{bmatrix}$$

A partir das prioridades idealizadas podemos apresentar o valor em proporção de uma alternativa relativamente a outra [38]. Interpretando os resultados apresentadas na matriz anterior, podemos dizer que a escolha da especificação *RAML* é 33% tão boa quanto a especificação *OpenAPI* (prioridade idealizada de 100%) e, seguindo o mesmo raciocínio, a *API Blueprint* é 12% tão boa quanto a *OpenAPI*.

4 Análise

Neste capítulo é descrita a análise de requisitos e de negócio efetuada.

4.1 Análise de Requisitos

A análise dos requisitos (funcionais e não funcionais) implica o estudo das necessidades dos utilizadores, de modo a encontrar uma definição correta e completa de um produto de *software* [40].

A fase de análise de requisitos tem como objetivo compreender e documentar as necessidades das partes interessadas, sendo um fator determinante para o sucesso ou insucesso da solução.

4.1.1 Requisitos Funcionais

Este trabalho tem apenas um ator, que é o cliente do E-goi. Os requisitos funcionais consistem nas funcionalidades que este ator pode despoletar. O cliente do E-goi pode utilizar todas as funcionalidades da *API* através do *SDK*; assim sendo, os requisitos funcionais são os mesmos da *API* pública, visto que o *SDK* é apenas uma abstração para esta. A Tabela 14 apresenta os requisitos funcionais mencionados, descritos em histórias de usuário [41].

Tabela 14 - Requisitos funcionais

Identificação	Descrição
US1	Eu, como cliente, quero criar, visualizar, editar ou remover uma lista de contactos.
US2	Eu, como cliente, quero listar todas as listas de contactos.
US3	Eu, como cliente, quero criar, visualizar, editar ou remover um segmento de uma lista.
US4	Eu, como cliente, quero listar todos os segmentos de uma lista.
US5	Eu, como cliente, quero criar, visualizar, editar ou remover um contacto de uma lista.
US6	Eu, como cliente, quero listar todos os contactos de uma lista.
US7	Eu, como cliente, quero importar um conjunto de contactos para uma lista a partir de um ficheiro.
US8	Eu, como cliente, quero criar, visualizar, editar ou remover um campo de uma lista.
US9	Eu, como cliente, quero listar todos os campos de uma lista.
US10	Eu, como cliente, quero criar, visualizar, editar ou remover uma etiqueta.
US11	Eu, como cliente, quero associar uma etiqueta a um contacto.
US12	Eu, como cliente, quero listar todas as etiquetas.
US13	Eu, como cliente, quero criar, visualizar, editar ou remover um remetente.
US14	Eu, como cliente, quero listar todos os remetentes.
US15	Eu, como cliente, quero criar, visualizar, editar ou remover uma campanha.
US16	Eu, como cliente, quero enviar uma campanha para uma lista ou segmento.
US17	Eu, como cliente, quero visualizar o relatório de uma campanha.

US18	Eu, como cliente, quero listar todos os relatórios de campanhas.
US19	Eu, como cliente, quero criar, visualizar, editar ou remover um utilizador.
US20	Eu, como cliente, quero listar todos os utilizadores da conta.
US21	Eu, como cliente, quero criar, visualizar, editar ou remover um formulário.
US22	Eu, como cliente, quero listar todos os formulários.
US23	Eu, como cliente, quero criar ou visualizar uma submissão de um formulário.
US24	Eu, como cliente, quero listar todas as submissões de formulários.

De forma a aumentar a qualidade de uma história de usuário, é possível a inclusão de critérios de aceitação, que definem ações específicas que modificam o ambiente da operação. Para o caso das histórias de usuário definidas na Tabela 14, existem diversos critérios de aceitação, sendo muitos destes idênticos. Para efetuar qualquer um dos pedidos é necessário:

- O cliente estar autenticado;
- O cliente ter permissões para efetuar a ação;
- O cliente aceitar respostas no formato *JSON*.

Em certos casos, como nas histórias de usuário que implicam criação ou edição de dados, o cliente deve ainda enviar o corpo da sua mensagem no formato *JSON*. Adicionalmente, para histórias de usuário que implicam a existência de uma entidade (visualizações, edições e remoções), é necessário que essa mesmo entidade exista no sistema.

4.1.2 Outros Requisitos

Nesta secção são capturados os requisitos do sistema, que não foram identificados na secção 4.1.1, aquando da definição dos requisitos funcionais. Para a definição dos requisitos não funcionais, foi utilizado o modelo *Functionality, Usability, Reliability, Performance and Supportability Plus (FURPS+)*, onde não foram identificadas restrições de confiança e desempenho:

- **Funcionalidade:**

- Sempre que um pedido à *API* do E-goi é efetuado, o sistema deve registar a informação desse pedido.

- **Usabilidade:**
  - A documentação criada deve ser fácil de entender por parte do utilizador da *API*.

- **Suportabilidade:**
  - A documentação, os *SDK* e os testes devem ser gerados de forma dinâmica, com base na especificação da *API*.

- **Outros (+):**
  - **Limitações de *design*:**
    - Utilização de uma tecnologia para especificação da *API REST* para geração de documentação, *SDK* e testes.
  - **Limitações de implementação:**
    - Utilização de *Zend Framework* para implementação do sistema de geração de testes e do sistema registos da utilização da *API*;
    - O *SDK* gerado deve estar disponível na linguagem *PHP*.

4.2 Análise de Negócio

Na presente secção é apresentada a análise do negócio, onde é descrito o modelo de domínio visa representar as entidades conceptuais do domínio, os seus atributos e as suas relações [42].

Na Figura 8 estão representados os principais conceitos do domínio da plataforma E-goi e da sua *API*. Como os *SDK* disponibilizam funcionalidades existentes numa *API*, o modelo de domínio é igual tanto para os *SDK* como para a *API*.

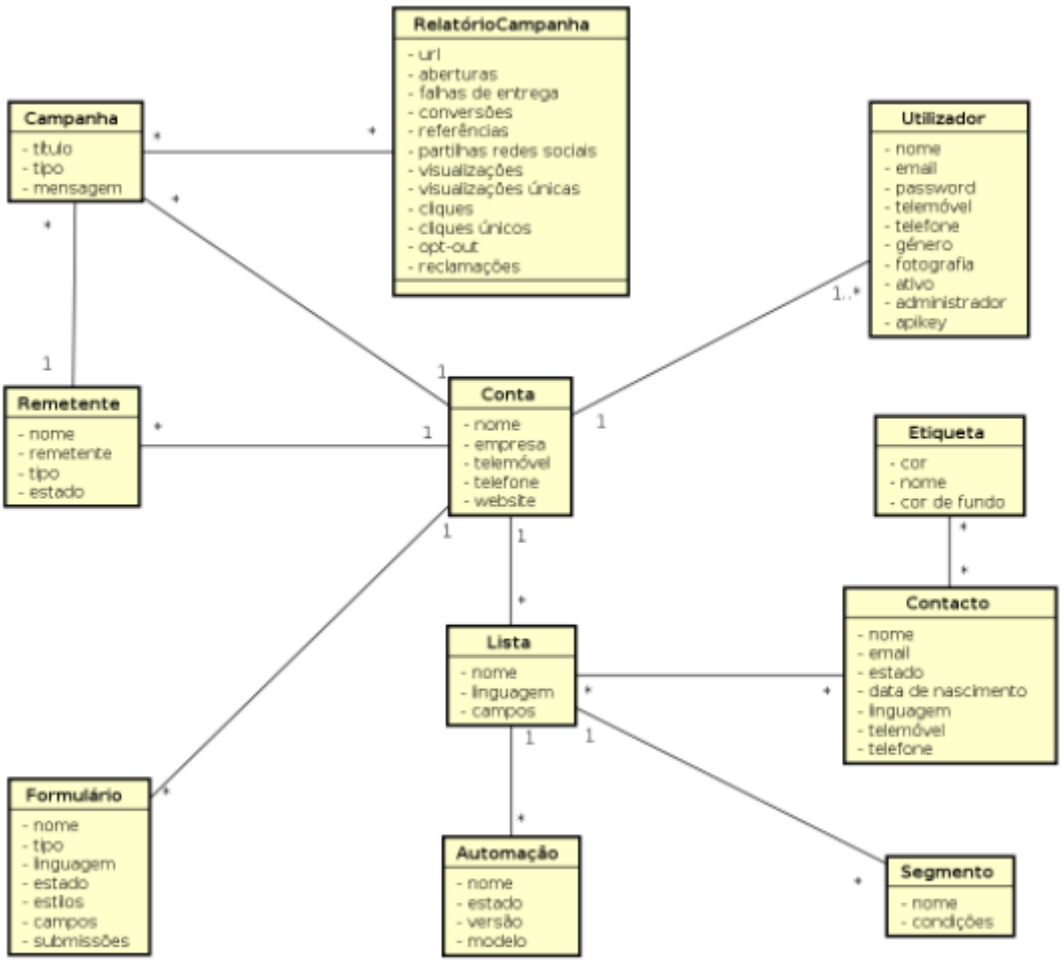

Figura 8 - Modelo de domínio

Com o glossário de domínio pretende-se explicar cada uma das entidades de domínio e também alguns dos seus atributos. Na Tabela 15 são descritas as entidades de negócio e os atributos mais relevantes das mesmas para o domínio do problema.

Tabela 15 - Glossário do domínio

Entidade	**Descrição**
Automação	Uma automação pode encontrar-se no estado “ativo” ou “inativo” e possui um atributo “versão” que indica a versão do editor de automações através do qual a automação foi criada. O atributo “modelo” contém o fluxo que a automação deve seguir.
Campanha	As campanhas podem ser de “*email*”, “fax”, “voz” ou “*SMS*”. O atributo “mensagem” varia conforme o tipo de campanha.
Conta	Conta criada na plataforma E-goi.
Contacto	Contacto pertencente a uma lista. Pode encontrar-se no estado “ativo”, “inativo”, “removido”, “não confirmado” e “à espera de nova confirmação”.
Etiqueta	Etiquetas que podem ser associadas a contactos e utilizadas para filtragem.
Formulário	Formulários que podem estar no estado “ativo”, “inativo” e “não publicado”.
Lista	Lista de contactos de uma conta. Pode possuir campos referentes aos contactos da lista (por exemplo, campo nome ou campo telefone).
Relatório de Campanha	Relatórios de campanhas baseados em diversas métricas. Possuem um *URL* público a partir do qual podem ser acedidos. O atributo *opt-out* é o total de contactos que terminaram a sua subscrição após o envio dessa campanha.
Remetente	Remetente das campanhas. Existe um tipo diferente de remetente para cada tipo de campanha existente (por exemplo para campanhas de *email* o remetente contém um *email* e para campanhas de *SMS* contêm um número de telefone ou telemóvel).
Segmento	Segmento de contactos de uma lista. Possuem variadas condições para auxiliar na filtragem de contactos de uma determinada lista (por exemplo, selecionar todos os contactos nascidos após 1980).
Utilizador	Utilizadores da conta. O atributo “administrador” indica se ele é administrador da conta.

5 Design Arquitetural

Neste capítulo é descrita a arquitetura da solução desenvolvida, bem como as alternativas arquiteturais encontradas, considerando os requisitos anteriormente descritos.

5.1 Vista Lógica

A vista lógica tem como objetivo ilustrar e detalhar os diferentes componentes do sistema.

A Figura 9 ilustra o sistema como um todo. Este sistema centra-se no suporte à evolução da *API* pública. Relativamente à *API* pública (*Proxy* entre as *API* internas do E-goi e as aplicações dos clientes), apenas parte das suas componentes foram desenvolvidas no âmbito do projeto, estando estas identificadas na Figura 9. Sempre que uma mudança evolutiva ocorre na *API*, esta reflete-se na alteração da especificação *OpenAPI*, o que implica uma alteração tanto na documentação, como nos *SDK*, como nos testes gerados através da especificação. Sendo que a geração de registos da utilização da *API* é essencial para determinar quando e como uma *API* deve evoluir [1], a vista lógica presente na Figura 9 representa também o funcionamento do sistema de registos da utilização da *API*.

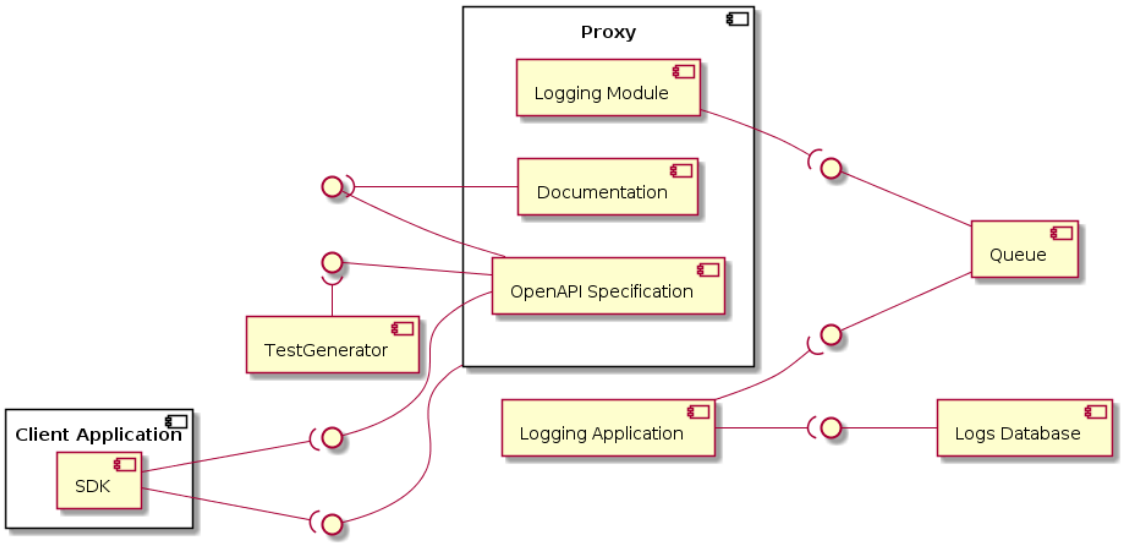

Figura 9 - Vista lógica das componentes desenvolvidas

Para gerar a documentação de forma dinâmica e com base na *OAS* foi utilizada a ferramenta *Swagger UI*. De forma semelhante, foi utilizada a ferramenta *Swagger Codegen* para a geração de *SDK*. Por essa razão, apenas as soluções arquiteturais referentes ao sistema de geração de testes e ao sistema de registos são apresentadas (nas secções 5.1.1 e 5.1.2, respetivamente).

5.1.1 Testes

Ao utilizar uma especificação de *API* é fundamental assegurar que a documentação e os *SDK* fornecidos aos clientes depois de cada mudança evolutiva na *API* são corretamente implementados [7]. Para, podem ser utilizados testes de aceitação, verificando assim se o contrato definido está corretamente implementado na *API*. Testes de aceitação são testes de *software* que validam um critério de aceitação definido por um cliente, testando assim se a funcionalidade desejada produziu o resultado esperado com sucesso [43]. Neste caso, os testes de aceitação são as respostas que os clientes esperam obter ao utilizar a *API*, em cada um dos casos de uso definidos na secção 4.1.1.

Ao desenvolver ou evoluir sistemas, os requisitos mudam frequentemente, e, em muitos casos, estas alteração afetam componentes relacionados com documentação [44]. As alterações que os clientes esperam são definidas na especificação da *API*, sendo que esta é utilizada para gerar a documentação. Assim sendo é necessário testar a *API* contra estes testes de aceitação, criados a partir da especificação.

Tal como a documentação e o *SDK*, estes devem ser gerado de forma dinâmica utilizando a componente *OpenAPI Specification*. Existem diversas alternativas para a resolução deste problema. É possível gerar os testes diretamente para a *API* pública ou isolar os testes na aplicação responsável pela sua geração.

Como é possível verificar a partir da Figura 10, a geração de testes pode ser executada a partir de uma rota de consola, que lê a especificação *OpenAPI* existente no *Proxy* e a envia para o controlador, que, por sua vez, instancia um objeto que contêm a informação existente na especificação. De seguida, o controlador invoca o gerador de testes, injetando a instância de *Specification* como dependência. Assim, os testes são gerados na componente *TestGenerator* e são executados através da ferramenta de testes existente nesta aplicação.

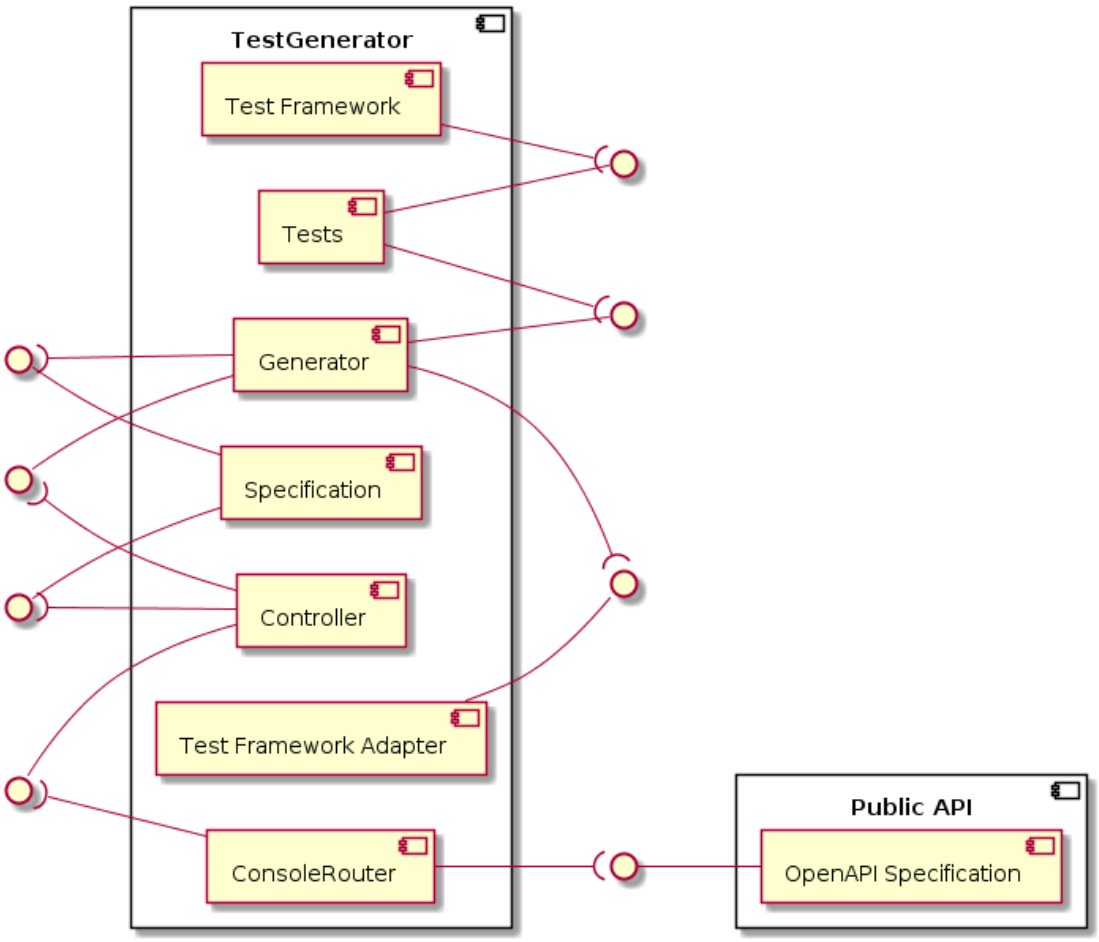

Figura 10 - Vista lógica da componente de geração de testes

Alternativamente, os testes podem ser gerados diretamente na *API* pública, como ilustra a Figura 11. A arquitetura desta alternativa de solução é semelhante à da solução apresentada na Figura 10, porém, como os testes poderão ser gerados para diferentes ferramentas de testes, é

necessária a utilização do padrão *Adapter* [45]. A componente *Test Framework Adapter* é responsável por transformar os dados contidos na especificação em testes específicos para a ferramenta de testes utilizada.

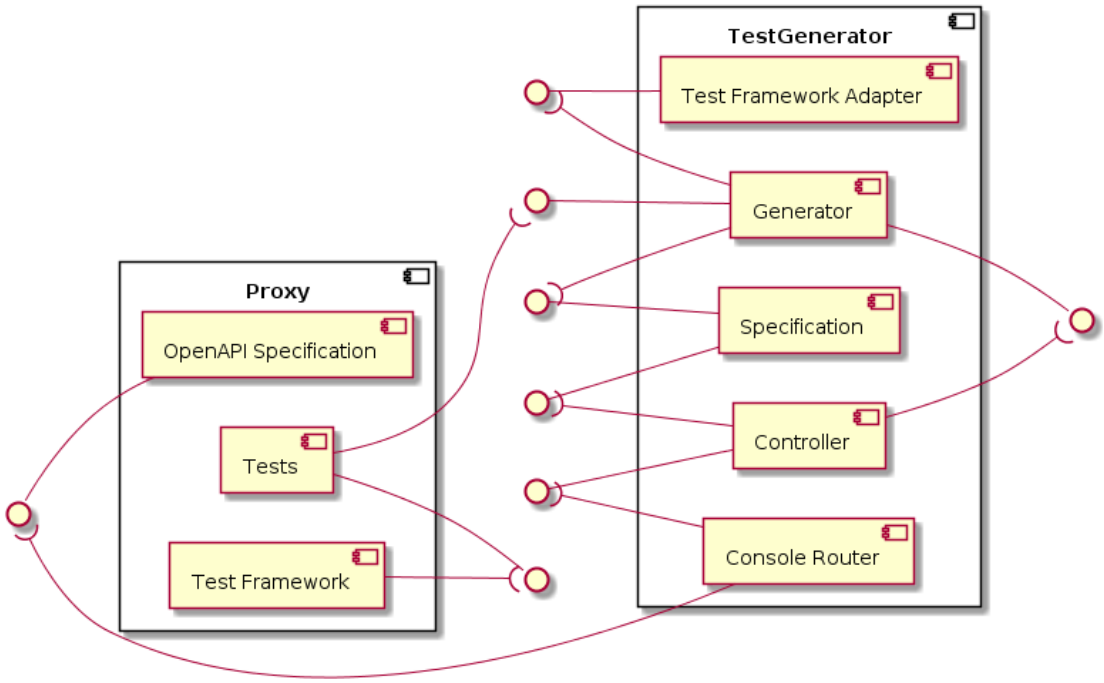

Figura 11 - Vista lógica da alternativa de solução da componente de geração de testes

Ambas as abordagens permitem a geração de testes para diferentes *API*, desde que essas *API* possuam uma especificação válida. Gerar os testes diretamente na *API* permite facilmente estender, reescrever e manipular esses testes. Contudo, para reaproveitar este sistema em múltiplas *API* é necessário desenvolver adaptadores para cada ferramenta de testes utilizada, visto que diferentes *API* podem utilizar diferentes ferramentas de testes. Por outro lado, ao gerar os testes na aplicação responsável por essa geração é necessário apenas que esses testes estejam de acordo com uma ferramenta de testes, visto que esta está incorporada no gerador de testes não depende de qualquer outra ferramenta de testes existente na *API* a testar. Consequentemente, conclui-se que a primeira abordagem apresentada é a solução mais dinâmica e mais adequada para a resolução dos problemas apresentados.

5.1.2 Sistema de Registos

O sistema de registos que se pretende desenvolver visa registar os pedidos efetuados pelos clientes à *API* pública do E-goi. Esta aplicação necessita de ser desenvolvida para a *API* pública,

mas é desejável que esta possa ser reaproveitada para outras *API*. Para isso é necessário desenvolver em cada *API* que utilize este sistema uma componente que seja capaz de enviar os dados para uma fila, dados esses que mais tarde serão processados pelo sistema de registos.

Como é possível verificar a partir da Figura 12, a componente *Logging Module* possui a componente *Listener* que observa a *API* e invoca o controlador sempre que um pedido é recebido na *API.* O *Logging Module* tem como objetivo enviar os dados a serem registados para um sistema de filas, através da componente *Producer*. O objetivo deste sistema de filas *(Queue)* é permitir a criação de registos, de forma assíncrona, sem que este processamento influencie o tempo de resposta dos serviços. Assim, sempre que um pedido é executado, os dados relativos a esse pedido, presentes no modelo, são enviados pelo *Producer* para a *Queue*, ficando à espera que uma ou mais instâncias de *Consumer* retirem esses dados da *Queue*.

O sistema de registos da utilização da *API* possui a componente *Consumer*, que lê a informação colocada na *Queue* e retorna-a para o controlador, que por sua vez transforma os dados num *Model* e invoca o repositório que regista os dados do modelo na base de dados *Logs Database*.

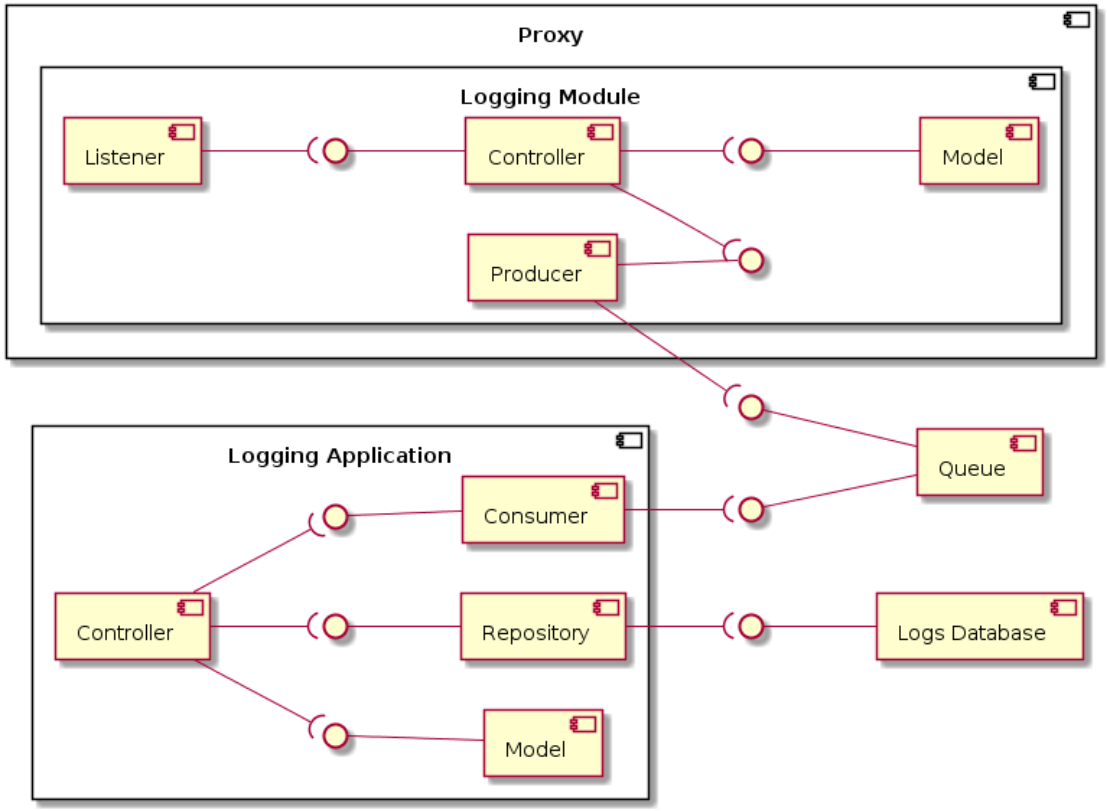

Figura 12 - Vista lógica da da componente dos registos

Relativamente ao sistema de registos da utilização da *API* existe uma alternativa arquitetural que é relevante considerar. Na solução apresentada na Figura 12, o *Model* da *Logging Application* é igual ao *Model* presente no módulo de *logging* da *API* pública, o que implica a alteração de ambos os modelos sempre que surja uma alteração. Este modelo pode ser reaproveitado caso os componentes presentes na *Logging Application* sejam movidos para o *Logging Module*, como é ilustrado na Figura 13.

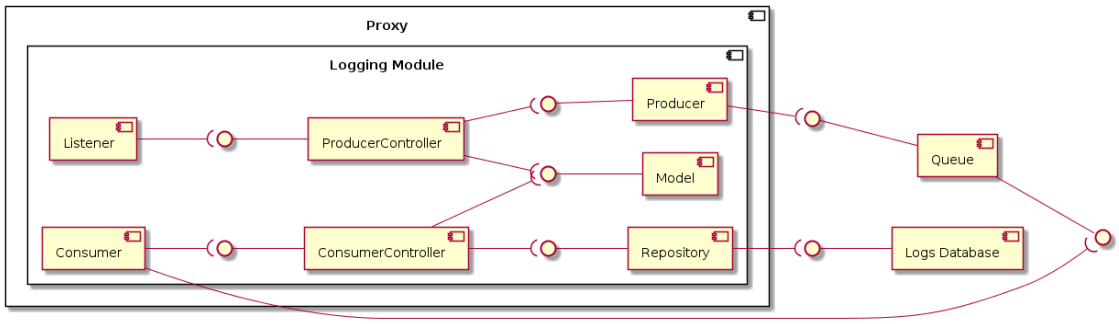

Figura 13 - Vista lógica da alternativa de solução da componente dos registos

Seguindo esta alternativa arquitetural, só seria necessário alterar uma vez o *Model* sempre que este necessitar de alterações. No entanto esta alternativa não permite, ao contrário da primeira abordagem descrita, a criação de múltiplas instâncias de *Logging Application*, que pode ser útil sobretudo se existirem muitos dados na *Queue* (ou seja a produção de dados está a ser efetuada a um ritmo mais elevado que o consumo) e seja necessário aumentar o número de consumidores da *Queue*. Esta alternativa também não permite que esta solução possa ser reaproveitada para criar registos para diferentes *API*.

Como a empresa E-goi espera uma grande quantidade de pedidos à sua *API* e consequentemente muitos registos a serem guardados, conclui-se que a primeira alternativa de solução é a mais adequada para resolver o problema.

5.2 Vista de Implantação

A vista de implantação tem como objetivo mostrar a implantação física dos diferentes sistemas desenvolvidos. A *API* pública, que foi especificada no âmbito desta dissertação, encontra-se isolada de todas as outras aplicações desenvolvidas, com exceção da sua documentação, como é visível na Figura 14.

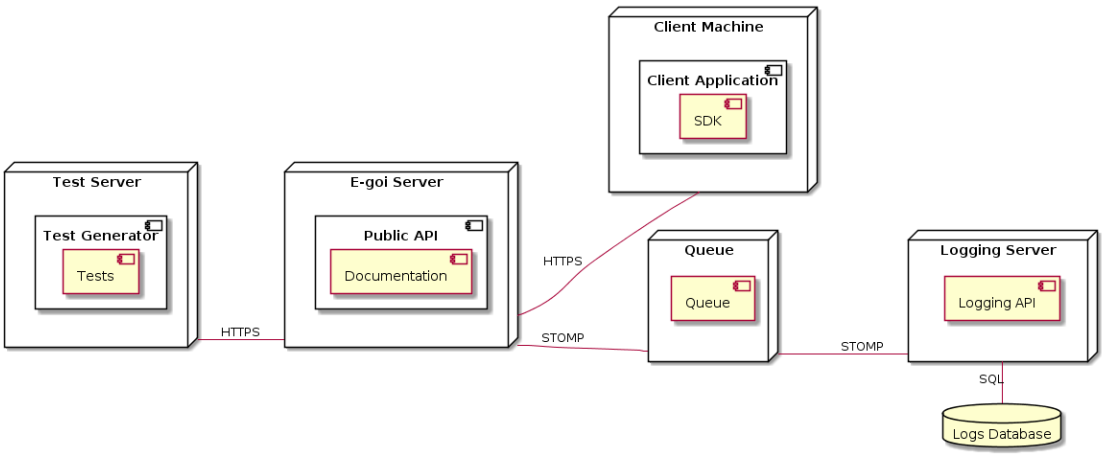

Figura 14 - Vista de implantação

A partir da Figura 14 é possível ainda verificar que os *SDK* são aplicações que existem dentro das aplicações dos clientes e são responsáveis pela comunicação com a *API*. O sistema de geração de testes também se encontra num servidor à parte da *API*, e, devido à alternativa de solução adotada, os testes são gerados no servidor onde este sistema se encontra. Ou seja, de forma a testar a aplicação, estes testes comunicam com a *API* pública através da *web,* a partir da qual podem obter o a especificação de forma a gerar os testes e os resultados dos serviços presentes na especificação e implementados pela *API*.

Adicionalmente, sempre que um pedido é recebido na *API* pública, este é enviado para uma fila *Queue*). Devido à grande afluência de dados, essa fila encontra-se numa estrutura isolada que atua como ponte de ligação entre a *API* pública e o sistema de registos. Assim, o sistema de registos interpreta os dados presentes na fila e regista-os numa base de dados.

6 Design e Implementação

No presente capítulo é apresentado o *design* das soluções propostas, bem como alguns detalhes de implementação relevantes. Encontra-se dividido na especificação da *API*, que por sua vez está subdividida nas componentes que são geradas a partir da especificação e no sistema de registos de utilização da *API*.

6.1 Especificação da API

A especificação de uma *API* estabelece um contrato que deve ser implementado pela *API*. Consequentemente, o desenho da *API*, incluindo os seus recursos, operações e representações são todos definidos a partir da especificação, o que significa que este deve ser um processo bastante cauteloso, especialmente em *API* públicas, que são destinadas aos clientes.

Nesta secção são apresentados e explicados excertos da especificação *OpenAPI*, focados na definição de uma operação para obter as listas de contactos de uma conta. A estrutura de dados do recurso referente à operação para obter listas de contactos não se encontra descrita na totalidade, devido à sua extensão. Pela mesma razão, não são descritos *links* de hipermédia. O conjunto formado pelos excertos apresentados na presente secção é utilizado como exemplo para geração de artefactos como documentação, *SDK* e testes de *software*, descritos nas secções 6.1.1, 6.1.2 e 6.1.3, respetivamente.

Inicialmente, é necessário indicar a versão da *OAS* e é possível descrever a informação básica da *API*, incluindo o seu título, descrição, termos do serviço, contactos, licença e versão atual, como mostra o Código 8, escrito em *YAML*.

```yaml
openapi: 3.0.1
info:
  title: Public API
  description: New E-goi's public API
  termsOfService: https://e-goi.com
  contact:
    name: (+351) 300 404 336
  license:
    name: E-goi
  version: 1.0.0
```

Código 8 – Especificação *OpenAPI* da informação básica da *API*

Seguidamente, a *OAS* permite a definição de recursos e operações. Essa definição contém um conjunto de etiquetas que podem ser utilizadas para agrupar operações, um resumo, a descrição, um identificador único, os seus parâmetros e as suas respostas. Como alguns parâmetros e outros esquemas podem ser reutilizados, é uma boa prática comum defini-los fora do âmbito da operação e invoca-los com a palavra-chave *ref*. As respostas contêm um código de estado, a descrição e o seu conteúdo, que se pode encontrar em diferentes formatos. Novamente, é possível referenciar na resposta um esquema de dados definido num âmbito global. O Código 9 mostra uma versão sumarizada da operação para obter todas as listas de contactos existentes numa conta E-goi.

```yaml
paths:
  /lists:
    get:
      tags:
        - Lists
      summary: Get all lists
      description: Returns all lists
      operationId: GetAllLists
      parameters:
        - $ref: '#/components/parameters/limit'
      responses:
        '200':
          description: OK
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: '#/components/schemas/list'
```

Código 9 - Especificação *OpenAPI* de uma operação para obter listas de contactos

Para cada parâmetro deve ser indicado o nome, a localização onde esse parâmetro deve ser enviado e o seu esquema de dados, que contém as restrições do parâmetro, como é possível verificar no Código 10. Também pode ser incluída uma descrição e a palavra-chave *required*, que assume o valor booleano falso por defeito.

```
parameters:
  limit:
    name: limit
    in: query
    description: Number of items to return
    required: false
    schema:
      type: integer
      minimum: 1
      example: 20
```

Código 10 - Especificação *OpenAPI* de um parâmetro

Um esquema pode ser definido e reutilizado não só dentro dos parâmetros, mas também dentro das respostas. Por exemplo, uma resposta pode conter um conjunto de propriedades onde é possível, para cada um, indicar restrições e informação como descrição, tipo, exemplo e enumerado. O Código 11 mostra um esquema de resposta onde a lista contém um identificador, um título e um estado que se pode encontrar no estado ativo, inativo ou bloqueado.

```
schemas:
  list:
    description: Success response schema for this operation
    properties:
      id:
        description: Id of the list
        type: integer
        example: 1
      title:
        description: Title of the list
        type: string
        example: Title
      status:
        description: Status of the list
        type: string
        enum:
          - active
          - inactive
          - blocked
        example: active
```

Código 11 - Especificação *OpenAPI* de um esquema de dados

Uma descrição *OpenAPI* também pode conter diferentes tipos de autenticação. É possível definir diferentes esquemas de segurança, à semelhança dos parâmetros e esquemas de dados. Estes esquemas de segurança podem ser reutilizados dentro das operações através da palavra-chave *security*, tal como pode ser invocada num escopo global, como é possível visualizar no Código 12.

```
security:
  - Apikey: []
components:
  parameters:
    …
  schemas:
    …
  securitySchemes:
    Apikey:
      type: apiKey
      name: Apikey
      in: header
```

Código 12 - Especificação *OpenAPI* de um esquema de autenticação

6.1.1 Documentação

Uma das grandes capacidades da *OpenAPI* é a geração automática de documentação. Uma documentação gerada de forma dinâmica pode ser implementada de raiz, no entanto isso é desnecessário, sendo que *Swagger* fornece a ferramenta *Swagger UI*. Com esta ferramenta, utilizando a especificação desenvolvida na secção 6.1, é possível gerar uma página de documentação interativa, ilustrada na Figura 15.

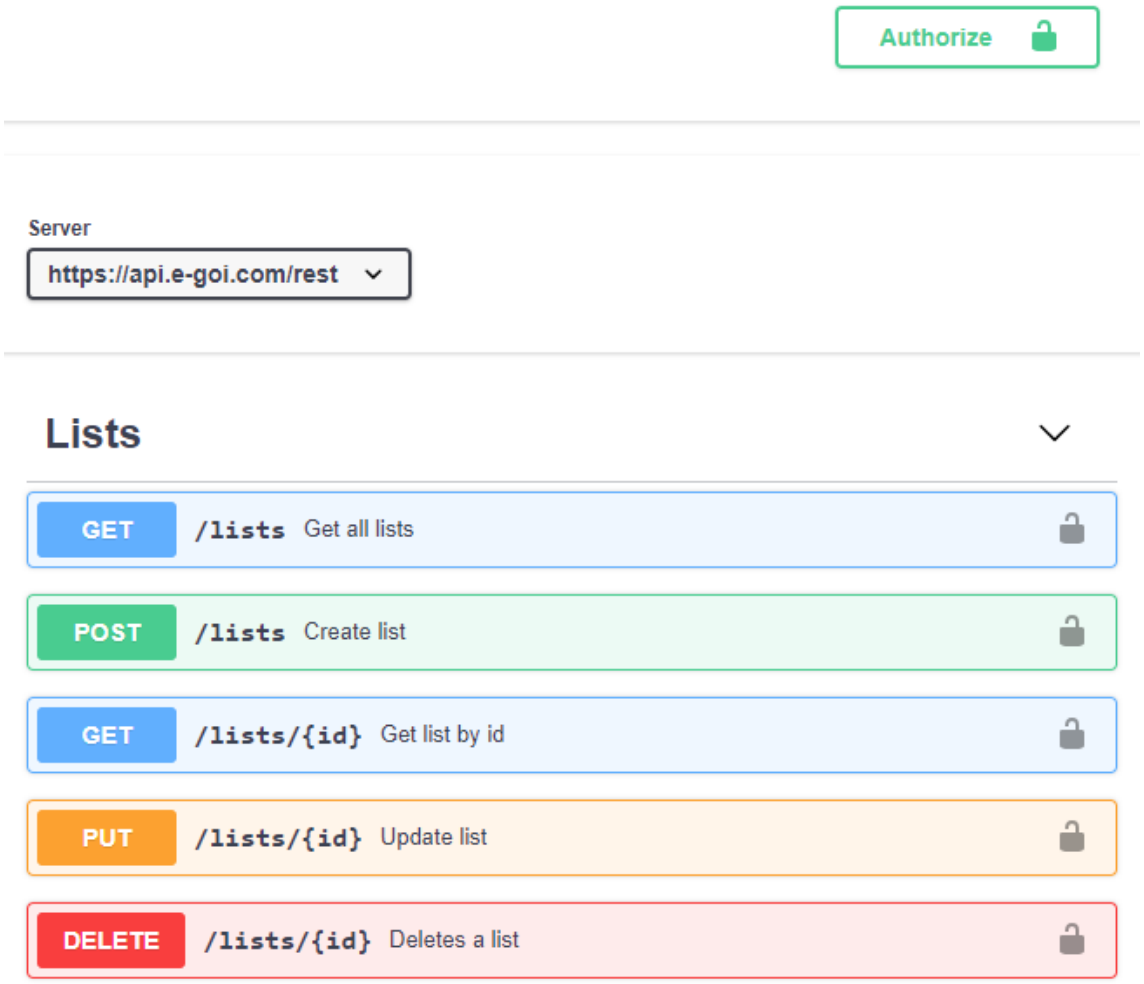

Figura 15 - Página de documentação criada com *Swagger UI*

Como demonstrado na Figura 16, é possível inserir os parâmetros e executar o pedido.

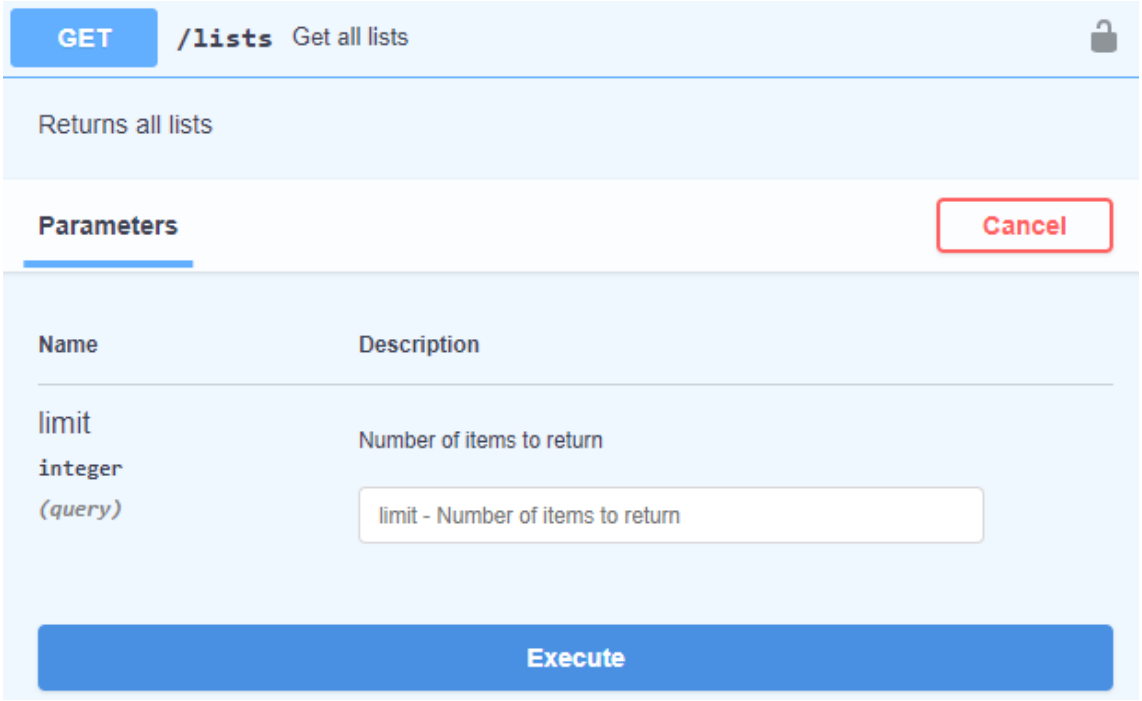

Figura 16 - Documentação gerada de uma operação para obter listas de contactos

Seguidamente o servidor devolve a resposta, ilustrada na Figura 17, que deve ser sempre coerente com a resposta definida na especificação.

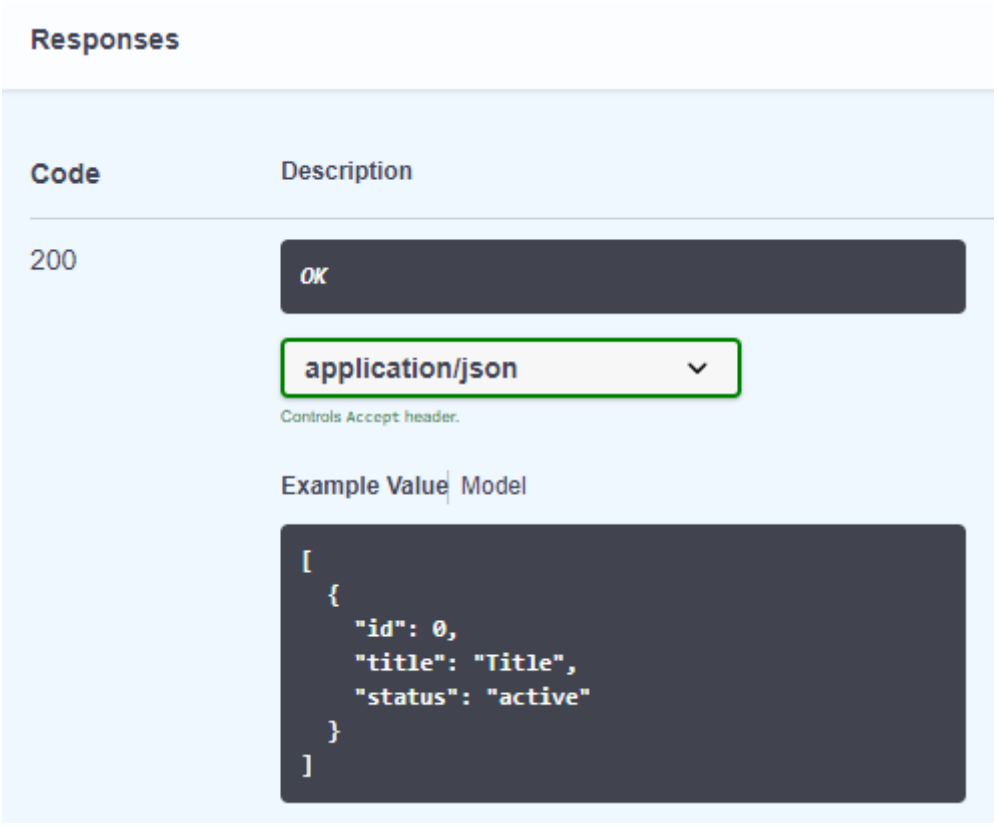

Figura 17 - Documentação da resposta de uma operação para obter listas de contactos

Apesar de ser desejável que a resposta do servidor esteja de acordo com a resposta especificada, *Swagger UI* apresenta a resposta do servidor mesmo que esta não apresente coerência com a resposta especificada. Por essa razão, é essencial a existência de testes de aceitação, que estão detalhados na secção 6.1.3.

Também é possível para quem consulta a documentação ver os esquemas de dados definidos na especificação *OpenAPI* e os seus detalhes, como apresentado na Figura 18.

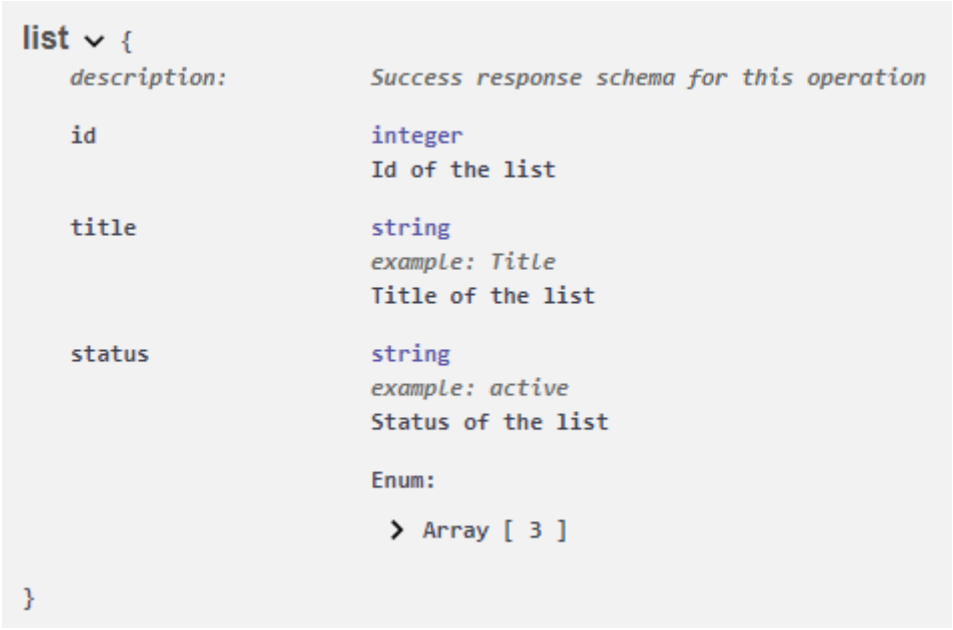

Figura 18 - Documentação do esquema de dados de uma lista de contactos

6.1.2 SDK

Algumas *API* pública, como a *API* do E-goi, oferecem *SDK* aos seus clientes para que estes possam facilmente integrar os seus projetos com a *API*, sem terem de se preocupar com lógica relacionada com implementação *web*.

Swagger oferece uma ferramenta, denominada *Swagger Code Generator*, ou simplesmente *Swagger Codegen*, que é capaz de gerar bibliotecas de código cliente em várias linguagens, através de uma especificação de *API*. Contudo, *Swagger Codegen* ainda não possui uma versão estável que suporte a versão 3.0 da *OAS* [23]. Apesar de a última versão estável do *Swagger Codegen* ser a versão 2, a versão 3 está atualmente a ser desenvolvida. Ainda que a geração de código cliente pudesse ser implementada manualmente, *Swagger Codegen* é uma ferramenta bastante útil, capaz de gerar código para várias linguagens e ferramentas. Portanto, de forma a gerar *SDK* com esta ferramenta foi necessário converter a especificação para a versão 2.

A aplicação de consola *Swagger Codegen* aceita diferentes parâmetros de forma a personalizar a geração de código. Alguns desses parâmetros são:

- **-a <autorização> ou --auth <autorização>:** *Headers* de autorização para obter dados de uma *API* através do *Swagger Codegen*;
- **-c <ficheiro de configuração> ou --config <ficheiro de configuração>:** Configurações adicionais que variam de acordo com a linguagem de programação. Pode ser usada, por exemplo, para selecionar um cliente *HTTP* específico;
- **-i <ficheiro da especificação> ou --input-spec <ficheiro da especificação>:** Localização do ficheiro que contém a especificação *OpenAPI*;
- **-l <linguagem> ou --lang <linguagem>:** Linguagem ou ferramenta para a qual é gerado o código;
- **-o <diretório> ou --output <diretório>:** Localização onde é gerado o código;
- **-s ou --skip-overwrite:** Utilizado quando não é desejável que os ficheiros sejam reescritos.

Através do comando *java -jar swagger-codegen-cli.jar generate -i https://api.e-goi.com/rest/openapi -l php -o /path/to/file* é gerado um *SDK* em *PHP* para a *API* do E-goi. Além do código cliente gerado, *Swagger Codegen* gera também ficheiros de documentação acerca de como usar a biblioteca. Um bom exemplo de código deve ser executado com o mínimo de esforço [46]; assim sendo, bons exemplos de código são frequentemente concisos, explicados passo a passo, com comentários e hiperligações para recursos externos quando necessário [47].

Os exemplos de código gerados pelo *Swagger Codegen* cumprem estes requisitos. Um ficheiro *readme* é gerado, contendo os pré-requisitos e passos necessários para a instalação do *SDK* na aplicação cliente. Adicionalmente, este ficheiro contém informações sobre autenticação e autorização e hiperligações para os recursos existentes. Para cada operação são também gerados ficheiros que descrevem a operação, incluindo os parâmetros e esquemas de dados definidos na especificação.

O Código 13, organizado em blocos lógicos com comentários, mostra um exemplo de utilização do *SDK*, onde o cliente necessita apenas de inserir a sua chave de *API*, instanciar um cliente e invocar um método existente (passando parâmetros se necessário).

```php
<?php
require_once(__DIR__ . '/vendor/autoload.php');

// Configure API key authorization: Apikey
$config = Swagger\Client\Configuration::getDefaultConfiguration()
   ->setApiKey('Apikey', 'YOUR_API_KEY');

$apiInstance = new Swagger\Client\Api\ListsApi(
   new GuzzleHttp\Client(),
   $config
);
$limit = 56; // int | Number of items to return

try {
   $result = $apiInstance->getAllLists($limit);
   print_r($result);
} catch (Exception $e) {
   echo 'Exception when calling ListsApi->getAllLists: ', $e->getMessage(),     PHP_EOL;
}
?>
```

Código 13 - Código cliente gerado pelo *Swagger Code Generator*

6.1.3 Testes

Para gerar testes, uma especificação com uma sintaxe válida não é suficiente, visto que alguns métodos necessitam de parâmetros. Uma especificação completa inclui frequentemente exemplos de parâmetros válidos, que podem ser usados como parâmetros nos testes. Contudo, uma especificação é válida mesmo que não sejam fornecidos quaisquer exemplos. Ainda assim, estes testes podem ser gerados com base nas restrições de cada parâmetro. Estas restrições incluem o tipo de dados, valores máximos e mínimos, enumerados e expressões regulares. Assim, é possível gerar dados que cumpram com as restrições definidas, criando parâmetros válidos para executar operações e testar o seu resultado contra a especificação.

Porém, para evitar faltas de coesão nos testes e aumentar a sua confiança, é importante garantir que não existe aleatoriedade nem outra fonte de não-determinismo [48]. Portanto, ao gerar dados a partir de restrições, é importante utilizar valores fixos. Assim, para gerar um teste de para uma operação, é utilizado o primeiro valor que satisfaz as restrições de um parâmetro.

Contudo, para automatizar completamente o processo de geração de testes de aceitação é necessário cumprir os requisitos de cada operação a ser testada. Por exemplo, não é possível gerar um teste de uma operação que remove um recurso sem que esse recurso seja previamente

criado. Portanto, antes de apagar uma lista de contactos específica, é necessário criar essa lista. Adicionalmente, operações que criam recursos frequentemente necessitam de uma operação para visualizar os dados da entidade criada.

Para resolver este problema, além da especificação, deve ser fornecido um ficheiro de configuração, onde as operações existentes estão listadas e associadas com as dependências de cada uma. Um exemplo desse ficheiro, no formato *JSON*, está presente no Código 14. Como é possível verificar, o ficheiro de configuração contém os recursos presentes na especificação, bem como as suas dependências, que podem ser referências para outras operações identificadas pelo identificador único de cada operação na especificação.

```
{
  "/lists/{id}": {
    "specificationDependencies": [
      "CreateList"
    ]
  }
}
```

Código 14 - Ficheiro de configuração para a geração de testes

No entanto, como “testes não devem ditar o código” [48], as dependências mencionadas não devem ser forçadas a existir em código de produção e, consequentemente, na sua especificação. Logo, estas dependências não necessitam de ser, inevitavelmente, referências para outras operações da *API*, podendo ser referências para serviços externos com o único propósito de testar a aplicação. Como o Código 15 ilustra, é possível definir dependências externas. Ao contrário das dependências existentes na especificação, estas requerem dados adicionais para invocar o serviço, consistindo no *URL*, método *HTTP*, parâmetros da *query*, *headers* e *body*.

```
{
  "/lists/{id}": {
    "externalDependencies": [
      {
        "URL": "protocol://domain-or-ip:port/path",
        "httpMethod": "POST",
        "query": {},
        "headers": {
          "Accept": "application/json",
          "Content-Type": "application/json"
        },
        "body": {
          "title": "Title"
        }
      }
    ]
  }
}
```

Código 15 - Ficheiro de configuração com dependências externas

Também é importante notar que na *API* do E-goi, tal como em muitas outras *API*, existem níveis hierárquicos mais profundos. Para tais recursos pode existir mais do que uma dependência, como o recurso */lists/{listId}/segments/{segmentId}* que necessita da criação de uma lista de contactos e um segmento para poder invocar uma operação de remoção.

Finalmente, após fornecer a especificação *OpenAPI* e o ficheiro de configuração contendo as dependências necessárias em cada recurso, é possível a criação de testes de aceitação. A Figura 19 apresenta o diagrama de sequência da solução implementada. Quando o processo de geração de testes é despoletado, é criada uma instância de *Specification*, caso a especificação *OpenAPI* seja válida. Seguidamente, é invocada a entidade *Generator*, que inicialmente cria as dependências necessárias, para cada recurso existente na especificação. Caso essas dependências sejam válidas, *Generator* cria os casos de teste necessários para cada operação. Após obter os casos de teste e dependências de um recurso, é criada uma instância de *TestClass*, que é composta por lista de *TestCase* e agrega uma lista de *Dependency*. Neste momento já existem objetos com toda a informação necessária agregada de forma coesa, sendo de seguida necessário transformar esses dados em testes para a ferramenta *PHPUnit*, através da classe *PHPUnitAdapter*. Por fim, para cada *TestClass* criada é necessário escrever esses testes na aplicação.

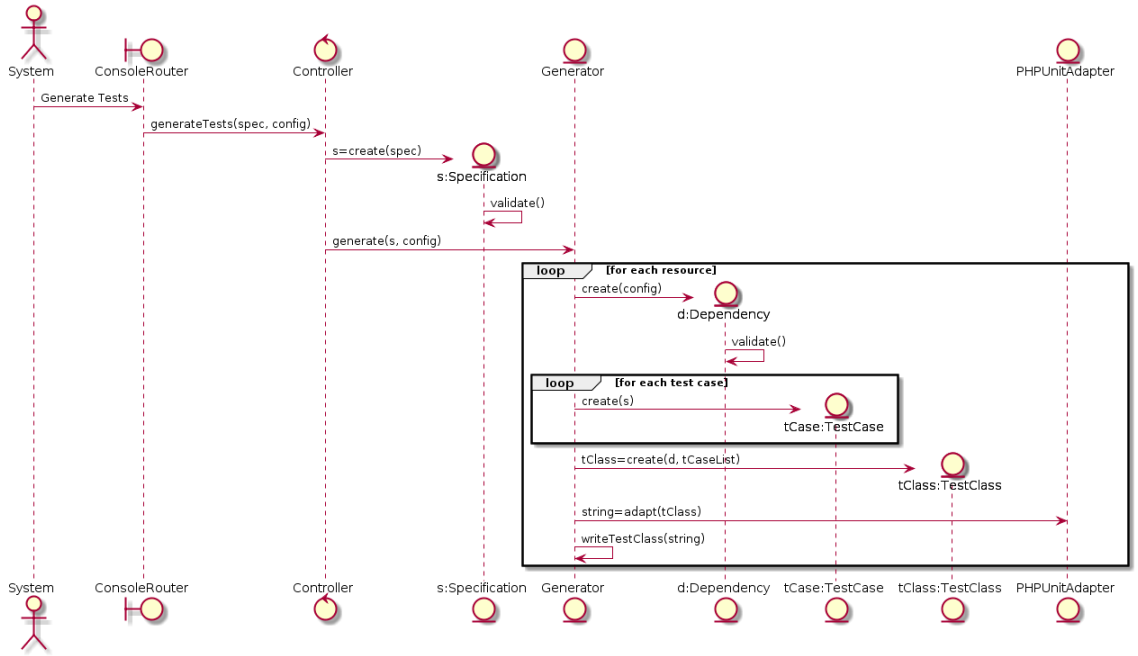

Figura 19 - Vista de cenário da geração de testes de aceitação

O Código 16 é um exemplo de teste gerado para a operação de remover uma lista de contactos, usando a ferramenta de testes *PHPUnit*. O método privado *dependencyCreateList* é gerado também com base na configuração fornecida e é utilizado para resolver as dependências necessárias.

```
public function testDeleteList(){
    // dependencies
    $listId = $this->dependencyCreateList('DeleteList');

    // delete request
    $request = new Delete('/lists/' . $listId);
    $response = $request->call();

    // assert server response
    $this->assertResponse($response);
}

private function dependencyCreateList($operation){
    // post request
    $request = new Post('/lists');
    $response = $request->call();

    // obtain id from specification
    $id = $this->getIdFromResponse($operation, $response);

    return $id;
}
```

Código 16 - Exemplo de um teste de aceitação para remoção de uma lista de contactos

O teste apresentado no Código 16 é um caso de sucesso, contudo os testes devem também cobrir falhas [48]. Isto significa que, para cada operação, mais que um teste de aceitação deve ser criado. Considerando a operação para obter as listas de contactos, os testes devem responder perguntas tais como [7]:

1. Qual o comportamento por defeito da *API* quando não é enviado nenhum parâmetro?
2. Qual o comportamento da *API* quando um parâmetro é enviado corretamente e com o respetivo valor correto?
3. Qual o comportamento da *API* quando é enviado um parâmetro incorreto?
4. Qual o comportamento da *API* quando o parâmetro não possui nenhum valor?
5. Qual o comportamento da *API* quando o valor do parâmetro está incorreto?
6. Qual o comportamento da *API* quando vários parâmetros são enviados em combinações corretas?
7. Qual o comportamento da *API* quando vários parâmetros são enviados numa combinação incorreta?
8. Qual o formato de dados por defeito da resposta da *API* quando não é enviada informação acerca do formato pretendido?
9. Qual o formato de dados da *API* para casos de sucesso e erro?
10. Qual o código de estado *HTTP* para casos de sucesso e erro?
11. Qual a resposta da *API* para métodos *HTTP*, *headers* e *URL* inesperados?

Para responder a estas perguntas é necessário criar vários testes de aceitação para cada operação. Para as questões 2, 3, 4 e 5 é necessário um teste de aceitação para cada parâmetro existente. A questão 6 e 7 também podem ser respondidas se a descrição da *API* contiver informação suficiente acerca das combinações de parâmetros (a *OpenAPI Specification* possui palavras-chave próprias para definir diferentes tipos de combinações). Acerca das restantes questões, a informação contida na especificação é suficiente para validar formatos de dados, códigos de estado e outro tipo de informação meta.

6.2 Registos de Utilização da API

O registo dos pedidos efetuados é essencial para a evolução de uma *API.* Os registos permitem identificar características comuns em vários pedidos, permitindo assim a produção de relatórios que ao serem analisados permitem a identificação de falhas e possíveis melhorias.

O modelo relacional presente na Figura 20 apresenta a relação entre um utilizador e os pedidos efetuados por este (um utilizador pode efetuar múltiplos pedidos e cada um desses pedidos corresponde apenas a um utilizador) e os dados necessários a registar na tabela *Request Log*.

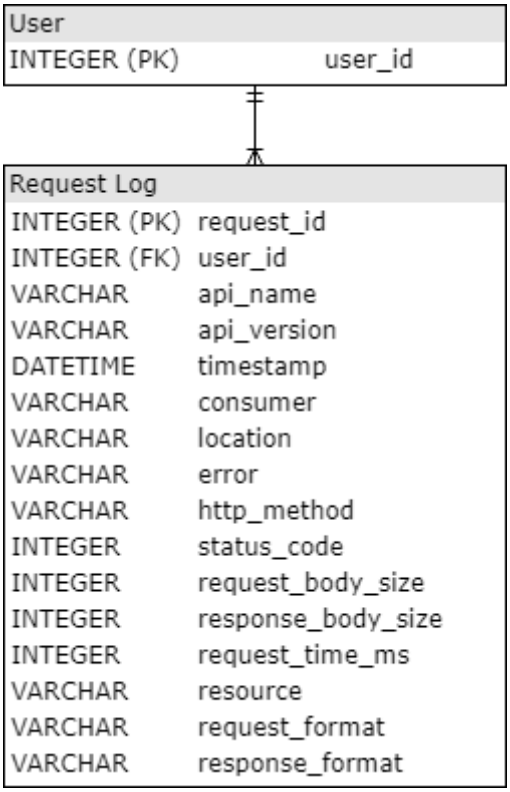

Figura 20 - Modelo relacional do sistema de registos de utilização da *API*

Esses dados consistem no nome da *API*, versão da *API*, data e hora, consumidor (origem do tráfego), localização, conteúdo em caso de erro, método *HTTP*, código de estado, tamanho do corpo do pedido, tamanho do corpo da resposta, tempo de execução do pedido, recurso requisitado, formato de dados do pedido e formato de dados da resposta. Os atributos mencionados no modelo relacional da Figura 20 estão descritos na Tabela 16.

Tabela 16 - Atributos da tabela *Request Log*

Atributo	Descrição
request_id	Chave primária e identificador único do pedido a ser registado
user_id	Chave estrangeira para a tabela de utilizadores; indica o utilizador que efetuou o pedido
api_name	Nome da *API* que recebeu o pedido a ser registado
api_version	Versão da *API* que recebeu o pedido a ser registado
timestamp	Data e hora do pedido
consumer	Aplicação cliente que invocou a *API*
location	Localização do cliente que efetuou o pedido
error	Registo de eventuais erros (desde erros na *API* a erros do utilizador)
http_method	Método *HTTP* utilizado para efetuar o pedido
status_code	Código de estado devolvido
request_body_size	Tamanho do corpo do pedido
response_body_size	Tamanho do corpo da resposta
request_time_ms	Tempo total de execução do pedido em milissegundos
resource	Recurso ao qual o pedido foi efetuado ou ação despoletada
request_format	Formato de dados do pedido recebido
response_format	Formato de dados da resposta devolvida pela *API*

Os sistemas de filas são utilizados sempre que existe grande afluência de dados, portanto o processo de criação de registos passa por duas fases. A primeira é a colocação dos registos numa fila e a segunda a leitura desses dados e respetiva colocação numa base de dados.

No caso da *API* pública do E-goi, o processo de colocação dos registos numa fila é despoletado por um observador de eventos da *Zend Framework*, que é acionado sempre que um pedido à *API* é efetuado. Quando um evento é captado, o controlador é acionado e cria um modelo com os dados necessários para os registos. Como é visível na Figura 21, os dados desse modelo são enviados para a componente *Producer*, responsável por colocar esses dados na *Queue*.

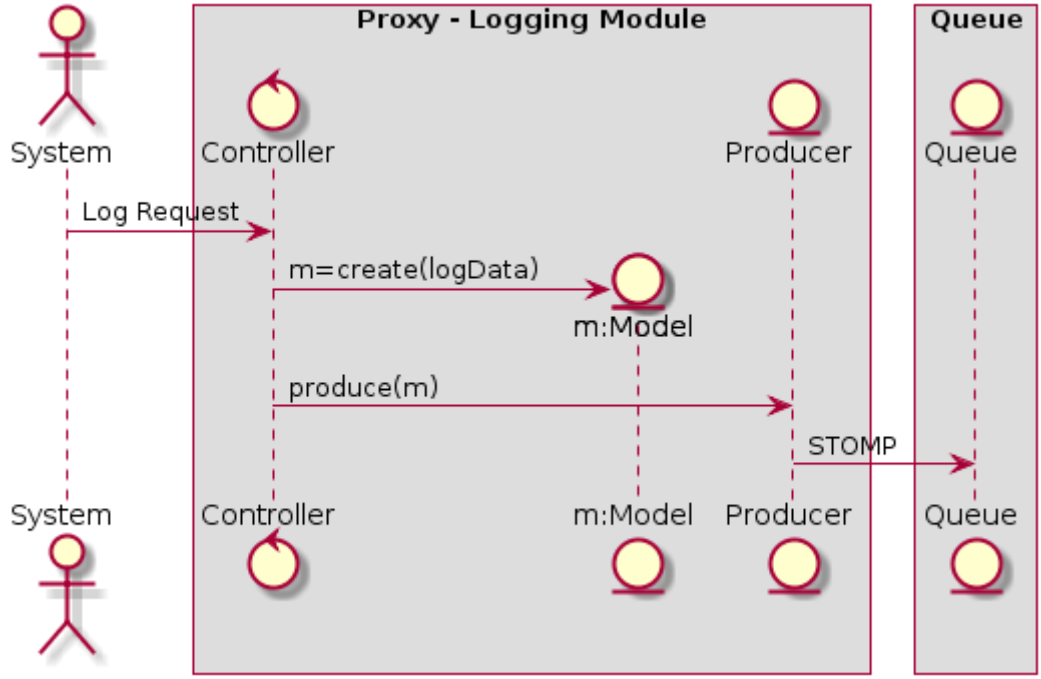

Figura 21 - Vista de cenário do envio de um registo para uma fila

Depois, para persistir os registos existentes numa fila é necessário primeiramente ler os dados da fila, que é responsabilidade da componente *Consumer*, como é verificável a partir da Figura 22. Seguidamente esses dados são convertidos num modelo e enviados para a componente *Repository*, que trata da comunicação com a base de dados, permitindo assim a persistência dos registos.

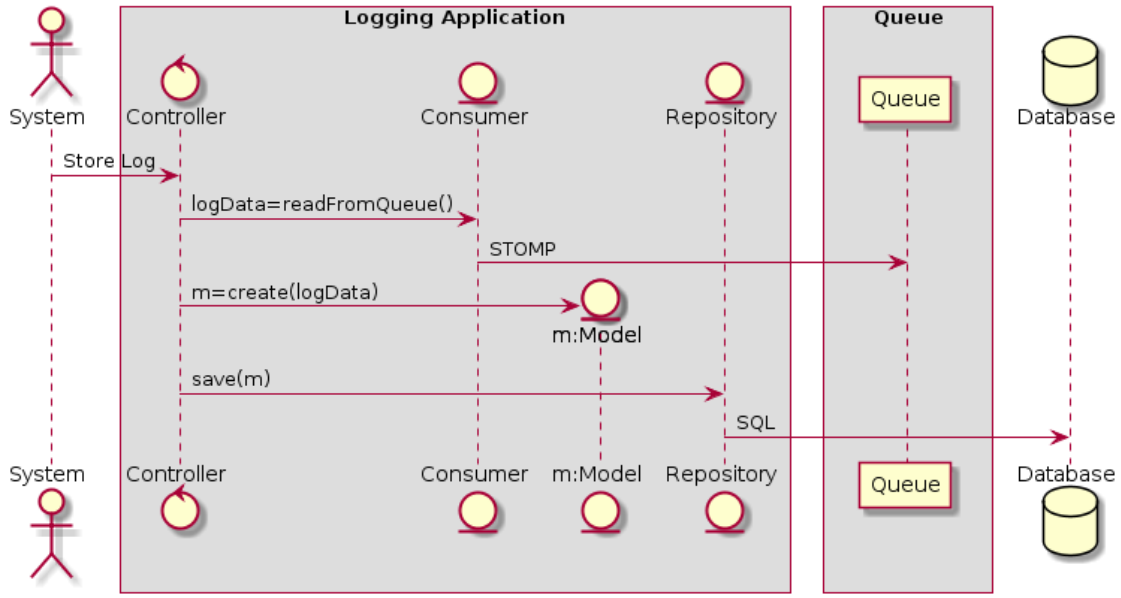

Figura 22 - Vista de cenário da leitura de registos de uma fila e respetiva persistência

No caso de *API* que expõem e registam dados que podem ser considerados sensíveis, é importante ter algumas considerações de segurança, como a encriptação de informação sensível [7]. Visto que, na *API* do E-goi, estes registos estão associados com os seus clientes, torna-se necessário proteger estes dados contra eventuais furtos de informação contida nas bases de dados. Para isso, foi utilizada a função de *MySQL* denominada *AES_ENCRYPT()*, baseada no algoritmo de encriptação *Advanced Encryption Standard (AES)*, que recebe como argumentos os dados a encriptar e uma chave de encriptação de 128 *bits*. Para segurança adicional, essa chave de encriptação é também cifrada primeiramente [49], recorrendo-se para tal ao algoritmo *Secure Hash Algorithm 2 (SHA2)*.

7 Experimentação e Avaliação

Como explicado na secção 1.2, existem diversos problemas a resolver, problemas esses que trazem custos de manutenção elevados e indesejados. Estes problemas consistem na existência de documentação e *SDK* obsoletos, por terem sido criados de forma estática e na ausência de testes de *software* e registos da utilização da *API*.

Dessa forma, como podemos verificar na secção 1.3, os objetivos são a geração de documentação, *SDK* e testes de forma dinâmica e também a criação de um sistema de registos da utilização da *API*. Relativamente à documentação, *SDK* e testes, a geração destes é permitida devido à existência de um contrato, definido por uma especificação *OpenAPI*. Assim sendo, para conseguir gerar estes artefactos é necessário especificar detalhadamente a *API REST*, pelo que o detalhe e qualidade da especificação se reflete nos objetos gerados.

Portanto, para avaliar a solução desenvolvida é necessário avaliar a sua especificação. Quanto aos artefactos gerados, a documentação foi criada recorrendo à ferramenta *Swagger UI* e o *SDK* com recurso à ferramenta *Swagger Codegen*. Estas ferramentas são amplamente utilizadas por várias empresas [23] e são constantemente atualizadas e testadas por contribuidores. Dessa forma, não é necessário avaliar o processo de geração de documentação e *SDK*. Por outro lado, o sistema de geração de testes deve ser avaliado, bem como o sistema de registos de utilização da *API*.

No presente capítulo, são apresentadas e explicadas as grandezas, hipóteses, metodologias e os testes estatísticos que se pretende realizar de forma a comprovar que as soluções cumprem os objetivos propostos.

7.1 Grandezas

Depois de identificados os objetivos a alcançar, verifica-se que existem várias componentes a serem avaliadas, sendo estas a especificação da *API*, o sistema de geração de testes e o sistema de registos de utilização da *API*. Para a especificação identificam-se a seguinte grandeza a avaliar:

- **Clareza da documentação para o utilizador:** esta grandeza traduz em percentagem a satisfação dos utilizadores perante a clareza da documentação da *API.* A clareza da documentação pode ser usada como uma métrica para testar a especificação, sendo que a documentação é gerada com base na informação contida na especificação;

Para o sistema de geração de testes, foi identificada a seguinte grandeza:

- **Quantidade de serviços coerentes entre a especificação e a *API*:** esta grandeza indica a quantidade de serviços da *API* que se encontram coerentes com a especificação;

Relativamente ao sistema de registos de utilização da *API*, a métrica identificada é:

- **Quantidade de pedidos registados:** esta grandeza indica a quantidade de pedidos à *API* que foram registados com sucesso. Caso o sistema de registos se encontre bem implementado, será possível registar todos os pedidos efetuados à *API*.

7.2 Hipóteses

Em relação às grandezas identificadas na secção 7.1, a "clareza da documentação para o utilizador" origina duas hipóteses:

- **Hipótese 1:** a clareza da documentação da nova *API* é superior a à da *API* anterior;
- **Hipótese 2:** a clareza da documentação da nova *API* é superior a 80%.

Quanto às restantes grandezas, a "quantidade de serviços coerentes entre a especificação e a *API*" deve ser igual a 100%, não sendo necessário estabelecer uma hipótese. O mesmo se aplica à "quantidade de pedidos registados", que deve ser igual à quantidade de pedidos efetuados, ou seja, 100% dos pedidos devem ser registados.

7.3 Metodologia

De forma a poder testar as hipóteses 1 e 2 foi realizado um questionário respondido pelos colaboradores das equipas da E-goi que utilizam a *API* pública ou estiveram envolvidos no

desenvolvimento da *API* anterior, totalizando 32 pessoas. O questionário realizado consiste em classificar a documentação de cada uma das *API* numa escala percentual e servirá para averiguar se a documentação é clara o suficiente para facilitar a aprendizagem e uso da *API*.

De forma a testar o sistema de geração de testes, é necessário garantir que todos os serviços da *API* estão coerentes com o contrato definido na especificação. Para averiguar se a solução proposta cumpre os requisitos, foram efetuados testes de aceitação criados de forma manual a serviços da *API*. De seguida foram executados os testes de aceitação gerados de forma dinâmica, e foi feita uma comparação dos resultados produzidos pelos testes criados de forma manual e os testes gerados dinamicamente. Como os resultados dos testes foram idênticos, é possível concluir que se obteve uma solução de qualidade.

Por fim, após assegurar a qualidade da especificação e do sistema de geração de testes, foi preciso verificar se os pedidos registados são iguais aos pedidos efetuados à *API*. Para isso foram realizados testes de integração entre a *API* pública e o sistema de registos de utilização da *API*, concluindo assim que 100% dos pedidos efetuados são registados com sucesso.

7.4 Testes estatísticos

Com exceção das hipóteses 1 e 2, como o resultado dos testes efetuados tem de ser igual a 100%, não é necessário efetuar testes estatísticos. A avaliação dessas componentes é efetuada através da comparação direta dos resultados obtidos com os resultados esperados.

No entanto, tanto para a hipótese 1 como para a hipótese 2, é possível realizar testes estatísticos. Pretende-se averiguar se a clareza da documentação da nova *API* para os utilizadores é superior à da *API* anterior e se é superior a 80%. Assim, inicialmente é comparada a documentação da *API* anterior com os 80% de clareza desejada. Caso a clareza da documentação seja superior a 80% é necessário comparar a documentação da nova *API* com a documentação da *API* anterior. Caso contrário, a documentação da nova *API* é comparada com os 80% de clareza mínima que se deseja obter na sua documentação.

Como os resultados obtidos nos questionários foram obtidos de 32 funcionários, é possível, para comparar a clareza da documentação, utilizar um *teste-t* de amostra única à direita, sendo que os dados obtidos permitem a realização de testes paramétricos, uma vez que possuem amostras com pelo menos trinta entradas, pelo que se podem considerar normais [50].

Para efetuar um *teste-t* para comparar a documentação *API* anterior com os 80% de clareza, define-se a hipótese nula (*h0*) e a hipótese alternativa (*h1*), onde *m0* é o valor de comparação com a média da amostra (80%), *h0* é a média da amostra ser igual a *m0* e *h1* é a média da amostra ser superior a *m0*:

$$h0: \mu = m0$$
$$h1: \mu > m0$$

Após efetuar o teste, obteve-se cerca de 39% como média da amostra e um *p-value* de 1. Como o *p-value* é maior que "0,05" (valor relativo ao grau de confiança de 95%), não pode ser rejeitada a hipótese nula, o que significa que a clareza da documentação da *API* anterior para os utilizadores é igual ou inferior a 80%.

Assim sendo, procede-se à comparação entre a documentação da nova *API* e o grau de 80% de clareza para o utilizador, novamente através de um *teste-t* à direita. As hipótese nula e alternativa são:

$$h0: \mu = 80\%$$
$$h1: \mu > 80\%$$

Efetuando o *teste-t*, obtém-se uma média da amostra de cerca de 87% e um *p-value* de aproximadamente "0,0003". Como o *p-value* é inferior a "0,05", rejeita-se a hipótese nula com 95% de confiança, podendo assumir-se que a clareza da documentação produzida a partir da especificação é superior a 80%, o que significa que a *API* está adequadamente especificada.

Os resultados obtidos provam ambas as hipóteses definidas na secção 7.2. Com o resultado do segundo *teste-t* prova-se, com um grau de confiança de 95%, que a clareza da documentação da nova *API* é superior a 80% (hipótese 1). Uma vez que foi também concluído que a clareza média da documentação referente à *API* anterior não é superior a 80%, conclui-se que a clareza da nova *API* é superior à da *API* anterior, provando-se assim a hipótese 2.

8 Conclusões

Documentação e testes de *software* são componentes críticas de qualquer *API*. Adicionalmente, uma especificação de *API* usada numa abordagem *design-first* pode ser útil para aumentar a qualidade do *software*. Atuando como um contrato para a *API*, este pode garantir que as componentes que derivam da *API*, como documentação e *SDK*, podem confiar nesse documento, visto que qualquer alteração na especificação vai ser refletida na *API* e suas componentes.

Como a especificação pode não estar coerente com a implementação de baixo nível dos serviços, testes de aceitação podem ser criados para testar a interconexão entre a *API* e a sua especificação. A geração dos artefactos mencionados pode também trazer uma redução de custos de manutenção significativa.

Ainda que existam várias ferramentas de código aberto que implementam a *OAS*, como *Swagger UI* e *Swagger Code Generator*, há uma escassez de ferramentas de código aberto para gerar testes de *software*. Por essa razão, foi proposta uma solução para ultrapassar este problema, capaz de gerar testes de aceitação com sucesso. No entanto, é importante reparar que ainda é necessário produzir outros tipos de testes de *software* e testes manuais. O objetivo da solução apresentada não é substituir estes testes, mas sim garantir que a especificação está sempre coerente com a implementação da *API*.

Conclui-se também que o sistema de registos da aplicação é um sistema que não está relacionado com a especificação, mas que é bastante importante para a evolução da *API*, já que é através da informação recolhida por este sistema que o E-goi pode decidir que mudanças efetuar na *API*.

8.1 Trabalho Realizado

Para cumprir os objetivos realizou-se um estado da arte onde foram analisadas diferentes *API* na área de *marketing*. Com isso identificaram-se aspetos essenciais a incluir na documentação de uma *API*. Também foi estudado *REST*, evolução e versionamento e analisaram-se linguagens de especificação de *API REST*. Para finalizar o estudo efetuado, foi estudada a importância da existência de registos de utilização da *API*.

Seguidamente efetuou-se uma de análise de valor da solução em que se detalhou o processo de inovação com aplicação do modelo *NCD*, a proposta de valor para cada um dos segmentos de clientes existentes e o valor para os clientes. Para visualizar o projeto e a sua importância para a empresa de forma mais clara e abrangente foi elaborado também o modelo *Canvas* e a cadeia de valor de *Porter*. Ainda na análise de valor realizou-se uma tomada de decisão utilizando o modelo *AHP*, com o objetivo de decidir a linguagem de especificação para *API REST* mais adequada e concluiu-se que a melhor opção era a especificação *OpenAPI*.

Terminada a análise de valor realizou-se uma análise de requisitos e de negócio, onde foram identificados os requisitos essenciais e representadas as entidades do domínio.

De seguida foi elaborado o *design* arquitetural da solução e possíveis alternativas. Esse design arquitetural consiste na vista lógica, que permite representar os principais componentes do sistema e suas relações e na vista de implantação, que permite representar a distribuição física do sistema.

Por fim foram avaliadas as soluções desenvolvidas em relação aos objetivos propostos. Para tal foram escolhidas as grandezas de relevo e especificadas as hipóteses a testar. Foi ainda detalhada a metodologia de avaliação e os testes estatísticos adequados às hipóteses.

8.2 Trabalho e Pesquisa Futura

A *OpenAPI Specification* está em constante evolução e todas as soluções e ferramentas relacionadas necessitam de acompanhá-la. Para comprovar o trabalho que está atualmente a ser feito na *Open API Initiative* e na comunidade *Swagger*, o número de tarefas abertas para os projetos *OAS* [10], *Swagger UI* [51] e *Swagger Code Generator* [23] estão apresentadas na Tabela 17.

Tabela 17 - Tarefas da *OAS*, *Swagger UI* e *Swagger Code Generator*

	Tarefas Abertas	**Tarefas fechadas**
OAS	338	664
Swagger UI	236	2932
Swagger Code Generator	1362	2644

Como informado neste documento, *Swagger Code Generator* ainda não é capaz de usar a versão 3.0 da *OAS*. A sua última versão estável é a versão 2.3.1. Ambas as versões 2.4.0 e 3.0.0 (versão na qual está previsto o suporte para *OpenAPI Specification* 3.0) estão a ser desenvolvidas, mas as suas datas de lançamento oficial ainda não foram definidas.

Apesar de a solução apresentada para a geração de testes de aceitação resolver os problemas relativos à escassez de testes, ainda há muito espaço para melhorias. Por exemplo, a integração deste projeto num sistema de entrega continua e a criação de uma interface gráfica. Uma interface gráfica ajudaria o utilizador a criar os ficheiros de configuração, sendo que a escrita deste tipo de ficheiros é suscetível ao erro humano. Através de uma interface gráfica, o sistema poderia sugerir possíveis dependências e isso ajudaria o utilizador a efetuar pedidos a sistemas externos, gravando esses pedidos como dependências de um ou mais recursos. Também seria útil para o utilizar definir valores adicionais, além dos exemplos presentes na especificação, para aumentar a eficácia dos testes.

Adicionalmente, algumas dependências externas podem introduzir atrasos ao correr casos de teste [48]. Por essa razão, é importante criar dados que simulem essas dependências. Assim sendo, o trabalho futuro no sistema de geração de testes passa por considerar a criação destes dados de forma a aumentar a qualidade dos testes.

Relativamente ao sistema de registos da utilização da *API*, como trabalho futuro será necessário utilizar esses dados para efetuar uma correta monitorização da *API*. Esses dados são essenciais para a produção de estatísticas e relatórios, que podem também ser apresentados aos clientes no futuro.

8.3 Contributos

Esta dissertação traz contributos para a *API* pública do E-goi, sendo que com este trabalho vai ser evitado que a documentação e *SDK* se tornem obsoletos, que é um dos maiores problemas enfrentados pelos clientes. Os testes têm também um papel bastante importante, sendo que estes forçam a *API* a seguir a sua especificação sempre que uma evolução ocorre. Por fim, este trabalho traz contributos aos desenvolvedores do E-goi, visto que os artefactos mencionados são gerados dinamicamente a partir do mesmo documento, sem grande esforço.

Como o E-goi não possui registos da utilização da *API*, o sistema de registos é também um projeto crucial, tanto para a E-goi como para os seus clientes, visto que auxiliará a E-goi entender melhor as necessidades desses mesmos clientes, permitindo a alteração de serviços menos conseguidos e melhorando no geral o processo evolutivo da *API*.

Este trabalho também contribuiu ainda com o capítulo *"Dynamic Generation of Documentation, Code and Tests for a Digital Marketing Platform's API"* [52] submetido para o livro editado *"Code Generation, Analysis Tools, and Testing for Quality"* [53], a ser publicado.

9 Referências

[1] A. Abelló, C. P. Ayala, F. Carles, G. Cristina, O. Marc e R. Oscar, "A Data-Driven Approach to Improve the Process of Data-Intensive API Creation and Evolution," *CAiSE-Forum-DC,* pp. 1-8, 2017.

[2] P. Leitner, A. Michlmayr, F. Rosenberg e S. Dustdar, "End-to-End Versioning Support for Web Services," em *IEEE International Conference on Services Computing*, Honolulu, HI, USA, 2008.

[3] S. a. G. C. a. E. G. Schwichtenberg, "From Open API to Semantic Specifications and Code Adapters," em *Web Services (ICWS), 2017 IEEE International Conference on*, Honolulu, 2017.

[4] M. a. S. S. a. S. A. Meng, "Application Programming Interface Documentation: What Do Software Developers Want?," *Journal of Technical Writing and Communication,* p. 0047281617721853, 2017.

[5] G. Uddin e M. P. Robillard, "How API Documentation Fails," *IEEE Software,* pp. 68-75, 30 Junho 2015.

[6] S. M. Sohan, C. Anslow e F. Maurer, "A Case Study of Web API Evolution," em *IEEE World Congress on Services*, New York, NY, USA, 2015.

[7] B. De, "API Management," em *API Management*, Berkeley, CA, Apress, 2017, pp. 15-28.

[8] M. P. a. D. R. Robillard, "A field study of API learning obstacles," *Empirical Software Engineering,* pp. 703-732, 2011.

[9] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” *National Institute of Standards and Technology,* vol. 7007, 2002.

[10] OpenAPI Initiative, “OpenAPI-Specification,” Github, [Online]. Available: https://github.com/OAI/OpenAPI-Specification. [Acedido em 31 Janeiro 2018].

[11] R. T. Fielding e R. N. Taylor, Architectural styles and the design of network-based software architectures, University of California, Irvine Doctoral dissertation, 2000.

[12] J. Li, Y. Xiong, X. Liu e L. Zhang, “How does web service API evolution affect clients?,” em *Web Services (ICWS), 2013 IEEE 20th International Conference on*, Santa Clara, CA, USA, 2013.

[13] M. Jakl, “Representational state transfer,” University of Technology Vienna, 2005.

[14] M. Massé, REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces, O'Reilly Media, Inc., 2011.

[15] L. a. A. M. a. R. S. Richardson, RESTful Web APIs: Services for a Changing World, O'Reilly Media, Inc., 2013.

[16] J. a. P. S. a. R. I. Webber, REST in practice: Hypermedia and systems architecture, O'Reilly Media, Inc., 2010.

[17] B. Mulloy, Web API design, 2013.

[18] B. a. P. P. F. a. D. F. C. a. M. P. Costa, “Evaluating a Representational State Transfer (REST) Architecture: What is the Impact of REST in My Architecture?,” em *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, Sydney, 2014.

[19] R. a. G. J. a. M. J. a. F. H. a. M. L. a. L. P. a. B.-L. T. Fielding, “Rfc 2616, hypertext transfer protocol--http/1.1, 1999,” 2009. [Online]. Available: http://www. rfc. net/rfc2616. html. [Acedido em 16 Maio 2018].

[20] M. Biehl, RESTful API Design: Best Practices in API Design with REST (API-University Series Book 3), API-University Press, 2016.

[21] T. Preston-Werner, “Semantic Versioning 2.0.0,” [Online]. Available: https://semver.org/. [Acedido em 2 Fevereiro 2018].

[22] R. Pinkham, “What Is the Difference Between Swagger and OpenAPI?,” SmartBear, 26 Outubro 2017. [Online]. Available: https://swagger.io/blog/difference-between-swagger-and-openapi/. [Acedido em 21 Novembro 2017].

[23] Swagger, “Swagger Code Generator,” 15 May 2018. [Online]. Available: https://github.com/swagger-api/swagger-codegen/tree/3.0.0.

[24] Apiary, “API Blueprint,” 25 May 2017. [Online]. Available: https://github.com/apiaryio/api-blueprint.

[25] raml-org, “raml-spec,” Github, [Online]. Available: https://github.com/raml-org/raml-spec. [Acedido em 31 Janeiro 2018].

[26] apiaryio, “api-blueprint,” Github, [Online]. Available: https://github.com/apiaryio/api-blueprint. [Acedido em 31 Janeiro 2018].

[27] W3, “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition),” 27 Abril 2007. [Online]. Available: https://www.w3.org/TR/soap12/. [Acedido em 23 Junho 2018].

[28] D. Winer, 30 Junho 2003. [Online]. Available: http://xmlrpc.scripting.com/spec.html. [Acedido em 22 Junho 2018].

[29] P. A. Koen, G. M. Ajamian, S. Boyce, A. Clamen, E. Fisher, S. Fountoulakis, A. Johnson, P. Puri e R. Seibert, Fuzzy Front End: Effective Methods, Tools, and Techniques, Wiley, New York, NY, 2002.

[30] S. Nicola, E. P. Ferreira e J. J. P. Ferreira, “ A novel framework for modeling value for the customer, an essay on negotiation,” *International Journal of Information Technology & Decision Making,* vol. 11(3), pp. 661 - 703, 2012.

[31] T. Woodall, “Conceptualising'value for the customer': An attributional, structural and dispositional analysis,” *Academy of marketing science review,* p. 1, 2003.

[32] J. C. Sweeney e G. N. Soutar, “Consumer perceived value: The development of a multiple item scale,” *Journal of Retailing,* vol. 77, p. 203–220, 2001.

[33] V. A. Zeithaml, “Consumer Perceptions of Price, Quality and Value: A Means-End Model and value: a means-end model and synthesis of evidence,” *The Journal of marketing,* vol. 52(3), pp. 2-22, 1988.

[34] 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.

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

[36] D. Barnes, Understanding Business: Processes, New Fetter Lane, London: Psychology Press, 2001.

[37] C. S. Marins, D. d. O. Souza e M. d. S. Barros, “O uso do método de análise hierárquica (AHP) na tomada de decisões gerenciais–um estudo de caso,” *XLI SBPO,* vol. 1, 2009.

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

[39] E. Mu e M. Pereyra-Rojas, “Understanding the Analytic Hierarchy,” em *Practical Decision Making using Super Decisions v3*, Springer, 2018, pp. 7-22.

[40] IEEE, “IEEE Standard Glossary of Software Engineering Terminology,” 28 Setembro 1990. [Online].

[41] A. M. Moreno e A. Yagüe, “Agile user stories enriched with usability,” em *International Conference on Agile Software Development*, Malmö, 2012.

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

[43] M. Finsterwalder, “Automating acceptance tests for GUI applications in an extreme programming environment,” em *Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, Hamburg, 2001.

[44] S. a. C. E. B. a. G. M. Hotomski, “Keeping Evolving Requirements and Acceptance Tests Aligned with Automatically Generated Guidance,” em *International Working Conference on Requirements Engineering: Foundation for Software Quality*, 2018.

[45] J. a. H. R. a. J. R. a. G. E. Vlissides, “Design patterns: Elements of reusable object-oriented software,” *Reading: Addison-Wesley,* p. 11, 1995.

[46] J. E. a. B. H. a. F. D. a. V. M. T. Montandon, “Documenting apis with examples: Lessons learned with the apiminer platform,” em *Reverse Engineering (WCRE), 2013 20th Working Conference on*, Koblenz, 2013.

[47] S. M. a. S. J. a. M. F. a. B. C. Nasehi, “What makes a good code example?: A study of programming Q&A in StackOverflow,” em *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, Trento, 2012.

[48] D. a. H. T. a. P. J. a. S. T. a. T. B. Bowes, “How good are my tests?,” em *Emerging Trends in Software Metrics (WETSoM), 2017 IEEE/ACM 8th Workshop on*, Buenos Aires, 2017.

[49] MySQL, “Encryption and Compression Functions,” [Online]. Available: https://dev.mysql.com/doc/refman/5.5/en/encryption-functions.html. [Acedido em 1 Junho 2018].

[50] A. Ghasemi e S. Zahediasl, “Normality tests for statistical analysis: a guide for non-statisticians,” *International journal of endocrinology and metabolism,* vol. 10(2), p. 486–489, 20 Abril 2012.

[51] Swagger, “Swagger UI,” 30 May 2018. [Online]. Available: https://github.com/swagger-api/swagger-ui.

[52] R. Santos, I. Pereira e I. Azevedo, “Dynamic Generation of Documentation, Code and Tests for a Digital Marketing Platform’s,” em *Code Generation, Analysis Tools, and Testing for Quality*, IGI Global, Por publicar.

[53] A. Simões, M. T. Pinto e R. A. P. d. Queirós, Code Generation, Analysis Tools, and Testing for Quality, IGI Global, Por publicar.

[54] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte e D. Winer, “Simple object access protocol (SOAP) 1.1,” 8 Maio 2000. [Online]. Available: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/. [Acedido em 10 Fevereiro 2018].