INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

RapiTest – Aplicação Web para testar API

Duarte Filipe de Melo dos Santos Felício

Licenciado

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor José Manuel de Campos Lages Garcia Simão
Doutor Nuno Miguel Soares Datia

Júri:

Presidente: Doutor Nuno Miguel Machado Cruz

Vogais: Doutor Carlos Jorge de Sousa Gonçalves
Doutor José Manuel de Campos Lages Garcia Simão

Novembro, 2022

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Electrónica e Telecomunicações e de Computadores

RapiTest – Aplicação Web para testar API

Duarte Filipe de Melo dos Santos Felício

Licenciado

Projecto Final para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientadores : Doutor José Manuel de Campos Lages Garcia Simão
Doutor Nuno Miguel Soares Datia

Júri:

Presidente: Doutor Nuno Miguel Machado Cruz

Vogais: Doutor Carlos Jorge de Sousa Gonçalves
Doutor José Manuel de Campos Lages Garcia Simão

Novembro, 2022

A todos os que ajudaram ao longo do caminho...

Agradecimentos

Começo por agradecer aos professores José Simão e Nuno Datia, por toda a orientação e ajuda que disponibilizaram no decorrer de todo o desenvolvimento deste projeto, especialmente na ajuda e *feedback* sobre a realização desta dissertação.

Agradeço ao ISEL e a todos os seus funcionários, especialmente a todos os professores, pois nunca tive um que não estivesse sempre disposto a ajudar e esclarecer qualquer dúvida, acompanhando sempre ao longo do percurso. Quero também dar um especial agradecimento aos meus amigos e família por toda a ajuda durante todo o caminho e por nunca duvidarem da minha paixão. Quero agradecer também ao Francisco Pêgo, um dos meus primeiros e melhores amigos, que podemos contar com em qualquer situação, ao Josué Batatas, por todos os trabalhos em que me ajudou imenso, ao Jorge Martins, por todas as boleias e momentos divertidos até ao metro. Ao Ivo Pereira, por todas as conversas filosóficas depois das aulas, ao Rúben Café, pelo seu especial sentido de humor, e claro, ao Bernardo Costa, pelo incomparável trabalho de equipa.

Posto isto, este trabalho foi realizado ao longo de muitas horas, muitos dias e muitas noites e quero, principalmente, agradecer a todos os que me fizeram quem sou hoje.

- Duarte Felício

Resumo

Quando se trata de serviços na *web*, as *RESTful Web* API tornaram-se o padrão por norma desde o ano 2000. *Application Programming Interface's* (API) expõem dados de *back-end*, portanto, é crucial que sejam robustas, seguras e confiáveis de forma a manter os dados confidenciais protegidos. Embora as ferramentas existentes para automatizar a geração de casos de teste para API tenham mostrado um potencial significativo, estas são limitadas na sua aplicabilidade, pois focam-se apenas em dados aleatórias por meio de *fuzzing*. Usando apenas especificações de API, é impraticável descrever casos de teste personalizados e específicos. Esta dissertação apresenta RapiTest, uma aplicação de teste contínuo de caixa preta e de código aberto para API REST. Esta tira partido da especificação da API para gerar testes automaticamente, mas, também tira partido de uma nova *Domain Specific Language* DSL chamada de Test Specification Language (TSL), para criar casos de teste personalizados. A aplicação *web* RapiTest permite a configuração de várias verificações nativas, relativas à segurança e exatidão das respostas, enquanto executa os testes em intervalos regulares, como a cada 24 horas. Dessa forma, a API pode ser supervisionada continuamente para garantir o seu correto funcionamento.

Palavras-chave: DSL; REST; API; Aplicação Web; Teste caixa-preta; Confiabilidade; Integração de Sistemas

Abstract

When it comes to web services, RESTful web API have become the *de facto* standard since 2000. Those Application Programming Interface's (API) expose back-end data, so it is crucial that they are robust, secure, and reliable to keep sensitive data protected. Although existing tools for automating API test case generation have shown significant potential, they are limited in their applicability since they focus solely on random inputs through fuzzing. Using only API specifications, it is impractical to describe personalized and specific test case workflows. This paper introduces RapiTest, an open-source continuous black-box testing application for RESTful web API. It takes advantage of the API specification to automatically generate tests, but also makes use of a new Domain Specific Language DSL named Test Specification Language (TSL), to create rich test cases. The RapiTest web application allows the setup of several predefined verifications, regarding security and correctness of the responses, while running the tests at regular intervals, such as every 24 hours. In this way, the API can be monitored continuously to ensure it is running correctly.

Keywords: DSL; REST; API; Web Application; Black-box Testing; Reliability; System Integration

Índice

Lista de Figuras xvii

Lista de Tabelas xix

Lista de Listagens xxi

Lista de Abreviaturas e Siglas xxiii

Glossário xxv

1 Introdução 1

1.1 Problema 3

1.2 Organização do Documento 6

2 Estado da Arte 7

2.1 Artigos Propostos 7

2.2 Ferramentas Disponíveis 9

2.3 Resumo 10

3 Abordagem 13

3.1 Tipos de Teste 13

3.1.1 Validação 13

3.1.2 Lógica de Utilização 16

3.1.3 Carga 16
3.1.4 Segurança 17
3.2 Arquitetura de Suporte aos Testes 19
3.3 *Test Specification Language*, TSL 19
3.4 Casos de Utilização Suportados 23
3.4.1 Estrutura da Interface Visual 28
3.5 Arquitetura 29

4 Implementação **33**
4.1 Tecnologias de Código Aberto 33
4.2 API da Componente Servidora 34
4.3 Autenticação de Utilizadores 35
4.4 Página de Entrada 36
4.5 Configurar um Novo Teste 37
4.5.1 Fornecimento de um Nome para o Teste 38
4.5.2 *Upload* Local ou por URL da OAS 38
4.5.3 Criação ou *Upload* de Ficheiro TSL 40
4.5.4 Configuração do Temporizador e Outros Dados 46
4.5.5 Configurar Outro Teste ou Observar os Resultados 47
4.6 Validação, Controlo e Execução dos Testes 48
4.6.1 Validação e Controlo - *Worker Service* 48
4.6.2 Suporte a Múltiplos Tipos de Linguagem 53
4.6.3 Execução dos Testes - *Worker Service* 54
4.7 *Monitor Tests* 57

5 *Deploy* e Resultados **65**
5.1 *Deploy* 65
5.1.1 Docker Compose 68
5.2 Resultados 70
5.2.1 Testes sem TSL 73
5.2.2 Testes com TSL 74

6 Conclusões e Trabalho Futuro **83**

6.1 Conclusões 83

6.2 Trabalho Futuro 84

Referências **87**

Lista de Figuras

3.1 *Workflow* para *Petstore*. 16

3.2 Exemplo de *Broken Object Level Authorization*. 17

3.3 Arquitetura de suporte aos testes. 19

3.4 Contrato para validações externas. 23

3.5 Casos de utilização - Não autenticado. 23

3.6 Casos de utilização - Autenticado. 25

3.7 Barra de navegação. 28

3.8 *RapiTest workflow*. 29

3.9 Arquitetura do Projeto. 31

4.1 Modelo da base de dados para a autenticação, *framework IdentityServer*. . 35

4.2 Página inicial da aplicação para um utilizador não autenticado. 36

4.3 Página inicial para um utilizador autenticado 37

4.4 Página para fornecer o nome do teste. 38

4.5 Página para *upload* da especificação 39

4.6 Página para escolha do utilizador 40

4.7 Página inicial da criação. 41

4.8 Janela para criar um *workflow*. 41

4.9 Página com um *workflow* criado 41

4.10 Formulário para criar um teste. 42

4.11 Página com um teste criado 43
4.12 Formulário para criar um teste de carga. 44
4.13 Página com um teste de carga criado 44
4.14 Página para o passo opcional da configuração de um teste 45
4.15 Página para o passo da configuração da frequência. 46
4.16 Modelo para a configuração de testes - `API` e *ExternalDll*. 47
4.17 Modelo de Dados para a deserialização do ficheiro `TSL`. 49
4.18 Modelo de Dados para a deserialização do ficheiro `TSL` final. 50
4.19 Modelo para o Suporte a múltiplos tipos de linguagem. 54
4.20 Página para a monitorização dos testes. 58
4.21 *Overview* do relatório. 60
4.22 Resultados dos *workflows*. 60
4.23 Resultados dos testes de carga. 61
4.24 Resultados dos testes gerados. 62
4.25 Testes em falta. 62

5.1 Estrutura *Docker*. 66
5.2 Estrutura máquinas virtuais. 66
5.3 Arquitetura através de contentores *Docker*. 68
5.4 Descrição da razão da falha da verificação do esquema - *PGIL-01*. 75
5.5 Descrição da razão da falha da verificação do esquema - *PGIL-02*. 75
5.6 Resultados dos testes de *workflow* - *petstore*. 76
5.7 Resultados da validação *custom*. 82

Lista de Tabelas

2.1 Características das ferramentas analisadas. 10

3.1 Validações fornecidas nativamente. 15

4.1 *Endpoints* disponibilizados pela API. 34
4.2 Trocas dos modelos antigos para os modelos finais da lógica da aplicação. 51

5.1 Dados sobre a API *petstore*. 71
5.2 Dados sobre a API *PGIL-01*. 72
5.3 Dados sobre a API *PGIL-02*. 72
5.4 *Hardware* do computador para testes. 73
5.5 Resultados sem TSL - *petstore*. 73
5.6 Resultados sem TSL - *PGIL-01* e *PGIL-02*. 73
5.7 Resultados com TSL - Validação *PGIL-01*. 74
5.8 Resultados dos testes de Carga - *PGIL-01*. 77
5.9 Estatísticas do teste de carga - *PGIL-01*. 77

Lista de Listagens

1.1 *Petstore* - Excerto da especificação da API. 4

3.1 *Petstore* - Excerto da especificação da API completado. 14

3.2 Exemplo da descrição de um teste em TSL. 20

3.3 Exemplo de ficheiro dicionário. 22

4.1 Exemplo de uma serialização do ficheiro de testes. 53

4.2 Método responsável pela realização dos testes. 55

4.3 Exemplo do ficheiro de resultados de teste. 56

4.4 Método que realiza o *download* do relatório. 58

4.5 Excerto de ficheiro TSL gerado automaticamente. 63

5.1 Ficheiro *Docker* para *RapiTest*. 67

5.2 Ficheiro *Docker Compose*. 69

5.3 Exemplo de teste para *Broken Object Level Authorization*. 78

5.4 Exemplo de teste para *Broken User Authentication*. 79

5.5 Exemplo de teste para *Lack of Resources and Rate Limiting*. 80

5.6 Exemplo de teste para *Mass Assignment*. 80

5.7 Exemplo de teste para *Improper Assets Management*. 81

5.8 Exemplo de teste para *Injection*. 81

5.9 Validação *Custom*. 82

Lista de Abreviaturas e Siglas

AMQP Advanced Message Queuing Protocol. 30, 48
API Application Programming Interface. xviii, xix, xxi, 1, 2, 3, 4, 5, 7, 8, 9, 10, 14, 15, 16, 17, 18, 19, 20, 22, 26, 27, 28, 30, 31, 33, 34, 38, 39, 42, 46, 47, 48, 49, 51, 52, 54, 57, 70, 71, 72, 73, 74, 75, 76, 77, 78, 83, 84

BFS Breadth-First Search. 8
BLOB Binary Large Object. 59

CML Câmara Municipal de Lisboa. 5, 70, 71, 72, 73, 74, 76
CRUD Create, Read, Update and Delete. 1, 76

DLL Dynamic-link library. 23, 27, 45, 46, 50, 68, 82
DoS Denial of service. 18
DSL Domain Specific Language. 5, 13, 19, 20, 83

GUI Graphical User Interface. 40, 42, 58, 71, 85

HTML HyperText Markup Language. 59
HTTP Hypertext Transfer Protocol. 3, 5, 8, 9, 19, 21, 23, 51, 55, 57, 61, 71, 72
HTTPS Hyper Text Transfer Protocol Secure. 67

IBM International Business Machines. 2
IDL Inter-parameter Dependency Language. 9

ISEL Instituto Superior de Engenharia de Lisboa. 5, 70

JAR Java Archive. 7

JSON JavaScript Object Notation. 3, 4, 14, 22, 27, 28, 30, 33, 39, 52, 53, 54, 56, 58, 59

OAS OpenAPI Specification. 2, 3, 4, 7, 8, 9, 10, 20, 21, 22, 38, 39, 40, 42, 48, 51, 52, 74, 83, 84, 85

ODG Operation Dependency Graph. 7

OWASP Open Web Application Security Project. 17

PGIL Plataforma de Gestão de Dados de Lisboa. 5, 70

REST Representational State Transfer. 1, 2, 5

SOAP Simple Object Access Protocol. 1, 5

SPA Single Page Application. 36

TSL Test Specification Language. xviii, xix, xxi, 13, 19, 20, 21, 27, 28, 30, 34, 35, 38, 40, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 58, 59, 60, 61, 62, 63, 71, 73, 74, 75, 76, 78, 79, 80, 81, 82, 83, 85

TXT Text Document File. 21, 45

URI Uniform Resource Identifier. 57

URL Uniform Resource Locator. 26, 27, 34, 38, 39, 70

XML Extensible Markup Language. 4, 14, 53

YAML YAML Ain't Markup Language. 3, 20, 27, 33, 39, 45, 49, 68

Glossário

Black-Box No contexto de testes para *software*, consiste no facto de desenvolver e executar testes sobre a funcionalidade de uma aplicação, sem ter qualquer conhecimento sobre a sua estrutura interna.. 2

Deploy Engloba todos os processos envolvidos em fornecer um novo *software* ou *hardware* corretamente no seu ambiente de produção, incluindo instalação, configuração, execução, teste e realização das alterações necessárias.. 2, 65, 66, 73

Endpoint Uma `API` é normalmente composta por múltiplos *endpoints*, sendo estes um local digital onde uma `API` recebe pedidos sobre um recurso específico do seu servidor. Um *endpoint* é normalmente um localizador uniforme de recursos (`URL`) que fornece a localização de um recurso no servidor.. 1, 2, 3, 4, 5, 33, 34, 35, 51, 52, 71, 72, 74, 76, 79, 84

Fuzzing Consiste numa técnica de teste de *software*, frequentemente automatizada ou semi automatizada, que envolve fornecer dados aleatórios, inválidos ou inesperados como entradas para programas de computador.. 8, 9, 15

Hash Uma função *hash* é um algoritmo que mapeia dados de comprimento variável para dados de comprimento fixo. Os valores retornados por uma função *hash* são chamados valores *hash*, códigos *hash*, somas *hash* (*hash sums*), *checksums* ou simplesmente *hashes*. Esta função é muito utilizada em palavras chave, de forma a não as guardar em claro.. 36

Web Sistema de informações ligadas através de hiperligações em forma de texto, vídeo, som e outras animações digitais que permitem ao usuário acessar uma infinidade de conteúdos através da internet. Para tal é necessário ligação à internet e um navegador onde são visualizados os conteúdos disponíveis.. 1, 5, 7, 11, 19, 23, 30, 66, 70, 71

White-Box No contexto de testes para *software*, consiste no facto de desenvolver e executar testes sobre a estrutura interna da aplicação, obtendo resultados como a cobertura total do código, o que naturalmente apenas é possível com acesso ao código fonte.. 2

1

Introdução

O termo *Application Programming Interface*, ou API, consiste numa interface de *software* com o objetivo de facilitar a comunicação entre diferentes componentes ou sistemas. Através desta interface o componente que invoca a operação não tem que se preocupar sobre procedimentos internos do sistema, apenas tem que saber como chamar a funcionalidade pretendida e como vêm estruturados os resultados. Hoje em dia, grande parte da comunicação entre diferentes sistemas é feita através da *web*, o que resultou na criação da *web* API. Existem hoje mais de 24.000 API listadas no *ProgrammableWeb* [48], API estas baseadas em diferentes protocolos ou estilos arquiteturais, sendo os mais populares *Representational State Transfer* (REST) e *Simple Object Access Protocol* (SOAP) [36]. No entanto, tendo em consideração os *websites* mais utilizados nos últimos anos [1], REST tornou-se a maneira *standard* de oferecer um serviço através da *web*.

API REST consistem num estilo arquitetural introduzido por Roy Fielding na sua tese de doutoramento no ano 2000 [25]. Estas são geralmente compostas por uma série de diferentes *endpoints*, cada um relativo a um certo tipo de informação. Através destes *endpoints* um utilizador consegue realizar diversas ações sobre esses recursos tais como criar, ler, alterar ou remover (*Create, Read, Update and Delete*, CRUD). Pela sua própria natureza, as API permitem que os utilizadores manipulem os dados do serviço que estão a utilizar, o que, se não for controlado, pode levar a falhas de segurança, possivelmente resultando no roubo, modificação ou mesmo remoção de dados sensíveis. Isto criou uma necessidade crescente para testar as API de forma a dificultar tais ataques maliciosos.

Muitas das grandes empresas fornecem API REST para acesso aos seus serviços, tais como: *Google Maps API*, *Twitter API*, *YouTube API*, *Facebook API*, *Amazon Product Advertising API*, entre outros. Muitas destas API são usadas globalmente e diariamente à escala global por milhões de utilizadores sem que estes se apercebam. Por exemplo, quando se pesquisa no *Google* pelo tempo numa certa zona, a informação meteorológica presente no pequeno *snippet* no topo da página provém de uma API da *International Business Machines* (IBM); ou quando se regista/*login* numa aplicação através da *Google* ou *Facebook* a sua respetiva API está a ser usada para esse efeito [39].

API REST, devido à sua natural separação de funcionalidades entre diferentes *endpoints*, encontram-se muitas vezes *deployed* em ambientes de execução distintos, como por exemplo contentores ou máquinas virtuais, onde facilmente e dinamicamente se podem alocar e desalocar recursos dependendo de que funcionalidades estão a ser mais ou menos utilizadas num certo momento [45]. Este tipo de desacoplamento dificulta, no entanto, a realização de testes sobre a API seguindo uma lógica *white-box*, onde se tem acesso ao código fonte, possibilitando testes de *code coverage*. Uma lógica *black-box*, onde não se tem acesso ao código fonte, depende apenas da interface disponibilizada para aceder à API. Além disso, muitas das vezes o código fonte não está disponível, sendo *black-box* a única lógica viável.

Métodos *black-box* dependem principalmente da especificação da API que estiverem a testar. A especificação descreve a API com grande detalhe e permite que tanto humanos como computadores descubram e compreendam os recursos do serviço sem ter acesso ao código fonte ou documentação adicional. API REST são frequentemente descritas usando a especificação *OpenAPI Specification* (OAS) [42]. Fundamentalmente, as técnicas de *black-box* derivam automaticamente os testes através da especificação da API. Para gerar esses testes, estes tiram partido de valores aleatórios e *default*, dados de ficheiros auxiliares, ou mesmo dados de respostas anteriores da API com a intenção de encontrar certas combinações de valores que resultam num código de *status* de resposta 5XX (erro do servidor). Estas ferramentas focam-se maioritariamente, ou mesmo exclusivamente, na deteção de códigos 5XX, e não validam o comportamento correto, em condições corretas [6, 7, 11, 31, 35, 61].

Embora as ferramentas analisadas com abordagem *black-box* mostrem resultados realtivamente promissores na deteção de erros, devido ao uso exclusivo da especificação da API, estas apresentam grandes limitações quanto à execução de outros tipos de teste, tais como testes de: 1) validação da resposta e do seu conteúdo, 2) *workflows* entre diferentes *endpoints*, 3) carga e latência, e, 4) identificação de vulnerabilidades de segurança.

Existem múltiplas ferramentas comercialmente disponíveis para realizar testes sobre uma API, no entanto grande parte destas não permite realizar testes sobre os quatro pontos fulcrais mencionados acima, sendo que as que os suportam, suportam apenas na sua versão *premium* com custo monetário associado. Muitas destas ferramentas, mesmo na versão *premium*, oferecem também pouca automatização, sendo necessário escrever e correr manualmente os testes pretendidos.

1.1 Problema

Para testar devidamente uma API, o primeiro passo fundamenta-se sempre na análise da sua especificação. No entanto, como foi mencionado na introdução, não existe uma forma *standard* de a descrever, sendo que uma das formas mais adotadas consiste na OAS. A OAS traduz-se num ficheiro JSON ou YAML [62] que contém informação sobre os *endpoints* da API e todos os métodos HTTP possíveis sobre esses (*Get*, *Post*, *Put*, etc...), como também parâmetros de *input* e *output*, formas de autenticação, entre outra informação.

Contudo, apesar de ser possível descrever e documentar uma API com grande detalhe através da especificação, muitas das vezes o problema surge de documentações incompletas ou desatualizadas. Assim, apesar de existirem ferramentas para gerar esta especificação em múltiplas linguagens como *Java* [30] ou *Python* [49], a especificação torna-se desatualizada após qualquer alteração sobre a API, o que implica também a atualização da especificação, algo que normalmente não é realizado. Naturalmente, se a especificação for feita de forma manual, muitas das vezes esta fica também incompleta. Devido a esta constante evolução da API, torna-se assim pertinente uma ferramenta que realize certos testes (e.g. testes de validação de esquema do corpo da resposta) num certo intervalo de tempo de forma a garantir que a especificação está consistente com a implementação.

A *Swagger*, criadora da OAS, disponibiliza diversas API de exemplo com a sua respetiva especificação para poderem ser feitos testes sobre os *endpoints* disponibilizados, o código fonte destas API não é público, impossibilitando uma abordagem *white-box*. Uma das API, *petstore* [46], simula uma loja de animais, onde é possível criar, obter, atualizar e remover animais de estimação da loja como também realizar certas operações sobre o utilizador em si. Um dos *endpoints* disponíveis nesta API, */pet*, consiste na criação de um novo animal de estimação através do método *Post*. A Listagem 1.1 apresenta um excerto da especificação sobre esse *endpoint* em formato YAML.

Listagem 1.1: *Petstore* - Excerto da especificação da API.

```
1  "/pet":
2      post:
3        summary: Add a new pet to the store
4        operationId: addPet
5          ...
6        responses:
7          '405':
8            description: Invalid input
9          ...
```

As linhas 6 a 8 apresentam que o pedido *Post* no */pet* pode gerar apenas um código dependendo do *payload* enviado no *body*. Porém, após múltiplos testes manuais enviando diversos *payloads* válidos e inválidos, foi concluído que muito provavelmente esse código nunca é retornado pelo servidor neste pedido. Não é possível de concluir com total certeza pois o código-fonte não é disponibilizado, sendo possível que haja alguma situação onde este pedido retorne 405, sendo esta uma limitação da OAS pois não é possível representar um exemplo de em que situação é que este ocorre, tendo no limite apenas uma descrição em texto. Foi no entanto observado que o código 405 *Method Not Allowed* é retornado quando é realizado um pedido ao mesmo *endpoint* com um diferente método não suportado, como por exemplo o *Get*.

Posto isto, tendo como base os testes realizados, foram na realidade observados três diferentes códigos retornados pelo pedido que não estão presentes na especificação, sendo estes:

- 200, quando o *payload* está bem formatado e contém os campos necessários,
- 400, quando o *payload* em XML não está bem formatado,
- 500, quando o *payload* em JSON não está bem formatado.

A especificação incompleta demonstra ser um problema significativo na automatização dos testes pois uma documentação incompleta torna-se num teste incompleto.

Tendo uma especificação completa da API, outro aspeto relevante é a automatização do processo de teste. Através da especificação é possível gerar testes automaticamente, e existem ferramentas que o fazem, apresentadas no capítulo 2. Todavia, testes mais específicos que requerem *inputs* específicos, verificações especificas, e inclusive o transporte de informação entre testes de forma a simular um *workflow*, necessitam de maior especificação, o que não existe atualmente de forma escrita.

Todos estes aspetos assentam sobre a necessidade de realizar testes, sendo estes tão simples como um pedido HTTP. A dificuldade surge no entanto, no grande número de pedidos que podem ser necessários de realizar, especialmente no que toca a testes de *stress*. Desta forma, realizar os testes pode tornar-se num processo muito dispendioso e demorado, onde uma solução de arquitetura desacoplada responsável pela realização dos testes se mostra uma solução ideal.

Devido à constante evolução tanto no desenvolvimento de API, como também nos protocolos HTTP, a solução desenvolvida que suporte estes diferentes tipos de testes deve também ter em consideração esse fator, facilitando a introdução de novos testes ou mesmo a possibilidade de testar não só API REST, como também API SOAP ou mesmo um novo tipo no futuro, garantindo a extensibilidade da aplicação.

Neste trabalho, apresenta-se a *RapiTest*, uma ferramenta de teste e validação para *web* API REST, disponibilizada como uma aplicação *web*. Este trabalho foi realizado no âmbito do protocolo "DADOS AO SERVIÇO DE LISBOA", celebrado entre a Câmara Municipal de Lisboa (CML) e o Instituto Superior de Engenharia de Lisboa (ISEL). Foi também desenvolvido ao longo de diversas reuniões com a CML tendo em conta o seu *feedback*, principalmente sobre a usabilidade sem ser necessário dominar conceitos demasiado técnicos, tendo como objetivo final o uso desta ferramenta para integrar dados na Plataforma de Gestão de Dados de Lisboa (PGIL) [55], dados estes vindos de API externas que devem obedecer a um certo esquema, que deve ser validado pela ferramenta. Esta ferramenta foi desenvolvida tendo em conta os seguintes objetivos principais:

- Realização de uma *Domain Specific Language* (DSL) que defina testes específicos que o utilizador pretende realizar sobre a API;
- Aplicação *web*, que recebendo a especificação da API e opcionalmente também a DSL, realize de forma automática e recorrente testes sobre a API;
- Possibilidade de criar a DSL de forma intuitiva e simples através da própria aplicação *web*;
- Análise da especificação da API de forma a validar que todos os *endpoints* estão a ser devidamente testados;
- Possibilidade de guardar os pedidos e verificações resultantes dos ficheiros de testes, de forma a facilitar e otimizar realizações futuras desses testes;
- Possibilidade de observar com detalhe na aplicação *web* os resultados de todos os testes realizados até ao momento de uma API;

- Utilização de tecnologias Código Aberto de forma a não existirem custos de licenciamento;
- Escalabilidade da arquitetura, separando logicamente e fisicamente diferentes funcionalidades chave da aplicação;
- Escalabilidade da linguagem de especificação de testes, de forma a facilitar a introdução de novas verificações.

Como resultado deste trabalho, foi submetido e aceite para publicação o artigo: *RapiTest: Continuous Black-Box Testing of RESTful Web APIs* na conferência internacional *CENTERIS - International Conference on ENTERprise Information Systems* [9]. A conferência encontra-se na sua décima quarta edição, e realizou-se em Lisboa, entre os dias 9 a 11 de Novembro de 2022.

1.2 Organização do Documento

A organização deste documento divide-se em 5 capítulos. No capítulo 2 são apresentadas ferramentas semelhantes ou com objetivos similares ao trabalho realizado, juntamente com artigos publicados relevantes ao problema descrito. No capítulo 3 são definidos objetivos e de que forma está planeado cumpri-los. No capítulo 4 é definido e descrito de que forma foi implementado o trabalho. No capítulo 5 são observados os resultados obtidos e por último, o documento termina com o capítulo 6 onde são apresentadas as conclusões de todo o trabalho realizado assim como algumas sugestões de trabalho futuro.

2

Estado da Arte

Neste capítulo é feita uma análise de trabalho relacionado com o tema tratado neste projeto, tendo por base diversos artigos propostos e ferramentas disponíveis atualmente. Todas as ferramentas comerciais apresentadas são gratuitas com o código-fonte disponível, sendo que algumas fornecem também uma versão paga, *premium*, com mais funcionalidades, versões estas não consideradas.

2.1 Artigos Propostos

RestTestGen *RestTestGen* é uma ferramenta *black-box* que gera testes automaticamente para *web* API, a partir da sua especificação em OAS, proposta por Viglianisi et al. [61]. Foi desenvolvida em *Java* e os JAR executáveis encontram-se disponíveis no *GitHub* [54].

Esta ferramenta introduz o conceito de *Operation Dependency Graph* (ODG), onde, através da especificação da API, esta tenta programaticamente deduzir dependências entre operações, de forma a construir uma ordem lógica de testes. Por exemplo, uma operação num *endpoint* pode retornar uma lista de identificadores, tendo depois outra que retorna um item especifico através desse identificador. Desta forma, a ferramenta consegue gerar testes com valores válidos, valores estes difíceis de adivinhar.

Em termos de validação, a ferramenta tem a capacidade de não só validar as respostas, como também o seu conteúdo, sendo apenas possível de validar se o esquema do

objeto devolvido confere com o esperado, não sendo possível introduzir validações *custom* definidas pelo utilizador da ferramenta. Testes de carga, segurança e autenticação também não foram suportados.

RESTler *RESTler* é uma ferramenta *black-box* que a partir de uma especificação em OAS, gera testes de forma a tentar encontrar erros numa API, proposta por Atlidakis et al. [7] na *Microsoft Research*. Foi desenvolvida em *Python* e encontra-se disponível no *GitHub* [53].

De forma a gerar as sequências de teste, *RESTler* tira partido de três diferentes algoritmos, cada um com a sua lógica e propósito diferente. O algoritmo principal, *Breadth-First Search* (BFS), gera múltiplas sequências de testes adicionando cada novo pedido válido a cada sequência existente, efetivamente gerando todas as sequências de pedidos possíveis. Outro algoritmo, *BFS Fast*, apenas adiciona um novo pedido válido a uma das sequências, sendo este um algoritmo mais rápido, não garante a criação de todas as sequências possíveis. Por último, o *Random-walk*, aleatoriamente seleciona uma sequência, adicionando um novo pedido, também aleatoriamente.

Após a geração das diferentes sequências de testes, *RESTler* tira partido dum ficheiro auxiliar dicionário, configurável pelo utilizador, que contém objetos para serem utilizados nos testes. *Fuzzing* [41] é depois realizado sobre diferentes valores de *input* provenientes do dicionário de forma a tentar encontrar erros por parte da API. Esta ferramenta, em termos de validação e verificação, tira partido apenas dos códigos HTTP, nomeadamente se alguma resposta retornou como código 5XX.

bBOXRT *bBOXRT* é uma ferramenta *black-box* que tirando partido duma especificação em OAS, gera múltiplos testes de forma a tentar encontrar erros, proposta por Laranjeiro et al. [31]. Foi desenvolvida em *Java* e encontra-se disponível no *GitLab* [5].

Esta ferramenta tem um grande foco em testes de *Fuzzing*, introduzindo 57 diferentes mutações sobre os valores de *input*, aplicáveis a valores do tipo numérico, *strings*, booleanos, datas, tempos, *arrays*, etc.. A execução começa por tentar gerar respostas válidas (códigos 2XX), onde os resultados retornados são reutilizados para testes futuros. Por outro lado respostas com código de erro (4XX ou 5XX) são tentadas novamente com diferentes mutações ou então totalmente descartadas.

A ferramenta não suporta qualquer tipo de validação de resposta e também requer intervenção manual para a análise dos resultados dos testes.

RESTest *RESTest* é uma ferramenta *black-box* que a partir de uma especificação em OAS e opcionalmente de um ficheiro de informação adicional, gera testes de forma a tentar encontrar erros numa API, proposta por Martin-Lopez et al. [35]. Foi desenvolvida em *Java* e encontra-se disponível no *GitHub* [52].

Esta ferramenta tem como peculiaridade a introdução do suporte de *inter-parameter dependencies*, que consiste na presença de dois ou mais parâmetros de *input*, onde o valor de um implica um certo valor específico noutro. Por exemplo, num dos pedidos na API do *YouTube*, se *videoDefinition* estiver presente, então o parâmetro *type* tem que ter o valor *video*. Isto torna-se especialmente relevante pois num estudo realizado anteriormente, Martin-Lopez notou a presença destas dependências numa grande parte de API globalmente utilizadas [34], o que levou à criação da *Inter-parameter Dependency Language* (IDL) [2], pois a especificação OAS não suporta a descrição destas dependências. Neste artigo, o autor descreve uma linguagem de fácil leitura por um humano e uma máquina, onde se encontra descrita a dependência entre parâmetros, finalizando com uma biblioteca que automatiza a leitura do documento. Desta forma, testes de *fuzzing* que normalmente resultavam em erro devido à falha de uma *inter-parameter dependencie* (testes sem grande propósito), são evitados.

A ferramenta não só suporta testes de validação do esquema e dos códigos HTTP, como também, a configuração de testes contínuos onde os resultados podem ser observados em tempo real. Contudo, não suporta qualquer tipo de testes de carga nem de segurança e autenticação.

2.2 Ferramentas Disponíveis

Dredd *Dredd* [10] é uma ferramenta em código aberto criada com o intuito de automatizar o processo de teste de uma API. Esta permite, através do ficheiro de especificação escrito no formato OAS ou *API Blueprint* [14], gerar um conjunto de testes para cada *endpoint*, para cada resposta possível do servidor. A ferramenta é mais indicada para API simples, onde verificações de estrutura da resposta são suficientes. Esta não tem qualquer suporte para testes de carga, segurança, e não suporta o transporte de informação das respostas entre testes, tornando impossível a realização de testes de *workflow*.

Postman *Postman* [47], lançado em 2014, é uma plataforma para construir, utilizar e testar API. É possível importar o ficheiro de especificação em OAS de forma a automaticamente criar todos os pedidos, no entanto, nenhum teste é criado automaticamente.

O *Postman* não fornece nenhuma verificação nativamente, sendo que todos os testes se baseiam em código *JavaScript* fornecido pelo utilizador. Este fornece também a possibilidade da criação de *workflows*, ou *collections* no *Postman*, incluindo a passagem de parâmetros entre pedidos. É também possível realizar testes de carga utilizando outras ferramentas como o *LoadView* exportando a coleção do *Postman* [32].

Posto isto, o *Postman* apesar de ser uma ferramenta muito completa, não está automatizada o suficiente e necessita de ferramentas externas para realizar todas as verificações pretendidas.

SoapUI *SoapUI* [57], lançado em 2005, é uma das ferramentas mais utilizadas para teste de `API`. Esta contém a possibilidade de importar a especificação da `API` em `OAS`, sendo que também não gera nenhum teste de forma automática. Contrariamente ao *Postman*, esta fornece diversas verificações nativamente, juntamente com a possibilidade de realizar verificações personalizadas através de *scripts*. Tem também a possibilidade de realizar *workflows* entre testes, passando o que for necessário entre pedidos. Fornece também a possibilidade de facilmente realizar testes de carga sobre pedidos individuais, ou mesmo sobre um *workflow*, e a realização de testes de segurança sobre cada pedido.

Sendo esta a ferramenta de teste mais completa no mercado (suportando os quatro tipos principais de testes) apresenta, no entanto, problemas de automatismo na sua criação, necessitando sempre de escrita manual.

2.3 Resumo

A Tabela 2.1 apresenta uma sucinta comparação dos aspetos principais de cada ferramenta.

Tabela 2.1: Características das ferramentas analisadas.

Características	Tipos de teste suportados					Automatização Testes		Acessibilidade
	Validação	Workflow	Stress	Segurança	Custom	Gerar	Recorrentes	
RestTestGen	Limitado	Não	Não	Não	Não	Sim	Não	Localmente
RESTler	Limitado	Não	Não	Não	Não	Sim	Não	Localmente
bBOXRT	Não	Não	Não	Não	Não	Sim	Não	Localmente
RESTest	Limitado	Não	Não	Não	Não	Sim	Sim	Localmente
Dredd	Limitado	Não	Não	Não	Sim	Sim	Não	Localmente
Postman	Sim	Sim	Não	Não	Sim	Não	Não	Localmente
SoapUI	Sim	Sim	Sim	Sim	Sim	Não	Não	Localmente
RapiTest	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Web app

Tendo como objetivo comparar todas as ferramentas mencionados nas secções anteriores, podemos concluir que o *SoapUI* é o mais organizado, abrangendo grande parte dos tipos de validação desejados. Destaca-se o *Postman*, que apesar de não suportar todas as características consideradas relevantes, cumpre o objetivo de suportar validações *custom*, de *workflow* e de *stress* apesar de necessitar de ferramentas externas. Estas ferramentas não suportam no entanto a criação de testes automaticamente, obrigando a ser o utilizador a criá-los manualmente, o que implica maior compreensão sobre todos os elementos em causa. Isto contrasta com os artigos propostos, que necessitam apenas da especificação para o seu correto funcionamento, no entanto a esse mesmo fator, encontram-se muito limitados nos tipos de teste que suportam.

Posto isto, *RapiTest* visa ser uma ferramenta de teste completa, suportando os quatro tipos principais de teste, como também testes personalizados para não se encontrar limitada. Suporta também a geração de testes de forma automática, e também realiza esses testes de forma recorrente. Apresenta no entanto, como diferença e objetivo principal, o facto de fornecer as suas funcionalidades através de uma aplicação *web*, não necessitando de qualquer instalação local.

3

Abordagem

Neste capítulo é especificada a arquitetura para concretizar todos os objetivos propostos. Inicialmente, são apresentados os diferentes tipos de teste como também a arquitetura que os suporta nas secções 3.1 e 3.2 respetivamente. Seguidamente, na secção 3.3, é apresentada a `DSL` *Test Specification Language* (`TSL`) proposta para resolução dos problemas, que especifica a utilização do descritor de testes. Finalmente, na secção 3.4, são descritos os casos de utilização suportados pela aplicação, terminando com a arquitetura proposta para suportar os requisitos inerentes da aplicação na secção 3.5.

3.1 Tipos de Teste

Como foi mencionado na introdução, existem diversos tipos de testes, cada um com o seu propósito distinto. Estes podem, no entanto, ser agrupados em quatro categorias, cada uma detalhada nas secções seguintes, nomeadamente testes de validação na secção 3.1.1, testes de lógica de utilização (*workflow*) na secção 3.1.2, testes de stress na secção 3.1.3 e testes de segurança na secção 3.1.4.

3.1.1 Validação

Os testes de validação são o tipo de teste mais comum. Estes testes visam validar o correto funcionamento de um certo pedido, sobre certo *input*. Utilizando como mesmo

exemplo a loja de animais, *petstore*, apresentada anteriormente, a Listagem 3.1 apresenta a descrição do *endpoint /pet* com método *Post*, desta vez completado e corrigido com a resposta de código 200 e a troca de 405 para 400 num *input* inválido, comparativamente com o apresentado na secção 1.1.

Listagem 3.1: *Petstore* - Excerto da especificação da API completado.

```
"/pet":
  post:
    tags:
    - pet
    summary: Add a new pet to the store
    operationId: addPet
    consumes:
    - application/json
    - application/xml
    produces:
    - application/json
    - application/xml
    parameters:
    - in: body
      name: body
      description: Pet object that needs to be added to the store
      required: true
      schema:
        "$ref": "#/definitions/Pet"
    responses:
      '200':
        description: successful operation
        schema:
          "$ref": "#/definitions/Pet"
      '400':
        description: Invalid input
```

Analisando a especificação é possível de idealizar oito testes diferentes para este pedido, através das combinações de tipos de linguagem como *input*, *output* e dos códigos de resposta. Um dos testes seria, como exemplo, *input* no formato JSON, *output* no formato XML e código resultante 400. Se a API disponibilizar também mais do que um servidor as combinações aumentam.

Manipular o tipo de linguagem com que a resposta vem formatada é relativamente banal, sendo possível de forçar qual o tipo de formato pretendido através do *header Accept:application/json* ou *Accept:application/xml*. É, no entanto, difícil de automatizar que *input* deve ser utilizado de forma a obter o código pretendido. Muitas das ferramentas

propostas nos artigos tentam gerar *inputs* de forma dinâmica tirando partido de dicionários externos com valores de exemplo reais, respostas obtidas por pedidos válidos anteriores ou simplesmente valores aleatórios do mesmo tipo, com o intuito de obter valores de *inputs* válidos com códigos de resposta 2XX.

Após obterem um conjunto de valores de *input* válidos, estas aplicam *fuzzing* a estes valores de forma a forçar erros na `API`. *Fuzzing* consiste no ato de intencionalmente e, normalmente, de forma automática, alterar campos nos dados de forma a torná-los inconsistentes com a especificação. Por exemplo, uma forma comum de *fuzzing* pode ser enviar um inteiro num campo que se esperava uma *string* ou vice-versa. Desta forma, é esperado que a `API` retorne um código no *range* dos 4XX devido à mal formação do *input*, e grande parte das vezes é esse o caso. No entanto, alguma combinação especifica de *fuzzing* pode provocar erros no range do 5XX, que podem tornar-se importantes falhas de segurança.

Optou-se pelo uso de um ficheiro auxiliar de dicionário externo, que contém valores de *input* para serem utilizados em certos testes, fornecido pelo utilizador, pois a ferramenta desenvolvida tem mais como foco a validação do correto funcionamento da `API` sobre certas condições específicas e não a procura extensiva de possíveis falhas no servidor que provoquem erros 5XX.

Posto isto, verificações exclusivamente ao código da resposta não são, e não devem ser, o único método de validação do correto funcionamento da `API`, pois, muitas das vezes, as respostas têm um corpo, que deve ser validado também. Como é possível de observar na especificação da `API` (Listagem 3.1), linhas 23 e 24, a resposta de código 200 deve conter no seu corpo um objeto com o mesmo esquema que o especificado. Desta forma, deve também ser verificado se o esquema recebido é igual ao especificado. Existem diversos tipos de validações possíveis, sendo que a Tabela 3.1 apresenta todas as validações nativas suportadas diretamente pela aplicação, juntamente com o seu *input* e uma descrição sobre o seu funcionamento.

Tabela 3.1: Validações fornecidas nativamente.

Validação	Input	Descrição
Code	Código HTTP	Compara o código com o recebido
Contains	String	Verifica se a string está presente no corpo da resposta
Count	String;Inteiro	Verifica se a string está presenta # vezes
Schema	Esquema	Compara o esquema da resposta, em json ou xml
Match	Path;Valor	Compara o valor no Path, em json ou xml

Existem, no entanto, cenários demasiado específicos, onde nenhuma das validações nativas satisfaz o teste que o utilizador deseja realizar. Desta forma, é também possível fornecer o teste desejado através de um contrato específico implementado.

3.1.2 Lógica de Utilização

Testes de lógica de utilização tentam simular o processo esperado por um utilizador, de forma a verificar o correto funcionamento intra-pedidos, um *workflow*. Tendo como exemplo novamente a loja de animais *Petstore*, a Figura 3.1 apresenta uma sequência lógica de pedidos, começando por adicionar um novo *pet* à loja, confirmar que foi adicionado corretamente com o *Get* pelo `id`, atualizar certos campos nele e, por fim, removê-lo da loja.

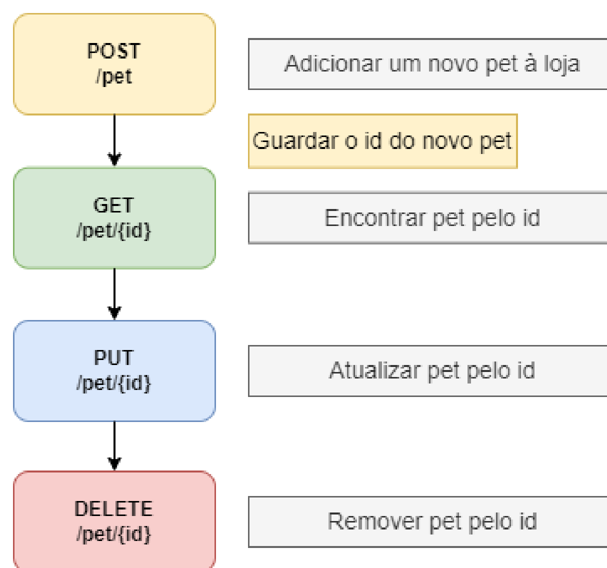


Figura 3.1: *Workflow* para *Petstore*.

A realização de *workflows* tem como peculiaridade a necessidade de guardar valores de certas respostas para futuros pedidos, normalmente chaves primárias de certos objetos, necessárias para realizar operações especificas sobre esse objeto. A Figura 3.1 demonstra essa necessidade através do campo `id`, campo este retornado na resposta do primeiro pedido, necessário para todos os pedidos futuros.

3.1.3 Carga

Testes de Carga desempenham um papel importante na identificação da velocidade, estabilidade e confiabilidade de uma `API`. Este tipo de testes, quando utilizados desde o início do desenvolvimento, ajudam a evitar tempos de espera demasiado elevados ou

mesmo, no pior caso, *crash* total. Estes testes têm como objetivo encontrar o ponto de rutura de uma API sobre carga extremamente alta durante um certo período de tempo. Desta forma é possível de identificar áreas especificas da API onde é possivelmente necessário aumentar os recursos para que a API continue a funcionar devidamente em alturas de elevado tráfego.

Estes testes baseiam-se principalmente num parâmetro, o tempo entre um pedido e a sua resposta. Tirando partido de múltiplos fios de execução, são enviados inúmeros pedidos à API de forma a avaliar o seu desempenho sobre carga. Podem ser realizados inclusivamente sobre *workflows* de forma a tentar perceber qual dos pedidos dentro de um *workflow* necessitam de mais recursos. Múltiplos servidores permitem também a execução de testes de carga mais sofisticados, não estando limitado às capacidades de uma única máquina.

3.1.4 Segurança

Grande parte das vezes, o bem mais valioso de uma empresa são os seus dados. Ameaças a esses dados devem ser identificadas e eliminadas para que estes não fiquem em risco [3]. De todos os componentes que formam uma aplicação, as API fornecem o ponto de acesso mais direto para ataques maliciosos onde uma vulnerabilidade na API afeta não só a própria aplicação, como todos os aplicativos que dependem dela.

Tratando-se de questões de segurança extremamente importantes, a realização de testes lógicos e funcionais não são suficientes para encontrar vulnerabilidades. Devem ser realizados testes que realmente simulem os tipos de ataques maliciosos que um atacante possa executar. O *Open Web Application Security Project* (OWASP) é uma fundação sem fins lucrativos que trabalha para melhorar a segurança de *software*. O OWASP *Top 10* é um documento padrão que representa um consenso sobre os riscos de segurança mais críticos para aplicações *web* [44]:

1. *Broken Object Level Authorization*, consiste na alteração de um campo `id` presente na chamada à API do seu próprio recurso, por um outro `id` de um recurso de outro utilizador onde a falta de verificações de autorização adequadas permite que o atacante aceda aos recursos de outros utilizadores, ilustrado pela Figura 3.2.

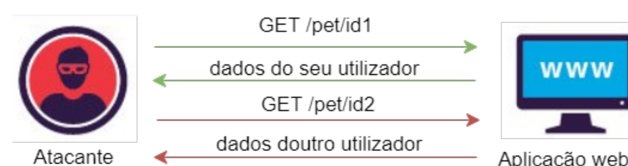


Figura 3.2: Exemplo de *Broken Object Level Authorization*.

2. *Broken User Authentication*, é um tipo de vulnerabilidade que os atacantes exploram onde se fazem passar por utilizadores legítimos, normalmente associado a má gestão de sessão e de credenciais, tirando partido de *tokens* de sessão sequestrados ou de credenciais de *login* roubadas.

3. *Excessive Data Exposure*, consiste no facto de uma resposta de um pedido à API retornar mais campos do que o necessário, campos estes que podem conter informação crítica e privada. Por exemplo, um pedido que mostra ao cliente apenas os nomes de todos os utilizadores, analisando o corpo da resposta este contém o objeto inteiro do utilizador, com informação privada presente.

4. *Lack of Resources and Rate Limiting*, é um tipo de vulnerabilidade onde a API não impõem quaisquer restrições ao tamanho ou número de recursos que podem ser solicitados pelo utilizador, o que leva a impactos no desempenho do servidor e à negação de serviço (*Denial of service*, DoS), mas também deixa a porta aberta para falhas de autenticação, como ataques de força bruta.

5. *Broken Function Level Authorization*, consiste no facto de utilizadores conseguirem ter acesso a funções reservadas a outros utilizadores. Estes problemas surgem em API com hierarquias, grupos e administradores.

6. *Mass Assignment*, consiste na tentativa de manipulação de propriedades de objetos devido a problemas como acesso ao código fonte ou construtores vazios. Por exemplo, num pedido para criar um utilizador, utilizador este que provavelmente tem um campo que diz se é administrador ou não, uma vulnerabilidade comum é adicionar aos campos *isAdmin=true* de forma a manipular os dados.

7. *Security Misconfiguration*, consiste no facto de, através de modificação de certos parâmetros ou falhas em configurações incompletas, o servidor responda com erros de exceção detalhados com informação do *stack*. Essa informação pode ser critica para um atacante sendo possível de obter uma compreensão da infraestrutura da API, bem como o próprio código, a fim de descobrir outras vulnerabilidades potencialmente exploráveis.

8. *Injection*, são ataques que visam comprometer a base de dados de forma a tentar manipular informação privada. Estes ataques visam executar comandos na base de dados de forma abusiva, normalmente através de manipulação de *strings* nos parâmetros do pedido.

9. *Improper Assets Management*, consiste em tentar encontrar versões descontinuadas da API, versões estas que estão menos protegidas contra atacantes, de forma a

encontrar vulnerabilidades.

10. *Insufficient Logging and Monitoring*, não sendo realmente possível de testar, consiste no facto de muitas destas API não terem *logs* suficientemente completos, resultando numa deteção de ataques muito tardia.

Tirando partido da linguagem TSL, apresentada na secção 3.3, todos os testes de segurança que não requerem acesso ao código fonte são possíveis de realizar, ou seja apenas o teste de *Insufficient Logging and Monitoring* não é suportado. No entanto, nenhum é fornecido de forma explícita e automática, necessitando todos de escrita manual no ficheiro TSL. Foi, contudo, tido em consideração no desenho e implementação da arquitetura a facilidade de acrescentar suporte explícito a estes testes em trabalho futuro.

3.2 Arquitetura de Suporte aos Testes

A Figura 3.3 apresenta a arquitetura que suporta os quatro tipos diferentes de testes.

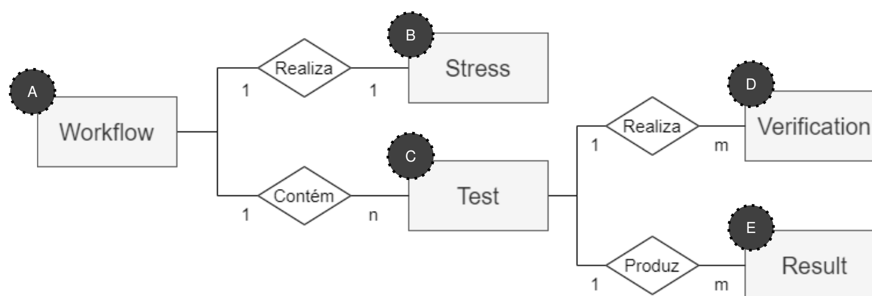


Figura 3.3: Arquitetura de suporte aos testes.

Esta contém cinco blocos principais, começando pelo *Workflow* (A). Este bloco é necessário de forma a suportar testes de lógica de utilização, contém um ou mais *Test* (C) de forma ordenada (simulando o *workflow*), e também um teste de carga (*Stress* B). De seguida, um *Test*, que contém toda a informação necessária para realizar o pedido HTTP, inclui uma ou mais verificações a realizar (*Verification* D) e todas as verificações produzem um *Result* (E), compilando todos os resultados no próprio teste.

3.3 *Test Specification Language*, **TSL**

Nesta secção é apresentada a DSL *Test Specification Language* (TSL), uma linguagem textual desenvolvida para a especificação de testes e verificações para *web* API. Especificamente, esta linguagem foi desenvolvida com o objetivo de suportar os quatro

diferentes tipos de testes apresentados anteriormente. Esta DSL baseia-se em YAML e teve como inspiração a linguagem de especificação da API OAS, de forma a facilitar a sua compreensão tanto por humanos como por máquinas. A Listagem 3.2 apresenta um exemplo de ficheiro TSL para a API *Petstore*, neste caso um *workflow* composto por dois testes, o primeiro adiciona um novo *pet* à loja e o segundo atualiza-o.

Listagem 3.2: Exemplo da descrição de um teste em TSL.

```
1  - WorkflowID: cu_pet
2    Stress:
3      Count: 40
4      Threads: 5
5      Delay: 0
6    Tests:
7    - TestID: createPet
8      Server: "https://petstore3.swagger.io/api/v3"
9      Path: "/pet"
10     Method: Post
11     Headers:
12       - Content-Type:application/json
13       - Accept:application/json
14     Body: "$ref/dictionary/petExample"
15     Retain:
16       - petId#$.id
17     Verifications:
18       - Code: 200
19         Contains: id
20         Count: doggie#1
21         Schema: "$ref/definitions/Pet"
22         Match: $.name#doggie
23   - TestID: updatePet
24     Server: "https://petstore3.swagger.io/api/v3"
25     Path: "/pet/{petId}"
26     Query:
27       - name=doggie
28       - status=sold
29     Method: Post
30     Headers:
31       - Accept:application/xml
32     Verifications:
33       - Code: 200
34         Custom: ["CustomVerification.dll"]
```

WorkflowID Cada ficheiro TSL necessita de incluir no mínimo um *workflow*. O *workflow* necessita de um id, que tem que ser único, neste caso, na linha 1, podemos observer o id definido como *cu_pet*. Um *worklfow* é obrigado a conter um ou mais testes e opcionalmente um teste de carga. Todos os testes dentro de um *workflow* são garantidamente executados sequencialmente de forma a suportar o transporte de informação da resposta de um pedido para outro.

Stress Um *workflow* pode conter apenas um teste de carga, este contém três campos que devem ser fornecidos, nomeadamente:

1. *Count*, define o número de vezes que o *workflow* irá ser executado na sua totalidade ;
2. *Threads*, o número de fios de execução por onde as execuções vão ser divididas (*Count* 40 e *Threads* 5, significa que cada fio de execução irá realizar 8 *workflows* completos);
3. *Delay*, define o atraso, em milissegundos, entre cada execução total de *workflow*, útil para prevenir erros como 429: "*Too Many Requests*".

Tests É possível definir um ou mais testes dentro de cada *workflow*, sendo que a sua criação começa por definir um id único, *TestID*, neste caso, temos na linha 7 o teste *createPet* e na linha 23 o *updatePet*. Após o id seguem-se três campos necessários para a realização de um pedido HTTP, nomeadamente o *Server*, *Path* e *Method* (linhas 8 a 10). É também naturalmente possível de especificar os *Headers* seguindo uma lógica de chave/valor (linhas 11 a 13), e de forma semelhante as *Queries* (linhas 26 a 28).

Em relação aos pedidos que necessitam de corpo, estes podem ser definidos diretamente no ficheiro TSL, todavia, estes podem ser objetos de grande dimensão, prejudicando a legibilidade do ficheiro em si. É no entanto possível, através de um único ficheiro dicionário auxiliar, definir o objeto nesse ficheiro, mantendo apenas uma referência para esse objeto no ficheiro TSL. Esta funcionalidade pode ser observada na linha 14, *$ref/dictionary/petExample*, onde, seguindo uma convenção semelhante à usada pela OAS, é definida uma referência para o dicionário através do prefixo *$ref/dictionary/*, para um objeto com id de *petExample*.

Dicionário O dicionário consiste num ficheiro auxiliar no formato TXT, formado por um identificador único para cada objeto seguido do seu valor. A Listagem 3.3 apresenta um exemplo de um objeto no ficheiro dicionário, neste caso com o identificador

único *petExample*, seguido do valor em JSON para ser utilizado num teste.

Listagem 3.3: Exemplo de ficheiro dicionário.

```
dictionaryID:petExample
{
  "id": 10,
  "name": "doggie",
  "category": {
    "id": 1,
    "name": "Dogs"
  },
  "status": "available"
}
```

A usabilidade do ficheiro foi também estendida para especificar esquemas de objetos, caso estes não se encontrem presentes na especificação da API, ou caso esta se encontre desatualizada.

Verifications Cada *Test* pode ter diversas validações, sendo que apenas a validação do código é obrigatória. É possível de observar nas linhas 17 a 22, todas as verificações nativas da aplicação que vão ser aplicadas sobre o teste *createPet*, nomeadamente:

- Verificação que o código retornado foi 200;
- Verificação que no corpo da resposta está presente a *string*: `id`;
- Verificação que no corpo da resposta está presente a *string*: *doggie* exatamente uma vez;
- Verificação que no corpo da resposta, o esquema deste é equivalente ao que está definido na OAS com identificador *Pet*;
- Verificação que no corpo da resposta está presente no JSON *path $.name* o valor *doggie*;

Validações Personalizadas Um dos requisitos consiste no suporte a verificações externas, personalizadas, vindas do utilizador, de forma a não estar limitado ao que a aplicação fornece nativamente. Para suportar este requisito foi definido um contrato que o utilizador deve implementar, neste caso a interface *Verification*, apresentada na Figura 3.4.

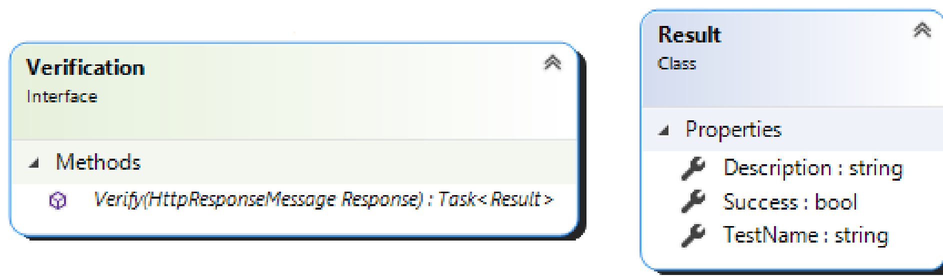


Figura 3.4: Contrato para validações externas.

A interface contém apenas um método, *Verify*, que recebe a resposta do pedido `HTTP` e retorna um *Result*, um simples objeto que descreve o resultado. Implementando a interface e gerando o ficheiro `DLL`, o utilizador consegue realizar uma verificação personalizada. Esta funcionalidade foi apenas realizada como prova de conceito, sendo que naturalmente o *upload* de ficheiros `DLL` pode consistir numa falha grave de segurança.

3.4 Casos de Utilização Suportados

A aplicação *web* suporta dois atores, um utilizador autenticado e um utilizador não autenticado, tendo ambos acesso a diversas ações sendo que grande parte das funcionalidades requerem que o utilizador esteja autenticado. A Figura 3.5 apresenta os casos de utilização para um utilizador não autenticado.

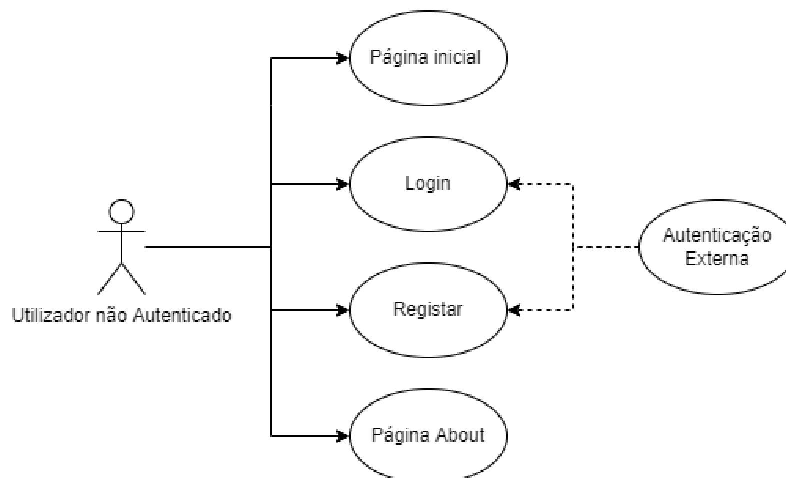


Figura 3.5: Casos de utilização - Não autenticado.

Registar. A autenticação de um utilizador lida com dados sensíveis que nunca devem ser comprometidos. No entanto, pouco se pode fazer em termos de segurança se

o utilizador utiliza uma palavra-passe simples como *pass123* que pode ser facilmente comprometida através de um simples ataque força bruta [8]. Logo, o utilizador é forçado a utilizar diferentes tipos de caracteres e um tamanho mínimo para dificultar tais ataques quando se regista na aplicação. Para um utilizador se registar este tem também que fornecer o seu *email* sendo que dois utilizadores não podem utilizar o mesmo endereço.

Login. Para um utilizador se autenticar ele tem que fornecer o seu *email* e palavra-passe. O sistema verifica as credenciais do utilizador, se este fornecer credenciais válidas, passa a estar autenticado e é redirecionado para a página onde se encontrava. No caso de falha de autenticação, é apresentada uma mensagem de erro e o utilizador pode repor as suas credenciais.

Autenticação Externa. Um utilizador também se pode autenticar ou registar através de um serviço externo, nomeadamente o *Google* ou o *Facebook* através da tecnologia *OAuth 2.0* [40]. O utilizador é redirecionado para a página do respetivo serviço onde tem que se autenticar com uma conta válida desse serviço. Se se autenticar com sucesso, o utilizador é redirecionado novamente para a aplicação, autenticado se já tinha conta ou, então, para uma página onde apenas tem que fornecer um nome de utilizador completando o registo. Naturalmente, nesta situação a palavra-passe não é necessariamente forte, estando limitada aos requisitos de palavra-passe dos respetivos serviços.

Página Inicial. Na página inicial da aplicação, é apresentada uma série de imagens ao utilizador não autenticado que ilustram o uso da aplicação esperado. Se o utilizador está autenticado, outro caso de utilização é apresentado. Na página *About*, a aplicação e as suas funcionalidades são apresentadas em maior detalhe.

A Figura 3.6 apresenta os casos de utilização para um utilizador autenticado.

Aceder aos dados da sua conta. Um utilizador autenticado ao clicar no seu *email* na barra de acesso é redirecionado para uma página onde tem acesso a funcionalidades relacionadas com a sua conta descritas nos quatro casos abaixo.

Alterar palavra-passe. Um utilizador autenticado tem também a possibilidade de alterar a palavra-passe. Este deve fornecer a sua palavra-passe atual para verificar novamente que é o próprio utilizador a tentar alterá-la. Ele deve então fornecer a nova

palavra-passe duas vezes de forma a prevenir contra eventuais erros de escrita na nova palavra-passe. Se a palavra-passe antiga estiver correta e as duas novas palavra-passes forem iguais a palavra-passe do utilizador é então alterada, se alguma dessas verificações falhar aparece uma mensagem de erro.

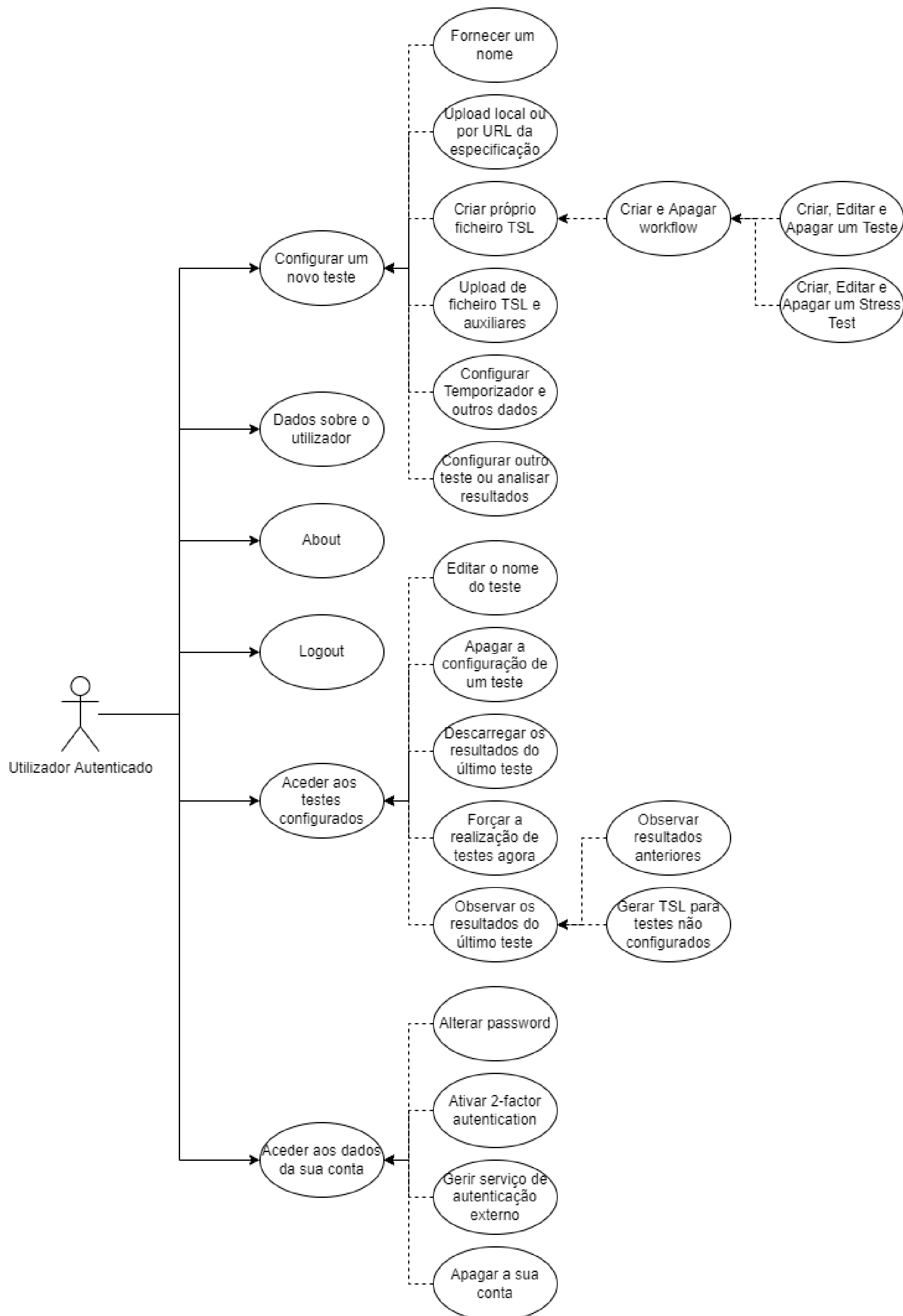


Figura 3.6: Casos de utilização - Autenticado.

Ativar *Two-factor-authentication*. Um utilizador autenticado tem a possibilidade de ativar *Two-factor-authentication* de forma a aumentar a segurança e integridade da sua conta [60]. A aplicação fornece ao utilizador uma chave ou um código `QR` que o utilizador deve utilizar numa aplicação como *Microsoft Authenticator* para *Android* ou *iOS* ou *Google Authenticator* para os mesmos sistemas operativos. Uma chave é dada ao utilizador por uma dessas aplicações que deve ser colocado na aplicação completando o caso de utilização.

Gerir serviço de autenticação externo. Um utilizador pode adicionar à sua conta um serviço de autenticação externo de forma a facilitar autenticações futuras. Este caso funciona similarmente ao descrito acima em **Autenticação Externa**.

Apagar a sua conta. Um utilizador tem a possibilidade de apagar todos os dados e ficheiros associados à sua conta. Este processo é irreversível e é pedida novamente a palavra-passe ao utilizador para ter a certeza que não foi pedido por engano.

***Logout*.** Um utilizador autenticado pode terminar a sua sessão fazendo, sendo redirecionado para a página inicial.

Dados sobre o utilizador. Se o utilizador estiver autenticado, a página inicial contém um historial das últimas ações, relacionadas com os últimos testes realizados, de forma a relembrar o uso da aplicação a quem não aceda há vários dias contendo também um *link* para ir diretamente aos resultados desses testes. Contém, ainda, alguns dados relevantes sobre o utilizador como quantos testes tem configurados, um *roadmap* dos próximos testes e quando foi o último *login*.

Configurar um novo teste. Este caso de utilização permite a um utilizador autenticado iniciar um processo de cinco passos para configurar um novo teste a uma `API`:

1. **Fornecer um nome**. Fornecer um nome para o teste, normalmente o da `API`, sendo que pode ser um nome repetido noutro teste pois este fica associado à conta do utilizador através de outro identificador. Este nome não pode conter caracteres especiais (e.g., `!@#$%^` etc...) com o intuito de proteger contra ataques de *Cross Site Scripting* e *Sql Injection* [26];

2. ***Upload* local ou por `URL` da especificação da `API`**. Fornecer, por *drag'n'drop* a especificação numa zona dedicada na aplicação, ou fazendo *browse* localmente.

A zona de *drag'n'drop* apenas aceita ficheiros JSON ou YAML. É também possível fornecer a API através de um URL. Após o *upload*, a especificação é verificada e caso não esteja bem formatada, uma janela de erro aparece e o utilizador deve fornecer novamente outra especificação válida;

3. **Criar próprio ficheiro TSL** ou **Upload de ficheiro TSL e auxiliares**. Neste passo o utilizador tem duas hipóteses principais, uma delas consiste em criar o próprio ficheiro TSL através da interface gráfica. Esta suporta a criação e remoção de *workflows*, a criação, edição e remoção de testes, incluindo verificações e, ainda, a criação, edição e remoção de testes de carga.

   Por outro lado, se o utilizador já tiver os ficheiros TSL, e caso necessário o ficheiro dicionário e DLL este pode realizar o seu *upload* diretamente por *drag'n'drop* ou fazendo *browse* localmente. O utilizador pode também ignorar este passo, o que implica que apenas os testes gerados automaticamente, seguindo a especificação da API, vão ser realizados;

4. **Configurar temporizador e outros dados**. Neste passo, o utilizador deve escolher a frequência para a realização dos testes como também se deseja correr os testes imediatamente após a configuração. Outra opção é a realização dos testes gerados automaticamente, no entanto se o utilizador decidiu ignorar o passo 3, esta opção é obrigatória, pois são os únicos testes que vão ser realizados;

5. **Configurar outro teste ou analisar os resultados**. Neste passo, o utilizador pode decidir realizar outra configuração recomeçando o processo de cinco passos, ou então pode analisar os resultados, sendo redirecionado para o caso de utilização apresentado abaixo.

Aceder aos seus testes configurados. Este caso permite ao utilizador aceder a todos os seus testes configurados. Ao clicar num, a informação sobre o teste é mostrada ao utilizador, nomeadamente o número de validações falhadas do último teste, o número de avisos sobre este, quando é que o teste foi realizado como também quando irá ser o próximo teste. Caso algum dos ficheiros fornecidos não esteja bem configurado, apenas as mensagens de erro são mostradas. Este caso também dá acesso a cinco outros casos que vão ser explicados nos próximos parágrafos.

Editar o nome do teste. Ao clicar num teste é possível editá-lo. Uma janela dá *pop-up* onde o utilizador pode fornecer outro nome para o teste.

Apagar a configuração de um teste. Ao clicar num teste é possível apagá-lo. Um alerta dá *pop-up* pedindo confirmação ao utilizador. Em caso de confirmação, o teste e todos os seus resultados são apagados.

Descarregar os resultados do último teste. Ao clicar num teste que já foi bem sucedido é possível fazer *download* desses resultados recebendo o respetivo ficheiro `JSON`.

Forçar a realização de testes agora. Ao clicar num teste, é possível forçar a realização de testes imediatamente, não tendo que esperar pela data do próximo teste.

Observar os resultados do último teste. Este caso deixa observar em maior detalhe os resultados de um teste, como quais verificações, em quais testes de quais *workflows* é que falharam e sucederem, gráficos temporais sobre os testes de carga e quais *endpoints* da `API` não foram testados. Este caso de utilização dá acesso a outros dois casos:

1. **Observar resultados anteriores**. Neste passo, o utilizador pode observar os resultados de qualquer outro teste realizado anteriormente;
2. **Gerar `TSL` para testes não configurados**. Neste passo, o utilizador pode gerar um ficheiro `TSL` para todas as diferentes combinações de testes que não foram realizados diretamente pelo ficheiro `TSL` fornecido na configuração.

3.4.1 Estrutura da Interface Visual

Todas as páginas da aplicação têm, no topo da página, uma barra de navegação com todas as funcionalidades principais da aplicação para fácil navegação entre elas. A Figura 3.7 apresenta a barra de navegação para um utilizador não autenticado.

Figura 3.7: Barra de navegação.

Cada uma das páginas engloba um ou mais dos casos de utilização mencionados acima. Estas páginas são:

1. *Home*, página inicial da aplicação que contém os casos **Página Inicial** e **Dados sobre o utilizador**;

2. *Setup Test*, onde o utilizador pode realizar a configuração de um novo teste que contém o caso **Configurar um novo teste** e os seus casos filhos;

3. *Monitor Tests*, onde o utilizador pode realizar operações sobre os testes configurados que contém o caso **Aceder aos seus testes configurados** e os seus casos filhos;

4. *Register*, para registar um novo utilizador que contém o caso **Registar** e **Autenticação Externa**;

5. *Login*, para autenticar um utilizador registado que contém o caso **Login** e **Autenticação Externa**;

6. *About*, página com informação sobre a aplicação que contém o caso **About**;

7. O *email* de um utilizador após estar autenticado (não presente na figura), para alterar dados da sua conta que contém os casos **Aceder aos dados da sua conta** e os seus casos filhos.

Os detalhes sobre como foram implementadas estas páginas incluindo os seus casos de utilização estão descritos no próximo capítulo, capítulo 4.

3.5 Arquitetura

O primeiro passo da abordagem foi identificar o fluxo da ferramenta *RapiTest* necessário para a realização do pretendido, apresentado na Figura 3.8.

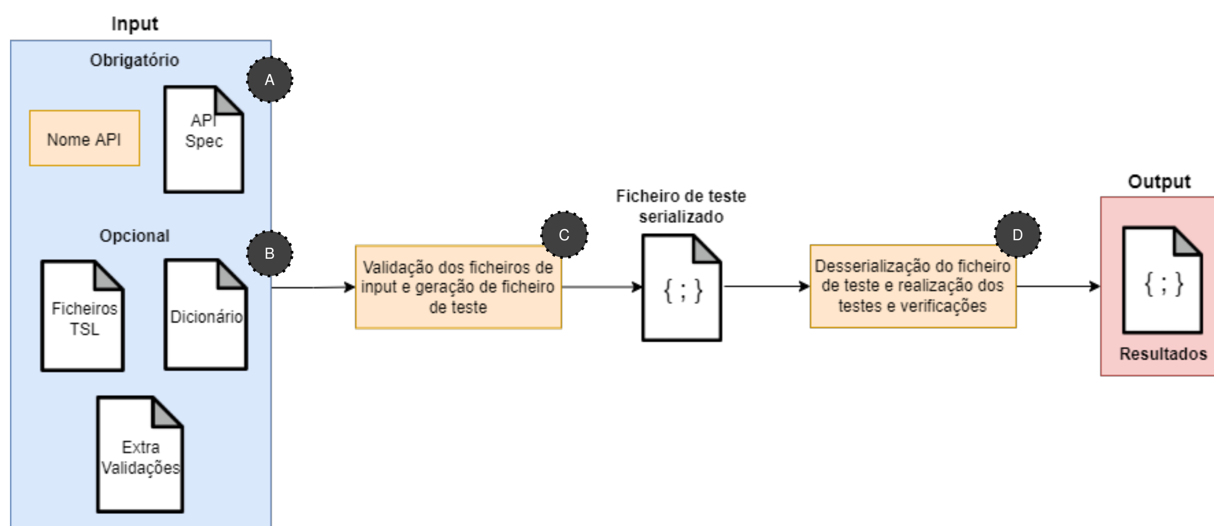


Figura 3.8: *RapiTest workflow*.

(A) Representa o mínimo necessário fornecido pelo utilizador para o funcionamento da ferramenta. Neste caso, o nome da API a ser testada, que é utilizado para diferenciar testes do mesmo utilizador, apesar de não ter que ser único, como também o ficheiro de especificação da API.

(B) Representa ficheiros adicionais não obrigatórios que o utilizador pode fornecer. Nomeadamente ficheiros que especificam testes para realizar sobre a API, denominados de TSL, apresentados na secção 3.3; um ficheiro de dicionário que contém informação adicional necessária à realização dos testes especificados nos ficheiros TSL; e ficheiros de extra validações, que contêm a concretização de um contrato de forma a realizar verificações não fornecidas nativamente pela aplicação.

(C) Responsável pela serialização e verificação de todos os ficheiros fornecidos. Tem, como produto final, um ficheiro em formato JSON com todos os testes que vão ser realizados com as suas respetivas verificações. Este ficheiro é útil para repetições recorrentes dos testes, pois deixa de ser necessário validar novamente os ficheiros de *input*.

(D) Responsável pela desserialização do ficheiro resultante do passo (C) e por realizar todos os testes e verificações. Tem como produto final um ficheiro JSON com os resultados.

Tendo em conta o *workflow* apresentado anteriormente, o projeto a realizar contém quatro componentes principais:

1. Servidor, que é responsável por receber e tratar pedidos de acordo com a lógica da aplicação;
2. Cliente, que trata de apresentar uma interface *web* interativa ao utilizador e que comunica com a componente servidora, suportando os passos (A) e (B);
3. Validar os ficheiros recebidos e gerar o ficheiro de teste ((C));
4. Correr todos os testes e verificações, gerando o ficheiro JSON com os resultados ((D)).

Os dois últimos componentes, de forma a melhorar o desempenho do servidor e oferecer uma arquitetura mais escalável, foram desacoplados do servidor. A comunicação entres estes é feita tirando partido do protocolo *Advanced Message Queuing Protocol* (AMQP) [4]. A Figura 3.9 apresenta a arquitetura da solução proposta para o projeto onde as setas representam o fluxo de dados.

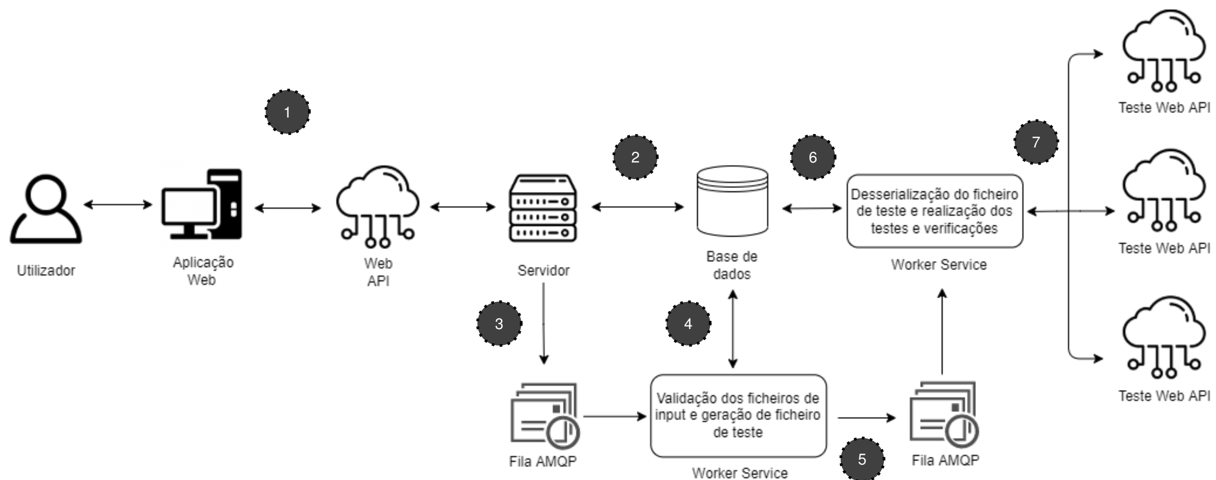


Figura 3.9: Arquitetura do Projeto.

Tendo como base um utilizador que queira configurar um novo teste, este irá aceder à aplicação *web*, completar o processo de configuração, que por sua vez comunica com o servidor através da *web* API (❶). O servidor ao receber este pedido, cria a entrada para esta nova configuração, e coloca todos os ficheiros na base de dados (❷), de seguida, envia uma mensagem para a fila de mensagens onde o serviço responsável por validar os ficheiros se encontra à escuta (❸). O serviço recebe essa mensagem, acede à base de dados para ter acesso aos ficheiros de *input* do utilizador e valida todos os ficheiros que, em caso de estarem válidos, cria o ficheiro de teste e coloca-o também na base de dados (❹). Este finaliza com uma nova mensagem para outra fila sinalizando que esta API está pronta para ser testada (❺). O serviço responsável por realizar os testes, ao receber a mensagem, acede à base de dados para ter acesso ao ficheiro de teste para o deserializar (❻), realizando os pedidos e verificações (❼). Após todos os testes terem sido executados, os resultados são compilados e postos na base de dados onde o servidor tem agora acesso de forma a mostrar os resultados ao utilizador.

4

Implementação

Neste capítulo é especificada a implementação proposta para concretizar todos os objetivos. Inicialmente, são apresentadas as tecnologias utilizadas para o desenvolvimento do projeto na secção 4.1. Seguidamente, na secção 4.2, é apresentada a `API` da componente servidora, como todos os seus *endpoints*. Na secção 4.3 é apresentado como foi feita a autenticação e gestão de utilizadores na aplicação *web*. De seguida são apresentadas todas as páginas principais da aplicação e detalhes sobre a sua implementação e funcionalidades. Começando pela página de entrada do site na secção 4.4, a página de configuração de um novo teste na secção 4.5 e os dois serviços desacoplados responsáveis por validar e correr os testes na secção 4.6, finalizando com a página de monitorização dos testes na secção 4.7.

4.1 Tecnologias de Código Aberto

As metodologias usadas no lado do servidor são focadas em análise de documentos e tratamento de texto com formato `JSON` e `YAML`, tornando a escolha da linguagem de programação um aspeto menos crítico, uma vez que grande parte das linguagens tem as capacidades necessárias para implementar as metodologias. No entanto, foi objetivo (sem prejuízo para a eficiência e desempenho) utilizar tecnologias Código aberto de forma a não existirem custos de licenciamento. Foi também dada a preferência a tecnologias que permitam desenvolver a solução para ser *cross-platform* para que as

aplicações não estejam limitadas quanto às máquinas, e respetivos sistemas operativos, em que podem correr. Para além destas restrições, foi tido em consideração a experiência passada, bem como as preferências pessoais. A ferramenta utilizada para desenvolver a aplicação servidor, bem como toda a lógica aplicacional, foi a *framework .NET Core* [38]. A tecnologia do lado do cliente da aplicação *web*, responsável por implementar a interface com o utilizador, foi desenvolvida usando *JavaScript*, recorrendo à *framework React* [23]. Para a implementação da base de dados foi usado *SQL Server* [59], este não é Código aberto, no entanto, o seu uso facilitou substancialmente a interação com o servidor e os serviços desacoplados, e como é *cross-platform* nas suas novas versões (desde 2017), o seu uso foi justificado.

4.2 API da Componente Servidora

A componente cliente da aplicação comunica com o servidor através de *endpoints*. Todos os *endpoints* do servidor estão presentes em classes controladoras de rotas (*controllers*). Cada *controller* contém os *endpoints* de uma determinada vista da aplicação. A Tabela 4.1 mostra a listagem dos *endpoints* da API da aplicação como também uma descrição sobre cada.

Tabela 4.1: *Endpoints* disponibilizados pela API.

MÉTODO	ROTA	DESCRIÇÃO
	SetupTestController	
POST	`SetupTestController/GetSpecificationDetails`	Valida a especificação da API e retorna dados sobre ela
POST	`SetupTestController/GetSpecificationDetailsURL`	Valida a especificação da API fornecida por URL e retorna dados sobre ela
POST	`SetupTestController/UploadFile`	Finaliza configuração de um novo teste
POST	`SetupTestController/RemoveUnfinishedSetup`	Remove configuração incompleta
	MonitorTestController	
GET	`MonitorTestController/GetUserAPIs`	Listar testes de um utilizador
GET	`MonitorTestController/DownloadReport`	Retornar os resultados do último teste para download
GET	`MonitorTestController/ReturnReport`	Retorna os resultados do último teste e mais informação para visualização
GET	`MonitorTestController/ReturnReportSpecific`	Retorna os resultados de um teste especifico e mais informação para visualização
GET	`MonitorTestController/RunNow`	Executa os testes para esta API imediatamente
GET	`MonitorTestController/GenerateMissingTestsTSL`	Gera e retorna o ficheiro TSL com todos os testes em falta
PUT	`MonitorTestController/ChangeApiTitle`	Altera o nome associado ao teste da API
DELETE	`MonitorTestController/RemoveApi`	Remover uma configuração de teste de um utilizador
	HomeController	
GET	`HomeController/GetUserDetails`	Retornar alguns detalhes sobre o trabalho do utilizador na aplicação

Todos os *endpoints* estão divididos em três controladores, nomeadamente o *SetupTestController*, *MonitorTestController* e *HomeController*. O *SetupTestController* contém todos os *endpoints* relacionados com a configuração de um novo teste, contendo dois *endpoints* para o *upload* da especificação da API, um fornecendo diretamente o ficheiro (*GetSpecificationDetails*) e o outro através de um URL (*GetSpecificationDetailsURL*). O *endpoint*

UploadFile é utilizado no final da configuração, para fornecer a restante informação necessária e o *RemoveUnfinisedSetup* é utilizado caso o utilizador cancele a configuração. O controlador *MonitorTestController* contém todos os *endpoints* relacionados com testes já configurados. Nomeadamente *endpoints* para receber informações sobre todos os testes de um utilizador (*GetUserAPIs*), incluindo *download* dos resultados de um teste (*DownloadReport*), informação detalhada de resultados (*ReturnReport* e *ReturnReportSpecific*), opção para correr os testes imediatamente (*RunNow*), gerar o ficheiro TSL para os testes em falta (*GenerateMissingTestsTSL*), alterar o nome de um teste (*ChangeApiTitle*) ou mesmo remover um teste na sua totalidade (*RemoveApi*). O controlador *HomeController* apresenta apenas um *endpoint*, *GetUserDetails*, que é responsável por devolver informação relevante sobre o utilizador.

4.3 Autenticação de Utilizadores

Para os processos de autenticação e controlo de acessos, foi recorrida à *framework IdentityServer* [28]. Este *middleware* é adicionado na configuração da aplicação e fica responsável por todas as ações relativas à autenticação. Adiciona automaticamente todos os *endpoints* necessários para o registo, autenticação e gestão de conta para um utilizador. Para guardar localmente os utilizadores registados, o *IdentityServer* necessita de uma base de dados local, tipicamente *SQL Server*. Na Figura 4.1 é possível observar o modelo de dados oferecido pelo *IdentityServer*.

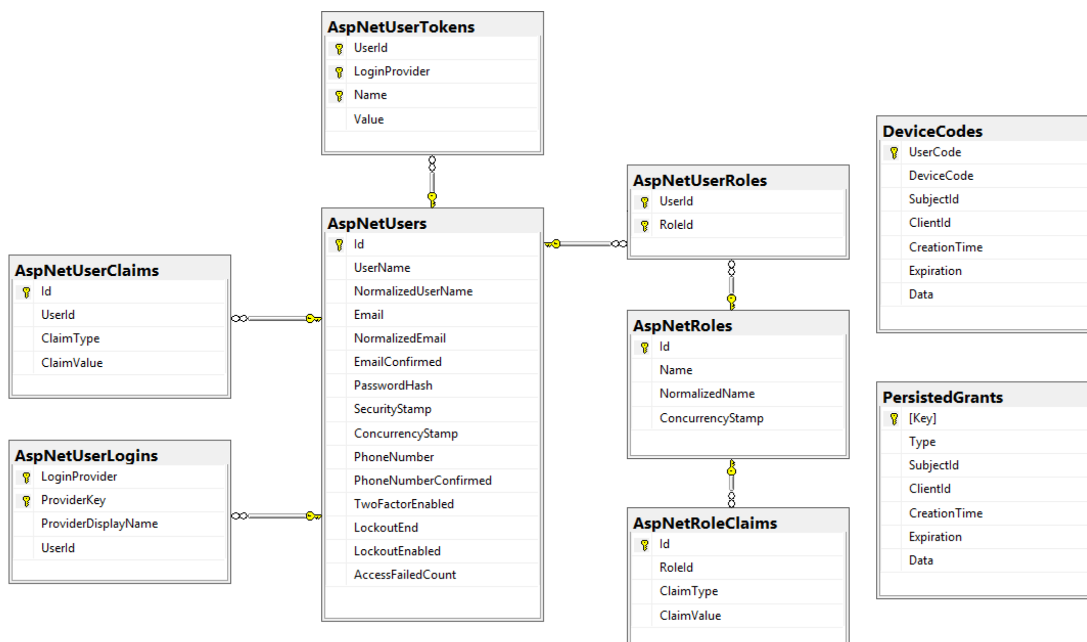


Figura 4.1: Modelo da base de dados para a autenticação, *framework IdentityServer*.

A tabela *AspNetUsers* é uma das mais importantes, pois é onde são armazenados os dados do utilizador como o seu *email*, palavra-passe (*hashed*) entre outros. A geração do *hash* inclui *salt* por forma a duas palavras-passe idênticas não gerarem o mesmo *hash*. O algoritmo, também, realiza o *hash* várias vezes, de forma iterativa, de maneira a que um atacante que esteja a tentar fazer um ataque de Força Bruta [8] tenha também de realizar o *hash* o mesmo número de vezes que a aplicação realiza, levando a que o tempo necessário para cada tentativa de palavra-passe aumente linearmente. As outras tabelas são utilizadas para diversos propósitos, como gestão de *tokens* de autenticação, autenticação externa, *two-factor authentication*, suporte a diferentes papéis (*roles*) para diferentes tipos de utilizadores entre outros.

4.4 Página de Entrada

A aplicação foi desenvolvida como *Single Page Application* (SPA). Uma SPA contém uma única página *web* e usa *JavaScript*, *HTML5* e *CSS* para a interação com o utilizador. Na SPA, após o carregamento inicial, não existe mais nenhum carregamento total durante o seu uso [27]. A interação da página com o utilizador vai iniciar processos de carregamento parciais, executados em segundo plano, usando *JavaScript*. A SPA faz *render* das páginas diretamente no *browser*, isto é facilitado graças às *frameworks JavaScript*, como *AngularJS* [13], *Ember.js* [17], *Meteor.js* [21], *Knockout.js* [20] e *ReactJS*. Aplicações *web* tais como *Gmail*, *Google Maps*, *Facebook* ou *GitHub* são SPA [58]. Tendo isto em mente a aplicação foi desenvolvida como uma SPA de modo a cumprir os requisitos referidos na secção 3.4.1. A Figura 4.2 apresenta a página inicial completa da aplicação para um utilizador não autenticado.

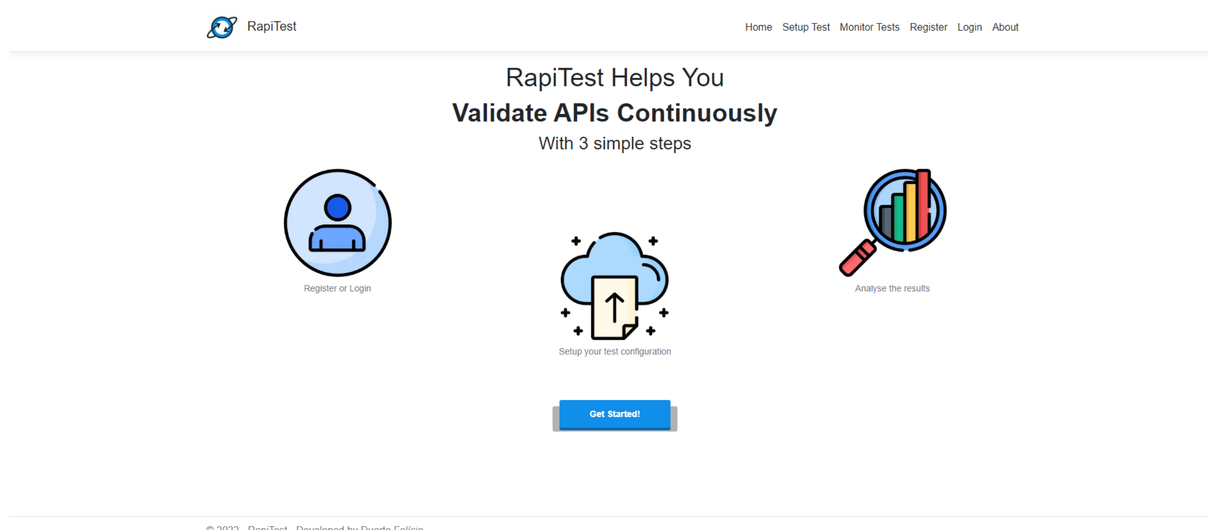


Figura 4.2: Página inicial da aplicação para um utilizador não autenticado.

O topo da página apresenta a barra de navegação, no centro, uma explicação sobre o uso da aplicação em três passos (*login*, configurar um teste e analisar os resultados) e no fim da página o rodapé com meta informação.

Página Inicial - Autenticado Após a autenticação, na página inicial, o utilizador tem informação sobre os testes recentemente realizados (Figura 4.3.Ⓐ), incluindo duas *badges* com o número de erros e avisos, sendo cada entrada também um *link* direto para a visualização dos resultados do teste. O número de testes configurados e a data do último acesso podem ser observados na Figura 4.3.Ⓑ e Figura 4.3.Ⓒ respetivamente. Na página encontra-se também uma tabela (Figura 4.3.Ⓓ), que apresenta todos os testes configurados do utilizador, como também quando será o próximo teste para cada um deles. Sempre que um utilizador faz *login* ou *register* é adicionada uma nova entrada à tabela *LoginRecord*, esta tabela serve como histórico de cada utilizador de forma a mostrar o último *login* de cada.

#	API Name	Next Test
0	CML	2022-07-04 13:48:02
1	Amazon	2022-06-28 13:50:59
2	COIMR2	2022-06-28 01:51:45

Figura 4.3: Página inicial para um utilizador autenticado. As áreas destacadas representam: (A) os testes recentemente realizados, (B) o número de testes configurados, (C) a data de acesso e (D) uma tabela que apresenta a data dos próximos testes.

4.5 Configurar um Novo Teste

A aplicação permite efetuar a configuração de um novo teste através de cinco passos:

1. Fornecimento de um nome para o teste;
2. *Upload* local ou por URL da OAS;
3. Criação ou *upload* de ficheiro TSL;
4. Configuração do temporizador e outros dados;
5. Configurar outro teste ou observar os resultados.

4.5.1 Fornecimento de um Nome para o Teste

Como primeiro passo, para fornecer o nome do teste, usualmente o da API, foi utilizado o elemento *html Form* de forma a obter o *input* do utilizador (Figura 4.4). O nome fornecido pelo utilizador é filtrado de forma a não aceitar nomes com caracteres especiais (e.g., !@#$%^ etc...) como foi mencionado na secção 3.4. A Figura 4.4 apresenta a página para fornecer o nome do teste. O topo da página apresenta uma indicação do passo atual e uma descrição de cada um.

Figura 4.4: Página para fornecer o nome do teste.

4.5.2 *Upload* Local ou por URL da OAS

A aplicação permite efetuar o carregamento de ficheiros diretamente da máquina do utilizador, ou através de um endereço URL. A Figura 4.5 apresenta a página para este passo, tendo na zona Ⓐ uma *dropzone* onde é possível de arrastar e largar o ficheiro OAS, ou então clicando sobre esta é também possível fazer *browse* do *file system* para escolher diretamente o ficheiro. Na zona Ⓑ é possível de fornecer um URL relativo à OAS como alternativa. A zona Ⓒ apresenta um botão para retroceder de passo.

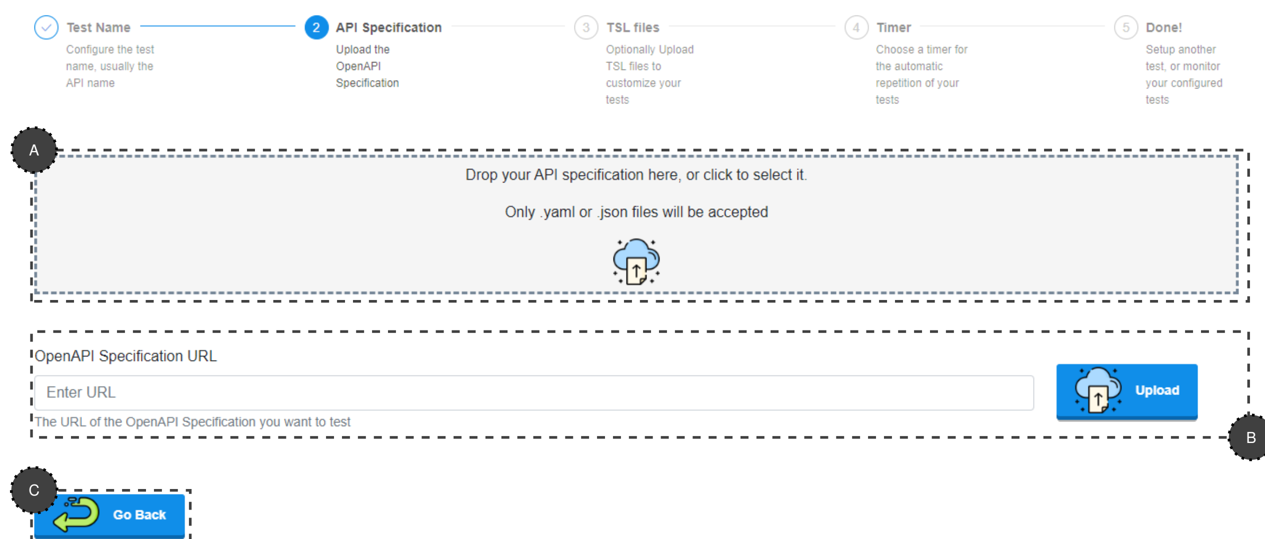


Figura 4.5: Página para *upload* da especificação. As áreas destacadas representam os principais componentes de interação com o utilizador: (A) *Dropzone* para *upload* local, (B) *Upload* por URL e (C) Retroceder para o passo anterior.

Lado Cliente A parte cliente para carregamento local de ficheiros (Figura 4.5.A) foi implementada através da biblioteca *React Dropzone* que facilita a criação e customização de uma zona de *drag'n'drop* [51], de modo a só serem aceites ficheiros no formato JSON ou YAML. Após o carregamento da OAS, localmente ou por URL, um pedido é enviado ao servidor onde este a valida. No caso de ser uma OAS válida o utilizador avança para o terceiro passo, caso seja inválida, uma mensagem de erro aparece e o utilizador deve fornecer outra OAS.

Lado Servidor Para o carregamento a partir de um URL o servidor recebe no pedido o URL do ficheiro que tem de descarregar. São feitas as verificações de forma a afirmar que o ficheiro é um ficheiro aceitável pela aplicação e de seguida é aberto um *stream* desse endereço e o ficheiro é descarregado. Após ter acesso ao ficheiro este deve ser validado e transformado num modelo de dados específico. Para implementar este passo foi tirado partido da biblioteca *OpenApi* [22], biblioteca oficial da *Microsoft* realizada em parceria com a *OpenApi Initiative* com o intuito de *standardizar* a forma de descrição de API oferecendo ferramentas automáticas para múltiplas linguagens de programação. Se a OAS for válida, dados como os *servers*, *paths* e *schemas* da API são enviados na resposta, úteis para os passos seguintes do utilizador, sendo que é também inserida uma nova entrada na base de dados com o nome e a OAS, de forma a não ser necessário enviá-la novamente. Caso seja inválida, uma resposta com código de erro é enviada e nenhuma entrada é criada.

4.5.3 Criação ou *Upload* de Ficheiro TSL

Como terceiro passo o utilizador é apresentado com três opções, criar um ficheiro TSL através da *Graphical User Interface* (GUI) (Figura 4.6.Ⓐ), fazer *upload* do seu próprio ficheiro TSL (Figura 4.6.Ⓑ), ou então simplesmente ignorar este passo (Figura 4.6.Ⓒ), o que implica que os únicos testes vão ser os gerados automaticamente através da OAS.

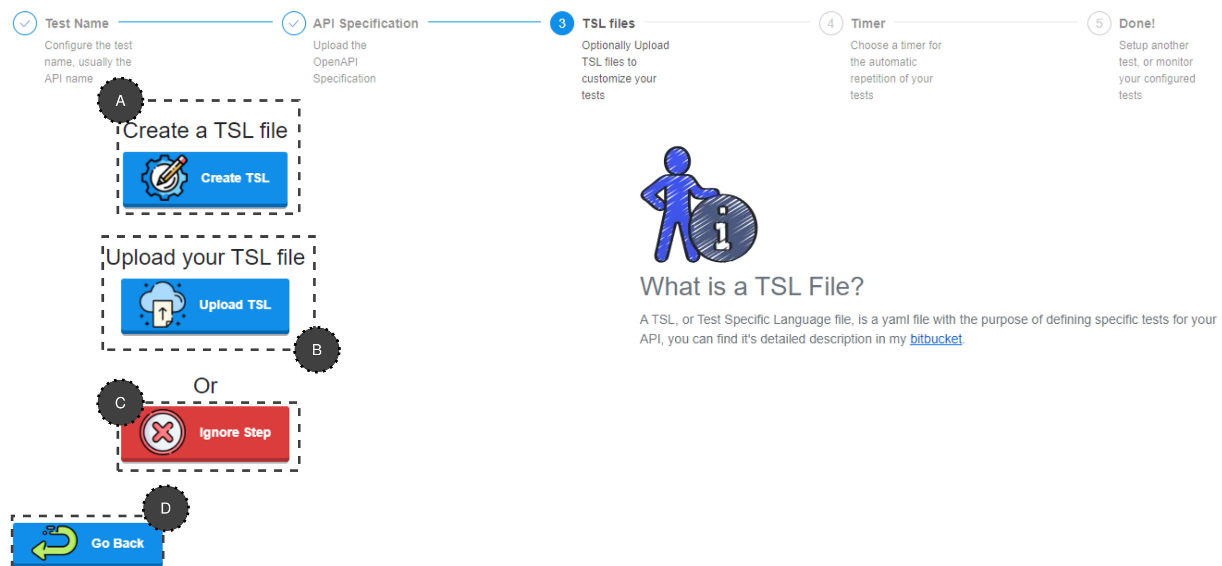
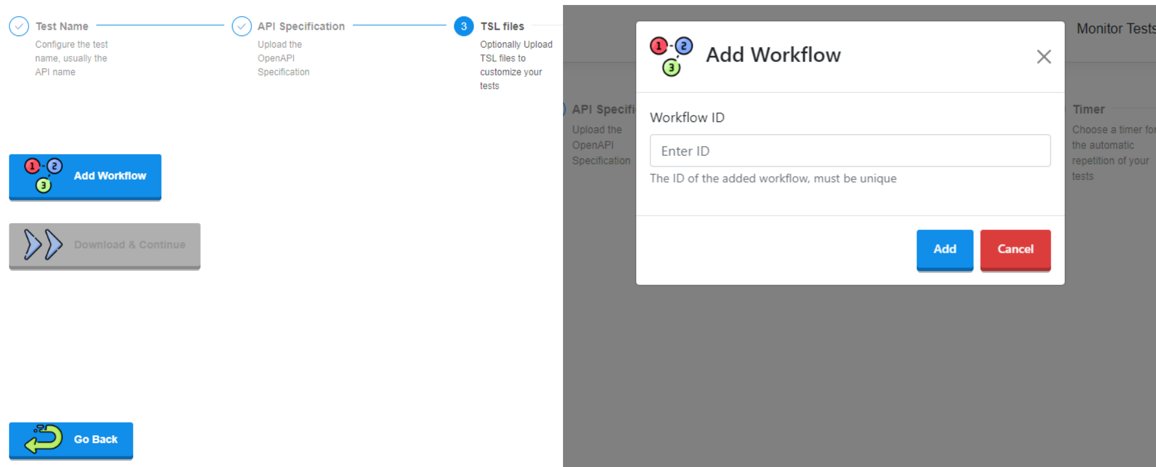


Figura 4.6: Página para escolha do utilizador. As áreas destacadas representam os principais componentes de interação com o utilizador: (A) Criação do TSL, (B) *Upload* do TSL, (C) Ignorar passo e (D) Retroceder.

Nesta situação, quando o utilizador retrocede para o passo anterior (Figura 4.6.Ⓓ), este retrocede para o passo de *upload* da OAS. No final desse passo, um pedido foi realizado ao servidor e como a OAS era válida foi criada uma entrada na base de dados. Ao retroceder, a entrada encontrar-se-á inválida e deverá ser removida. É enviado um pedido ao servidor sinalizando para que esta seja retirada. É também relevante mencionar que se o utilizador sair do processo de configuração antes de estar completo, este pedido é também enviado de forma a garantir a consistência da base de dados.

Criar um ficheiro TSL através da GUI Ao escolher esta opção o utilizador tem agora a possibilidade de criar um ficheiro TSL. No início, o utilizador apenas tem a hipótese de adicionar um *workflow*, pois o botão para finalizar ainda se encontra desativado. A Figura 4.7 apresenta a página inicial da criação, com o botão para adicionar um *workflow*, outro *disabled* para finalizar e outro para retroceder. A Figura 4.8 apresenta a janela que aparece no topo da página quando o utilizador clica no botão *Add Workflow*. Esta janela pede apenas o *id* do *workflow*.



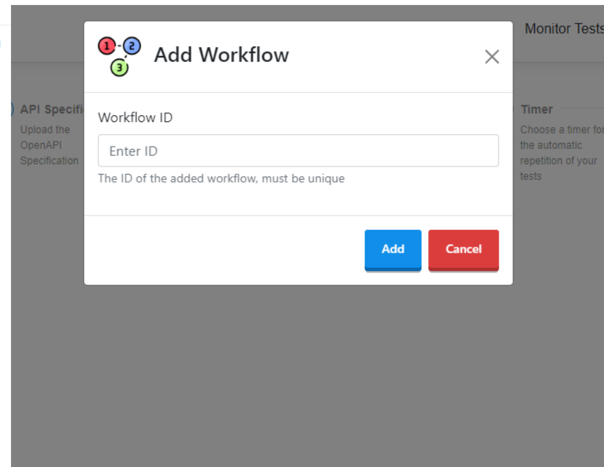


Figura 4.7: Página inicial da criação.

Figura 4.8: Janela para criar um *workflow*.

Tendo o *workflow* criado o utilizador tem agora acesso a novas funcionalidades apresentadas na Figura 4.9.

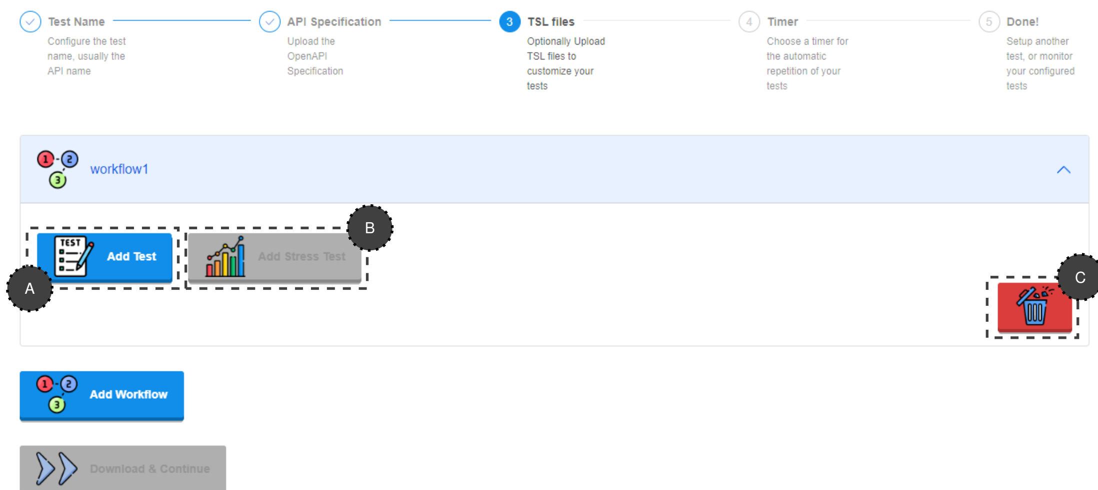


Figura 4.9: Página com um *workflow* criado. As áreas destacadas representam os principais componentes de interação com o utilizador: (A) Criação de um teste, (B) Criação de um teste de carga e (C) Apagar *workflow*.

Nomeadamente, este pode criar um teste para o *workflow* (Figura 4.9.Ⓐ), ou mesmo apagar o *workflow* (Figura 4.9.Ⓒ). O botão para adicionar um teste de carga (Figura 4.9.Ⓑ) encontra-se *disabled* pois o *workflow* tem que ter no mínimo um teste para poderem ser feitos testes de carga.

Ao clicar no botão para adicionar um teste (Figura 4.9.Ⓐ) o seguinte formulário dá *popup*, apresentado na Figura 4.10.

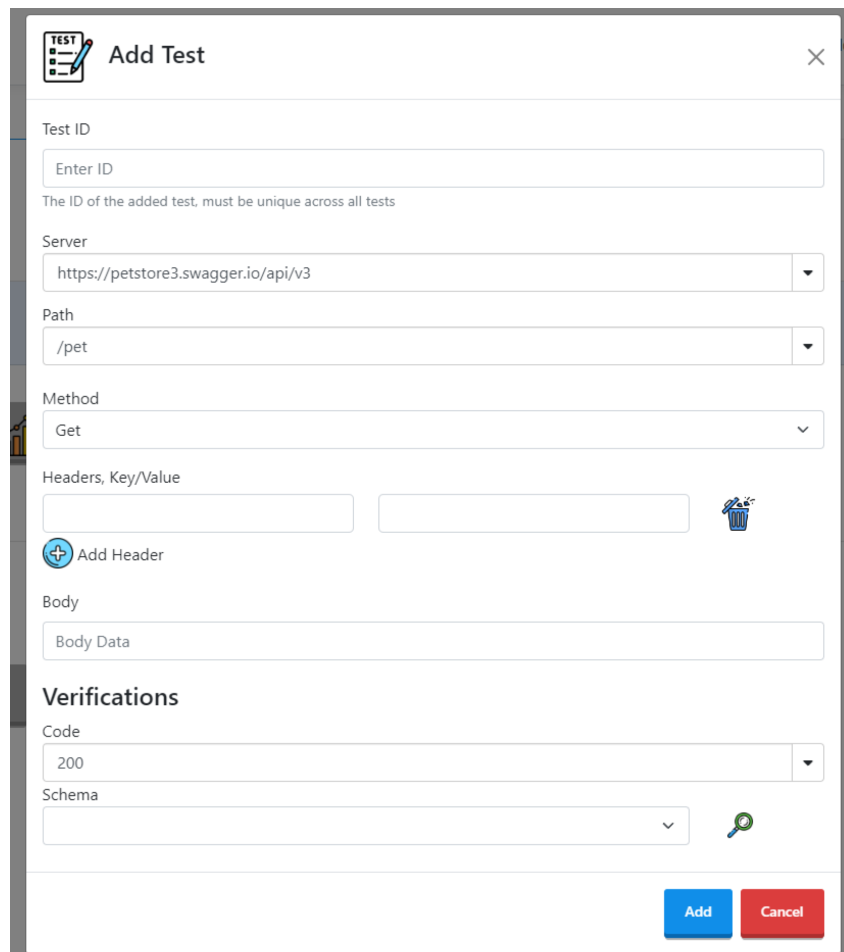


Figura 4.10: Formulário para criar um teste.

O formulário apresenta todas as funcionalidades principais necessárias para a criação de um teste. É nesta situação que os valores retornados após a analise da OAS são úteis, sendo possível de oferecer uma criação mais especifica do TSL para a API, ao invés de uma abordagem totalmente livre, deixando o utilizador escrever qualquer *input*. Desta forma campos como o *Server*, *Path* e *Schema* contêm diretamente os valores da API. É também relevante mencionar que é possível de observar os esquemas em detalhe através da própria aplicação, mantendo o utilizador na aplicação sem o obrigar a mudar de ecrã para analisar os esquemas.

É então possível fornecer um *id* para o teste, o *server*, *path* com *query string*, *method*, *headers*, *body*, verificação de *Code* e outra de *Schema*. Naturalmente, os únicos campos obrigatórios são o TestID, o *Server*, *Path*, *Method* e verificação de *Code*. Esta criação é no entanto limitada, não fornecendo funcionalidades como o *Retain*, validações externas ou mesmo todas as validações nativas. Estas funcionalidades foram excluídas para trabalho futuro, sendo que a sua ausência simplifica também a GUI.

A Figura 4.11 apresenta a página após a criação de um teste.

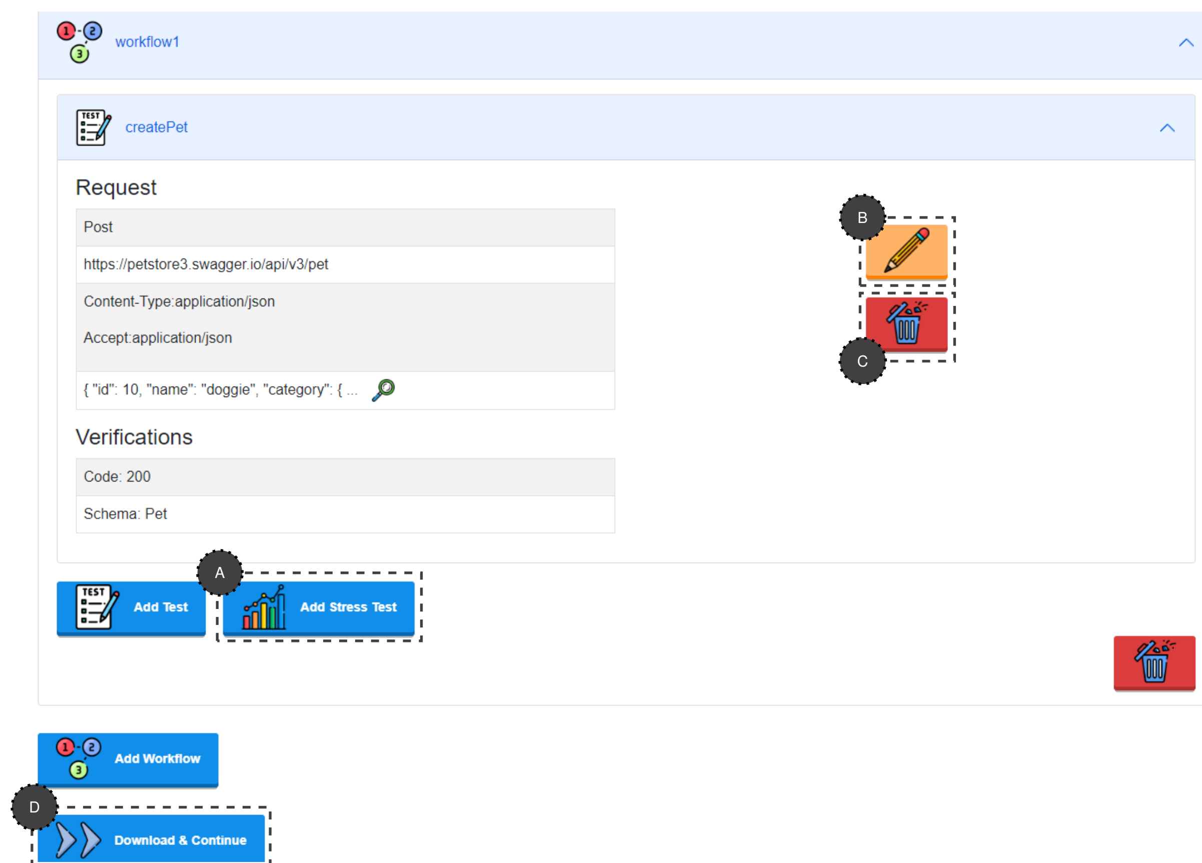


Figura 4.11: Página com um teste criado. As áreas destacadas representam os principais componentes de interação com o utilizador: (A) Criação de um teste de carga, (B) Editar o teste, (C) Apagar o teste e (D) Finalizar a criação.

Esta página apresenta agora novas funcionalidades para o utilizador, nomeadamente, a possibilidade de adicionar um teste de carga (Figura 4.11.Ⓐ), este botão encontra-se agora disponível pois o *workflow* já contém um teste, sendo este o número mínimo de testes para realizar um teste de carga. O utilizador pode também editar o teste que criou (Figura 4.11.Ⓑ), ao clicar neste botão a janela de criação de teste aparece com algumas alterações, como o facto das zonas para preencher os dados se encontrarem diretamente com os valores do teste que está a ser editado, de forma a facilitar o utilizador. O teste pode também naturalmente ser apagado (Figura 4.11.Ⓒ).

Tendo agora criado um *workflow* com um teste, o utilizador pode finalizar o processo de criação (Figura 4.11.Ⓓ), ao clicar no botão o processo é finalizado e o utilizador recebe automaticamente o ficheiro `TSL` com o respetivo ficheiro de dicionário caso os testes contenham corpo. O utilizador recebe estes ficheiros, sendo que não é obrigado a realizar qualquer operação sobre eles, estes servem apenas caso o utilizador deseje editar/adicionar um teste, este pode realizá-lo diretamente sobre o ficheiro sem ser

necessário criar tudo novamente. Este serve também como uma forma de familiarizar o utilizador sobre os ficheiros TSL, de forma a entender a ponte entre o que foi criado e o resultado em ficheiro, sendo que certas funcionalidades dos ficheiros TSL não são suportadas através da interface gráfica. Clicando sobre o botão para adicionar um teste de carga a seguinte página aparece na topo, apresentada na Figura 4.12.

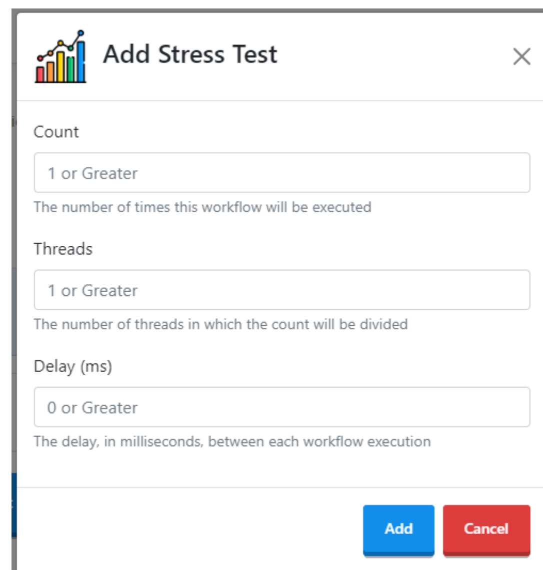


Figura 4.12: Formulário para criar um teste de carga.

Após a criação do teste de carga o utilizador tem acesso a duas novas funcionalidades, apresentadas na Figura 4.13.

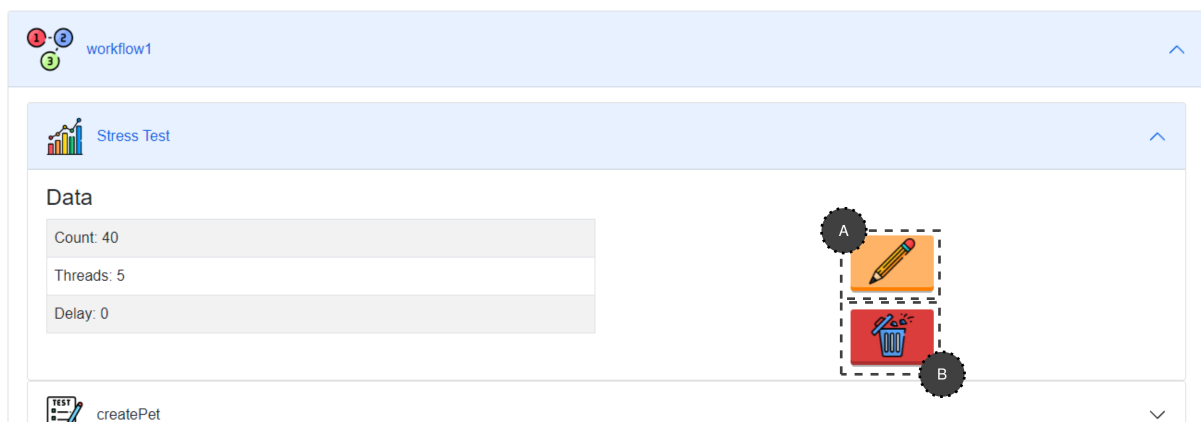


Figura 4.13: Página com um teste de carga criado. As áreas destacadas representam: (A) Editar o teste de carga e (B) Apagar o teste de carga.

Nomeadamente, a possibilidade de editar o teste de carga (Figura 4.13.Ⓐ), onde de forma semelhante à edição de teste, os campos para preencher contêm diretamente os

dados de *stress test* a ser editado, e tem naturalmente a possibilidade de apagar o teste de carga (Figura 4.13.Ⓑ). Após criar o teste de carga o botão para adicionar um teste de carga desativa-se, pois apenas pode haver um teste de carga por *workflow*.

Upload do próprio ficheiro TSL Como segunda opção o utilizador pode fornecer os seus próprios ficheiros TSL. Estes ficheiros podem também incluir a necessidade de um ficheiro de dicionário como também de ficheiros DLL de extra validações. De forma a realizar o *upload* de todos estes ficheiros foi reutilizado o componente *React Dropzone* configurando as extensões para cada tipo de ficheiro, YAML para os ficheiros TSL, TXT para o ficheiro dicionário e DLL para as validações externas, possibilitando o *upload* fazendo *browse* do *file system* ou arranstando o ficheiro. A Figura 4.14 apresenta a página para *upload* dos ficheiros.

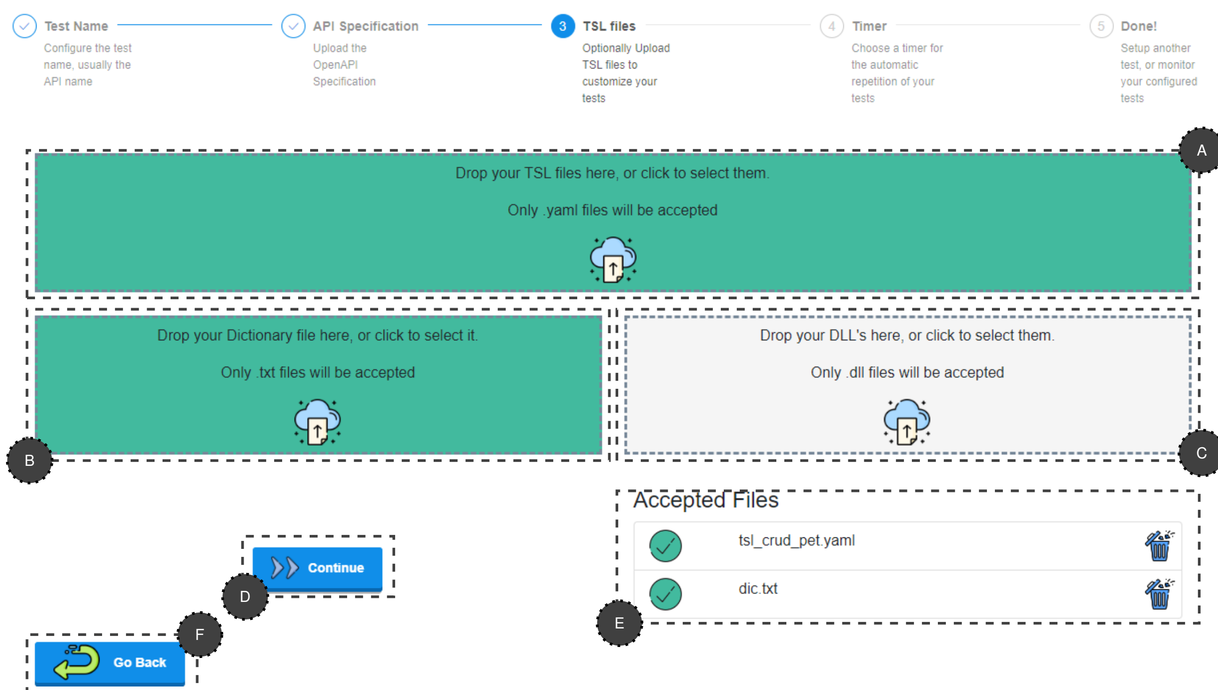


Figura 4.14: Página para o passo opcional da configuração de um teste. As áreas destacadas representam: (A) zona para o *upload* dos ficheiros TSL, (B) zona para o ficheiro dicionário, (C) zona para os ficheiros de validações externas, (D) botão para continuar, (E) registo dos ficheiros que foram aceites e (F) botão para retroceder de passo.

Esta página contém seis blocos principais:

1. Zona para o *upload* dos ficheiros TSL (Figura 4.14.Ⓐ);
2. Zona para o ficheiro dicionário (Figura 4.14.Ⓑ);

3. Zona para os ficheiros de validações externas (Figura 4.14.Ⓒ);

4. Zona com todos os ficheiros que foram aceites de forma a que o utilizador tenha noção de que ficheiros está a enviar (Figura 4.14.Ⓔ), sendo também possível remover algum ficheiro que o utilizar tenha feito *upload* por engano;

5. Zona com botão para continuar, sendo que o utilizador deve ter feito no mínimo *upload* de um ficheiro `TSL` para poder continuar (Figura 4.14.Ⓓ);

6. Zona com botão para retroceder, sendo que todos os ficheiros que tenham sido enviados são descartados (Figura 4.14.Ⓕ).

Algumas zonas de *dropzone* estão apresentadas a verde pois o utilizador já enviou um ficheiro desse tipo, como nenhum ficheiro de extra validações `DLL` foi enviado, a zona ainda se encontra a cinzento sendo que a zona para ficheiros `TSL` é a única obrigatória.

4.5.4 Configuração do Temporizador e Outros Dados

Como penúltimo passo, a seguinte página é apresentada ao utilizador, Figura 4.15.

Figura 4.15: Página para o passo da configuração da frequência.

Nesta página o utilizador deve escolher se quer correr imediatamente os testes após a conclusão da configuração, correr os testes gerados automaticamente, e deve também escolher com que frequência, ou nenhuma, quer que a aplicação corra novamente os testes, de forma a validar o continuo correto funcionamento da `API` como também monitorizar o *uptime* da `API`. Neste caso, a Figura 4.15 apresenta a configuração de um

utilizador que escolheu correr imediatamente os testes, correr também os testes gerados automaticamente e que estes corram a cada 1 hora. A aplicação apresenta quatro diferentes valores de frequência possíveis, sendo estes valores arbitrários e sujeitos a mudança em futuras iterações. É também relevante mencionar que se o utilizador ignorou o terceiro passo para *upload* ou criação de ficheiro `TSL`, o utilizador é obrigado a correr os testes gerados automaticamente, sendo estes os únicos executados.

4.5.5 Configurar Outro Teste ou Observar os Resultados

Após a conclusão destes quatro passos, o pedido final é enviado ao servidor, este pedido contém toda a informação fornecida depois do segundo passo, nomeadamente, os ficheiro `TSL`, todos os ficheiros opcionais, se deseja correr os testes gerados, se os deseja correr imediatamente e com que frequência. Após a receção da resposta, o cliente como último passo demonstra uma janela de sucesso ou erro dependendo do resultado, dando a hipótese de configurar outro teste, ou de observar os resultados.

Lado Servidor Uma das decisões necessárias durante a implementação foi onde guardar a informação proveniente do pedido do cliente. Umas das opções seria guardar no *file system*, no entanto esta opção prejudica a escalabilidade da solução, pois todos os sub-sistemas teriam que ter acesso ao mesmo *file system*. Uma solução a este problema seria através do uso de um *file system* distribuído disponibilizado em serviços como a *Amazon Web Services* [12] ou o *Gluster* [18] para uma solução que não dependa de serviços da *cloud*. Posto isto, a opção implementada foi a de utilizar na totalidade a base de dados *SQL Server*, não tendo qualquer dependência do *file system*, obtendo assim a escalabilidade, caso necessário, desejada. A Figura 4.16 apresenta o modelo da base de dados que suporta este pedido, nomeadamente as tabelas `API`, *ExternalDll* e *Report*.

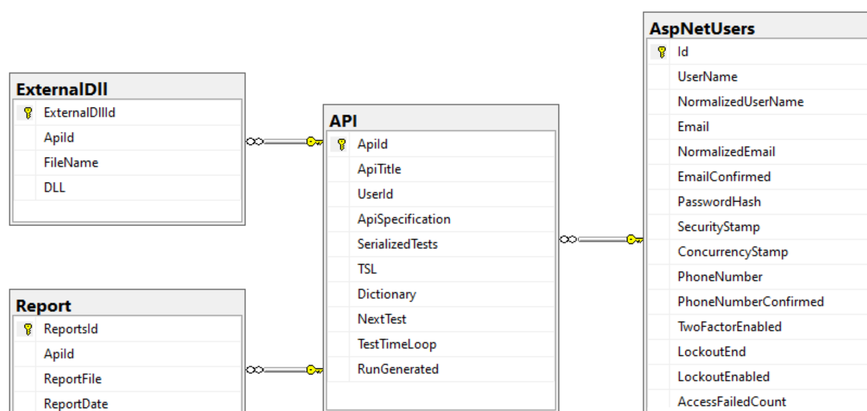


Figura 4.16: Modelo para a configuração de testes - `API` e *ExternalDll*.

A tabela API contém como chave estrangeira o *id* do utilizador e contém toda a informação que o utilizador forneceu à exceção dos ficheiros de validação externa, que são colocados na tabela *ExternalDll*, pois estes podem ser vários, sem forma de os agrupar, em contraste com os ficheiros de TSL que podem ser todos agrupados num único ficheiro. É também relevante mencionar que todos os ficheiros foram guardados na base de dados como *varbinary* pois estes ficheiros são todos de pequena dimensão. Outra opção seria criar uma estrutura na base de dados para o ficheiro, no entanto, devido à complexidade dos diferentes campos e valores, a opção de *varbinary* tornou-se a mais adequada durante o desenvolvimento. Após a inserção das novas entradas nas tabelas, uma mensagem de aviso é enviada a outro serviço, explicando em maior detalhe na secção seguinte. A tabela *report* é utilizada para guardar os resultados de cada teste. Esta encontra-se associada a uma API devido a poderem existir múltiplos relatórios, sendo que guarda o relatório em formato *varbinary* pelas mesmas razões apresentadas acima.

4.6 Validação, Controlo e Execução dos Testes

Para não degradar o desempenho da aplicação e não prejudicar a interface com o utilizador foram criados dois serviços que correm em paralelo com a aplicação. O primeiro serviço é responsável por validar todos os ficheiros OAS fornecidos pelo utilizador, presentes na base de dados e, no caso de estarem bem configurados, enviar outra mensagem ao segundo serviço, este responsável por realizar todos os pedidos e validações. A comunicação entre as diferentes partes é feita através de duas filas de mensagens que seguem o protocolo AMQP implementado através do *message broker RabbitMQ* [50], que introduz o conceito de produtor e consumidor. O primeiro serviço, responsável pela validação e controlo, irá estar à escuta, como consumidor na fila, por instruções provenientes do servidor, o produtor. Após o processo desta mensagem, este age agora como produtor, noutra fila, onde o segundo serviço, consumidor, processa a mensagem.

4.6.1 Validação e Controlo - *Worker Service*

O primeiro serviço, ao receber a mensagem proveniente da aplicação, inicia o processo de validação. Cada mensagem contém o *id* da API que foi configurada e se devem ser realizados os testes imediatamente após a validação. O *id* é enviado pois o serviço tem que poder aceder e alterar informação sobre essa entrada na base de dados. O processo da validação é composto por oito passos distintos:

***Parsing* da especificação da API** O ficheiro da especificação da API presente na base de dados encontra-se no formato *varbinary*, no entanto, de forma a ser possível analisar este ficheiro de forma programática este deve ser validado e *parsed* para um modelo de dados especifico. Para implementar este passo foi tirado partido novamente da biblioteca *OpenApi*, sendo que neste caso não é necessário validar a especificação pois já foi validada no segundo passo da configuração.

***Parsing* do ficheiro TSL** De forma a realizar o *parsing* do ficheiro TSL, definido em YAML, foi tirado partido da bibilioteca *YamlDotNet* [24] que simplifica todo o processo de deserialização do ficheiro. O ficheiro é deserializado para o seguinte modelo de dados, apresentado na Figura 4.17.

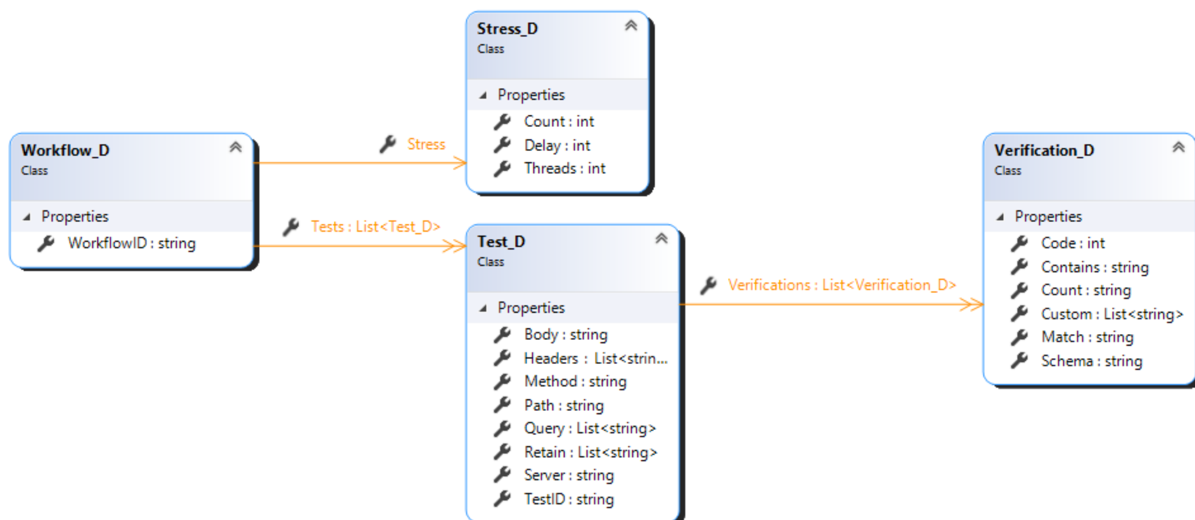


Figura 4.17: Modelo de Dados para a deserialização do ficheiro TSL.

O modelo segue naturalmente a arquitetura apresentada na secção 3.2, *Workflow* que contém um teste de carga e múltiplos testes, com múltiplas verificações. Este passo consiste apenas numa deserialização direta do ficheiro, onde os campos ainda são maioritariamente texto, uma segunda deserialização é realizada para objetos mais detalhados num passo seguinte.

***Parsing* do ficheiro dicionário** O ficheiro dicionário, como apresentado anteriormente, é composto por um conjunto de blocos de informação separados por uma linha vazia. Estes blocos estão identificados por um *id* seguidos do seu conteúdo, ambos em texto. Desta forma, foi tirado partido de uma estrutura de dados par chave valor, onde a chave é o *id* e o valor o conteúdo associado.

Validação dos ficheiro de verificações externas Neste ponto de desenvolvimento, a aplicação apenas suporta verificações externas através do *upload* de ficheiros de biblioteca, nomeadamente ficheiros DLL. É reconhecido que este fator consiste numa falha de segurança para a aplicação, no entanto foi implementado como prova de conceito de forma a comprovar a possível extensibilidade da aplicação para verificações que os utilizadores desejam realizar, sem estarem limitados apenas às oferecidas nativamente. É contudo verificado que a biblioteca obedece ao contrato estabelecido na secção 3.3. Estes dados são guardados numa estrutura chave valor, sendo a chave o nome do ficheiro utilizado como próprio identificador no ficheiro TSL e o valor um objeto do tipo *dynamic*, objeto este cujas operações são resolvidas apenas em *runtime*.

***Parsing* para objetos de lógica da aplicação** Como foi mencionado anteriormente, uma segunda deserialização é necessária de forma a validar e preparar por completo os testes a serem realizados. Diversos campos sofreram alterações sobre o seu tipo, a Figura 4.18 apresenta o modelo de dados final.

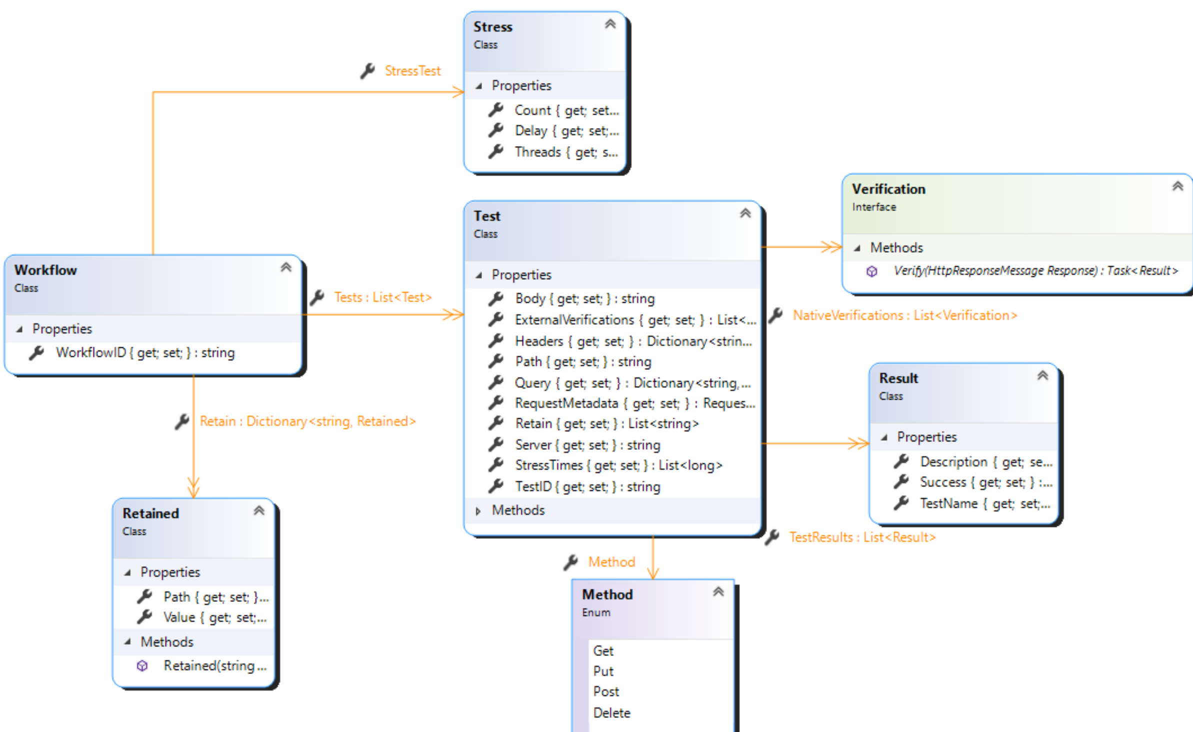


Figura 4.18: Modelo de Dados para a deserialização do ficheiro TSL final.

Este modelo apresenta toda a informação necessária para a realização dos testes. Começando pelo *Workflow* este apresenta o seu *id*, um teste de carga, uma lista de testes

e um dicionário com os valores *retained* entre testes deste *workflow*, relevante para suportar a retenção de valores entre diferentes testes. O teste apresenta toda a informação necessária para a realização de um pedido HTTP, nomeadamente, o *Server*, *Path*, *Method*, *Query*, *Headers* e *Body*. Apresenta também duas lista das verificações nativas e externas a realizar neste teste e outra para guardar os tempos de resposta dos testes de carga. Os objetos das verificações são instanciados, tendo em conta os parâmetros de entrada de cada verificação. Todos os testes têm como resultado final um *Result*, e são guardados numa lista no próprio teste. A Tabela 4.2 apresenta todas as alterações de modelo realizadas.

Tabela 4.2: Trocas dos modelos antigos para os modelos finais da lógica da aplicação.

	Tipo Antigo	**Tipo Novo**
Query	List<string>	Dictionary<string,string>
Headers	List<string>	Dictionary<string,string>
Retain	List<string>	List<string,Retain>
Method	string	Enum
Body	string	Trocar referência pelo texto no dicionário
External	List<string>	List<dynamic>
Match	string	string path, string value
Count	string	string target, int count
Schema	string	Trocar referência pelo esquema da especificação/dicionário

Verificação de erros Nestes passos anteriores, no caso de qualquer erro, como uma falha no *parsing* da especificação, TSL e dicionário devido a má formatação, falha devido à repetição de *id* no dicionário, contrato não válido nas validações externas, entre outros, uma compilação dos erros é guardada e estes erros são guardados na base de dados para o utilizador poder observá-los na aplicação cliente. Se for detetado pelo menos um erro, os dois seguintes passos não são realizados e não é enviada nenhuma mensagem ao serviço de execução dos testes.

Verificação de *endpoints* não testados e geração automática de testes Após a validação dos testes a realizar, a especificação da API é utilizada para observar se existe alguma combinação de *server*, *endpoint*, *input*, *output*, *code* que não está a ser testada. Se for esse o caso, é guardado um aviso de que essa combinação não está a ser validada. Esta mesma análise à OAS é utilizada para a geração automática de testes, para cada uma das combinações mencionadas anteriormente é criado um teste. Valores como o *Server*, os tipos do conteúdo do pedido e da resposta e os códigos de resposta possíveis

são diretamente dados pela `OAS` e não necessitam de processamento extra. No entanto, campos como o *Path* podem conter variáveis, que necessitam de ser especificadas, e pedidos com corpo necessitam naturalmente de o conter. Múltiplas das ferramentas apresentadas no Capítulo 2 apresentam diversas alternativas e soluções a estas problemas, como a análise de respostas anteriores, análise dos *endpoints* para tentar encontrar dependências, entre outros. Posto isto, a geração automática de testes não é um ponto fulcral da solução e esta foi apenas concretizada como prova de conceito. Os *paths* com variáveis são substituídos por um inteiro aleatório e os corpos dos pedidos são criados tendo como base o esquema, no entanto, o objeto gerado contém como valores apenas os valores *default* de cada tipo. É tido em consideração que esta não é a solução ideal, contudo a sua melhoria é um fator de trabalho futuro. Em relação às verificações de cada teste, a verificação do código é realizada, e caso o teste retorne um objeto com certo esquema, esta verificação é também realizada.

Serialização dos testes Sendo um dos objetivos principais a realização recorrente dos mesmos testes de forma a validar o constante correto funcionamento da `API` em avaliação, foi importante encontrar uma solução para uma forma eficiente de carregar rapidamente os mesmos testes. A solução proposta foi a de serializar toda a informação do teste, em formato `JSON`, que é de seguida guardado na base de dados de forma a dar acesso ao serviço que executa os testes. A Listagem 4.1 apresenta o ficheiro resultante dos oito passos de processamento.

Esta apresenta um exemplo de um *workflow* (linha 2-22), com um teste de carga (linha 4) e com apenas um teste (linha 6-22), sendo que iriam ser realizadas diversas validações sobre esse teste, onde apenas é apresentada a validação *Code* (linhas 17-18). A serialização das verificações nativas apresenta uma peculiaridade na linha 17, por utilizar como opção o *auto TypeNameHandling* na serialização, ao realizar a deserialização os objetos respetivos são automaticamente instanciados, recebendo o seus valores no construtor. No final, estão também guardados todos os avisos referidos anteriormente (linhas 23-25) seguido de todos os testes gerados automaticamente (linhas 26-28).

De forma a suportar os recorrentes testes a cada período de tempo, foi tirado partido da *framework System.Timers.Timer*. Esta permite a criação de um objeto que após certo período de tempo um evento é disparado. Quando o serviço recebe uma nova mensagem vinda da aplicação é então verificado se essa nova configuração pretende realizar testes recorrente-mente e se sim, é criado um *Timer* com esse intervalo. Quando o evento é disparado uma mensagem é enviada ao serviço responsável pela execução de testes sinalizando que é necessário realizar testes e outro *Timer* é criado em função do intervalo definido para o próximo teste.

Listagem 4.1: Exemplo de uma serialização do ficheiro de testes.

```
1  { "ApiId": 6097,
2     "Workflows": [{
3           "WorkflowID": "crud_pet",
4           "StressTest": {"Count": 40, "Threads": 5,"Delay": 0 },
5           "Retain": {"petId": {"Path": "$.id","Value": null }},
6           "Tests": [ {
7                 "TestID": "createPet",
8                 "Server": "https://petstore3.swagger.io/api/v3",
9                 "Path": "/pet",
10                "Method": 2,
11                "Headers": { "Content-Type": "application/json",
12                    "Accept": "application/json"},
13                "Body": "{ \"id\": 10, \"name\": \"doggie\"}",
14                "Retain": [ "petId#$.id" ],
15                "Query": {},
16                "NativeVerifications": [{
17                       "$type": "ModelsLibrary.Verifications.Code",
18                       "TargetCode": 200}, ...],
19                "ExternalVerifications": [],
20                "TestResults": null,
21                "RequestMetadata": null,
22                "StressTimes": null }]}],
23    "MissingTests": [{
24          "TestID": "https://petstore3.swagger.io/api/v3/pet200",
25          ... }],
26    "GeneratedTests": [{
27          "TestID": "https://petstore3.swagger.io/api/v3/pet200",
28          ... }]}
```

4.6.2 Suporte a Múltiplos Tipos de Linguagem

Certas funcionalidades na aplicação usufruem do suporte de múltiplos tipos de linguagem, nomeadamente, as verificações de *Schema* e *Match* como o *Retain*. Para estas funcionalidades foram suportadas as linguagens de JSON e XML sendo importante uniformizar a forma como é verificado que linguagem foi utilizada tendo como prioridade não ter código repetido por todas as suas utilizações como a fácil introdução de outra linguagem suportada em trabalho futuro. Para este efeito foi tirado partido do padrão de desenho *Factory*, que consiste na definição de um contrato, onde diversas subclasses o implementam, onde existe outra classe responsável por decidir que subclasse é que é realmente instanciada. A Figura 4.19 apresenta o modelo.

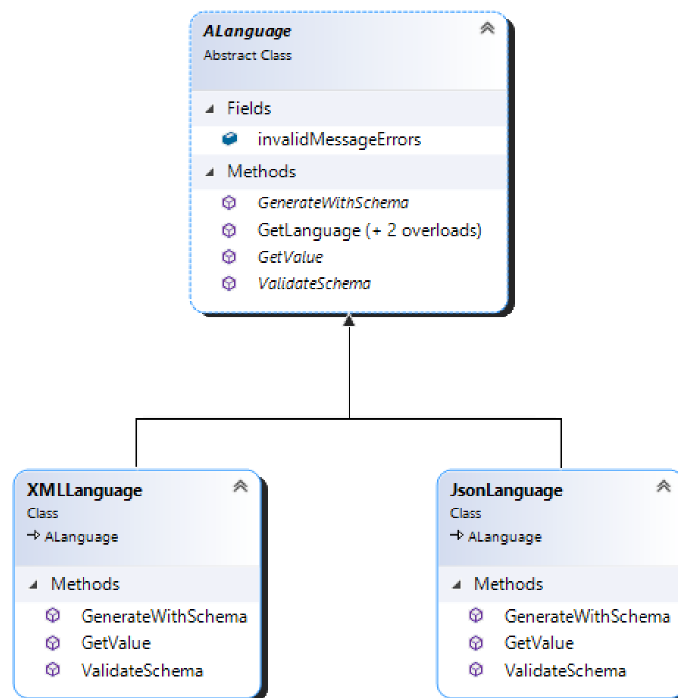


Figura 4.19: Modelo para o Suporte a múltiplos tipos de linguagem.

Nesta variação do padrão *Factory* a classe *ALanguage* funciona tanto como o contrato como a classe que instância a subclasse correta. O método *GetLanguage* é responsável por identificar qual o tipo de linguagem a instanciar e contém dois *overloads* devido às diferentes fases onde é necessário identificar a linguagem. Uma linguagem necessita de dar *override* a três métodos distintos, nomeadamente o *GenerateWithSchema*, que através de um esquema definido na linguagem deve gerar um objeto desse esquema com valores aleatórios, *GetValue*, que através de um *Path* retorna o valor associado e *ValidateSchema* que valida se um objeto obedece ao esquema especificado. Deste forma, a introdução de suporte a outras linguagens é bastante simplificado.

4.6.3 Execução dos Testes - *Worker Service*

O segundo serviço, ao receber uma mensagem proveniente do serviço anterior, que contém apenas o *id* da `API` de forma a identificar que testes devem ser realizados, inicia um processo de três passos:

Reload External Validations De forma a suportar o *TypeNameHandling auto* do *serializer* do `JSON` as bibliotecas externas têm que ser primeiro carregadas neste contexto aplicacional, pois antes estas encontravam-se em contextos diferentes, nomeadamente o do outro serviço.

Deserialização do ficheiro de teste Este processo consiste na deserialização do ficheiro resultante do serviço anterior, onde, tirando partido das funcionalidades do *JsonSerializer*, este processo é totalmente automático, instanciando todos os modelos necessários.

Execução dos testes O aspeto principal da execução dos testes baseia-se sobre a construção e envio de pedidos `HTTP`. Para este propósito, foi recorrida à biblioteca *System.Net.Http* [19], mais especificamente a classe *HttpClient*. Esta classe permite a criação de uma sessão possibilitando o envio de múltiplos pedidos `HTTP` de forma assíncrona. Foi criada uma classe *singleton*, *HttpUtils*, responsável por criar um *HttpClient* e realizar pedidos `HTTP`.

Para os testes, foi criada uma classe para cada tipo, nomeadamente a classe *RunWorkflow* que suporta a execução de um *workflow*, a classe *AutoGeneratedTests* que suporta a execução dos testes gerados de forma automática e a classe *StressTests* que tirando partido da classe *RunWorkflowMultiple*, sendo que esta é uma extensão da classe *RunWorkflow*, suporta a execução de testes de carga. Estas são de forma geral semelhantes, no entanto contêm algumas diferenças chave, particularmente as classes *RunWorkflow* e por extensão *RunWorkflowMultiple* têm que suportar a funcionalidade de *Retain* quando for necessário dentro de cada *workflow*, em contraste com a classe *AutoGeneratedTests* que não suporta *Retain* e que caso o *Path* contenha uma variável, esta é simplesmente substituída por um valor aleatório. A classe responsável pelos testes de carga apresenta também outra diferença, sendo que esta não executa nenhuma verificação, pois os testes de carga focam-se simplesmente nos tempos de resposta.

A Listagem 4.2 apresenta pseudocódigo do método responsável pela realização dos testes na sua totalidade.

Listagem 4.2: Método responsável pela realização dos testes.

```
1  procedure RunTests(Workflows){
2    List<Tasks> tasks;
3    tasks.Add(RunGeneratedTests())
4    loop Workflows
5      tasks.Add(RunWorkflow())
6      if(hasStressTest)
7        tasks.Add(RunStressTests())
8
9    await all tasks;
10   WriteReport(); }
```

Devido à natureza assíncrona dos pedidos `HTTP`, é necessário guardar uma lista de

todas as tarefas assíncronas para, de seguida, poder esperar que estas acabem (linha 2). De seguida, é sinalizado para correr todos os testes de geração automática, adicionando as tarefas assíncronas à lista (linha 3). Naturalmente, é de seguida percorrido todos os *workflows* e adicionado as tarefas à lista (linhas 4 e 5) e caso o *workflow* contenha um teste de carga, esse é também realizado (linhas 6 e 7). O método finaliza com a espera assíncrona da conclusão de todas as tarefas e com a escrita de um ficheiro em formato JSON que é depois guardado na base de dados com todos os resultados dos testes. A Listagem 4.3 apresenta um exemplo de relatório final de um teste.

Listagem 4.3: Exemplo do ficheiro de resultados de teste.

```
{    "Errors": 59,
     "Warnings": 49,
     "date": "2022-07-02T22:55:09.3293678+01:00",
     "WorkflowResults": [{
             "WorkflowID": "crud_pet",
             "Tests": [{
                     "TestID": "createPet",
                     ...
                     "TestResults": [{
                             "TestName": "Code",
                             "Success": true,
                             "Description": null
                         },{
                             "TestName": "Schema",
                             "Success": true,
                             "Description": null
                      }],
                     "RequestMetadata": {
                         "Method": "POST",
                         "URI": "https://petstore3.swagger.io/api/v3/pet",
                         "Headers": ...,
                         "Body": ...,
                         "ResponseCode": 200,
                         "ResponseBody": ...,
                         "ResponseHeaders": ...,
                         "ResponseTime": 868
                     },
                     "StressTimes": [ 988, 106, 103, 102, 96, 103, 117, ...]
                 },]}],
     "GeneratedTests": [{
             "TestID": "https://petstore3.swagger.io/api/v3/pet200", ... }]}
```

O teste apresentou 59 falhas de verificações, 49 avisos e a data a que foi realizado (linha 1, 2 e 3), o que pode significar possíveis erros na `API` em relação à sua especificação. De seguida os *workflows* são apresentados em detalhe, dando um ênfase aos resultados das verificações, linhas 9 a 17, onde é possível de observar que duas verificações foram bem sucedidas. Depois dos resultados é apresentada a secção de *metadata* do pedido `HTTP` onde está presente toda a informação tanto do pedido como da resposta: Método, *Uniform Resource Identifier* (`URI`), *Headers* e *Body* do pedido e Codigo, *Body*, *Headers* e tempo da resposta (linhas 18 a 27). Está também associado a cada teste os tempos de resposta de cada vez que este pedido foi realizado num contexto de teste de carga (linha 28). Por fim, está também presente o mesmo tipo de dados para os testes gerados (linhas 30 a 31).

4.7 *Monitor Tests*

Nesta página, o utilizador tem acesso aos seus testes configurados, bem como os detalhes dos mesmos. É possível realizar um conjunto de operações sobre esses testes, sendo estas:

1. Apagar a configuração;
2. Editar a configuração;
3. Descarregar os resultados do último teste;
4. Forçar a realização de testes;
5. Observar os resultados do último teste;

Estas operações vão ser explicadas em maior detalhe nas secções seguintes.

Na Figura 4.20 podemos observar a página de monitorização de um utilizador, a imagem apresenta uma página com 6 testes configurados, onde o utilizador está a observar os detalhes da configuração *PetStore* onde se revelam as ações possíveis.

Apagar a configuração Como a configuração apenas está presente na base de dados, não existindo qualquer informação no *file system* ou outro repositório de dados, a remoção de uma configuração limita-se à remoção da sua entrada na base de dados e de todas as entradas dependentes. A aplicação cliente limita-se a enviar o pedido com o *id* de qual `API` deve ser removida.

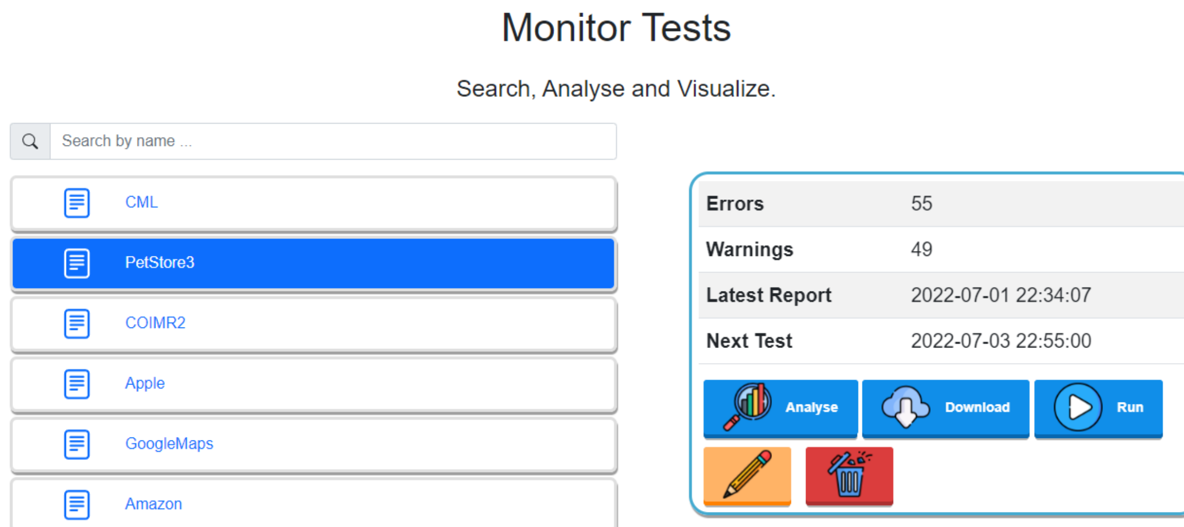


Figura 4.20: Página para a monitorização dos testes.

Editar a configuração Existe a possibilidade de editar a configuração, o utilizador pode então fornecer um diferente nome, que é o nome demonstrado na `GUI`. É no entanto importante mencionar que como trabalho futuro seria interessante suportar edições mais relevantes à configuração como atualizar a especificação, atualizar os ficheiros `TSL` como também editar o intervalo de tempo entre testes.

Descarregar os resultados do último teste Um dos requisitos funcionais consiste na possibilidade de descarregar os resultados de um teste. A implementação por parte do servidor é bastante direta, este apenas devolve o ficheiro `JSON` com os resultados, logo, apenas a parte cliente vai ser explicada com mais detalhe. A Listagem 4.4 apresenta o código responsável por descarregar o relatório no lado do cliente.

Listagem 4.4: Método que realiza o *download* do relatório.

```
async DownloadReport(apiId, apiTitle, latestReportDate) {
    const token = await authService.getAccessToken();
  fetch(`MonitorTest/DownloadReport?`+new URLSearchParams({apiId:apiId,}),{
        method: 'GET',
        headers: !token ? {} : { 'Authorization': `Bearer ${token}` }
    }).then(response => {
        response.blob().then(blob => {
            let url = window.URL.createObjectURL(blob);
            let a = document.createElement('a');
            a.href = url;
            a.download = apiTitle + '_' + latestReportDate+'.json';
            a.click();
}); })  ;}
```

Quando o utilizador pressiona o botão para descarregar o relatório, é feito o pedido ao servidor para obter os resultados (linhas 2 a 5). De seguida, é criado um *Binary Large Object* (BLOB) que contém o ficheiro JSON gerado (linha 8), finalizando com a criação de um novo elemento HTML, *a*, que contém os campos *href* que aponta para o *blob* criado e o campo *download* para descarregar o ficheiro (linhas 9 a 12).

Forçar a realização de testes Devido à natureza da aplicação, alguns testes podem ser realizados enquanto o utilizador está nesta página seja devido a testes previamente agendados como devido ao botão que força a realização de testes. Isto implica que a página deve ser atualizada automaticamente, sem necessidade de um *refresh* por parte do utilizador. Para resolver este problema quando o utilizador se encontra na página, é criado um *setInterval* que a cada minuto realiza o mesmo pedido ao servidor que foi realizado quando a página foi carregada, essencialmente recarregando a página completa, atualizando o conteúdo caso algum teste tivesse sido realizado. O intervalo de tempo pode numa versão futura ser uma variável que dependendo de quando forem realizados os próximos testes realiza o pedido apenas nessas alturas. No entanto, os pedidos também terminam se o utilizador mudar de página pois não é necessário saber se foi realizado algum teste estando noutra página. Quando o utilizador volta a aceder à página do ambiente de trabalho, o intervalo é criado novamente.

Observar os resultados do último teste Ao clicar no botão para observar o último teste, o utilizador é redirecionado para uma página que demonstra em detalhe todos os resultados obtidos. A página está dividida em quatro secções principais:

1. *Overview*, com todos os dados principais sobre os resultados;
2. TSL *Workflows*, focando nos resultados dos ficheiros TSL;
3. *Generated Tests*, focando nos resultados dos testes gerados automaticamente;
4. *Missing Tests*, detalhando os testes em falta dos ficheiros TSL;

Na secção *Overview*, o utilizador é apresentado com um resumo de todos os dados importantes, sendo estes: o número total de erros, o número total de avisos, o número de *workflows*, o número de testes gerados de forma automática e o tempo total que o teste levou a completar. Esta apresenta também um gráfico de pizza, mostrando o número de verificações bem sucedidas em comparação com as que falharam e um gráfico que demonstra o tempo que cada bloco principal levou a completar. A Figura 4.21 apresenta a página de *Overview* para um teste realizado sobre a loja de animais *PetStore*.

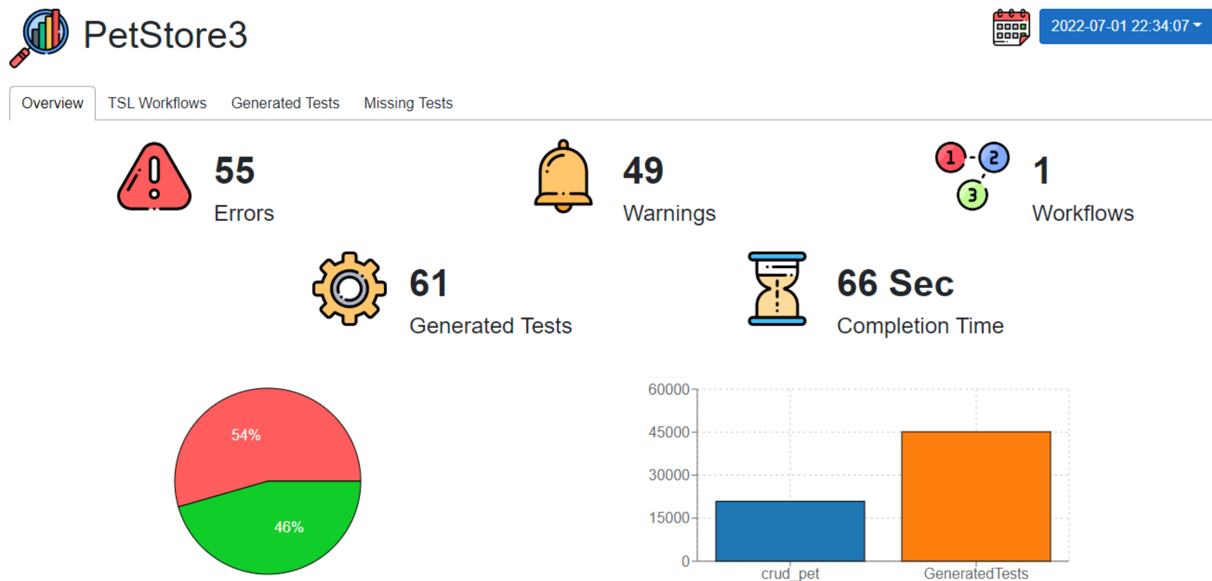


Figura 4.21: *Overview* do relatório.

Na secção `TSL` *Workflows*, o utilizador tem acesso a detalhes sobre todos os *workflows* realizados, como os seus testes de carga e verificações. A Figura 4.22 apresenta os detalhes para o *workflow* `crud_pet`, o *worflow* é representado como um círculo azul, seguido de todos os testes desse *workflow*, representados em círculos cinzentos.

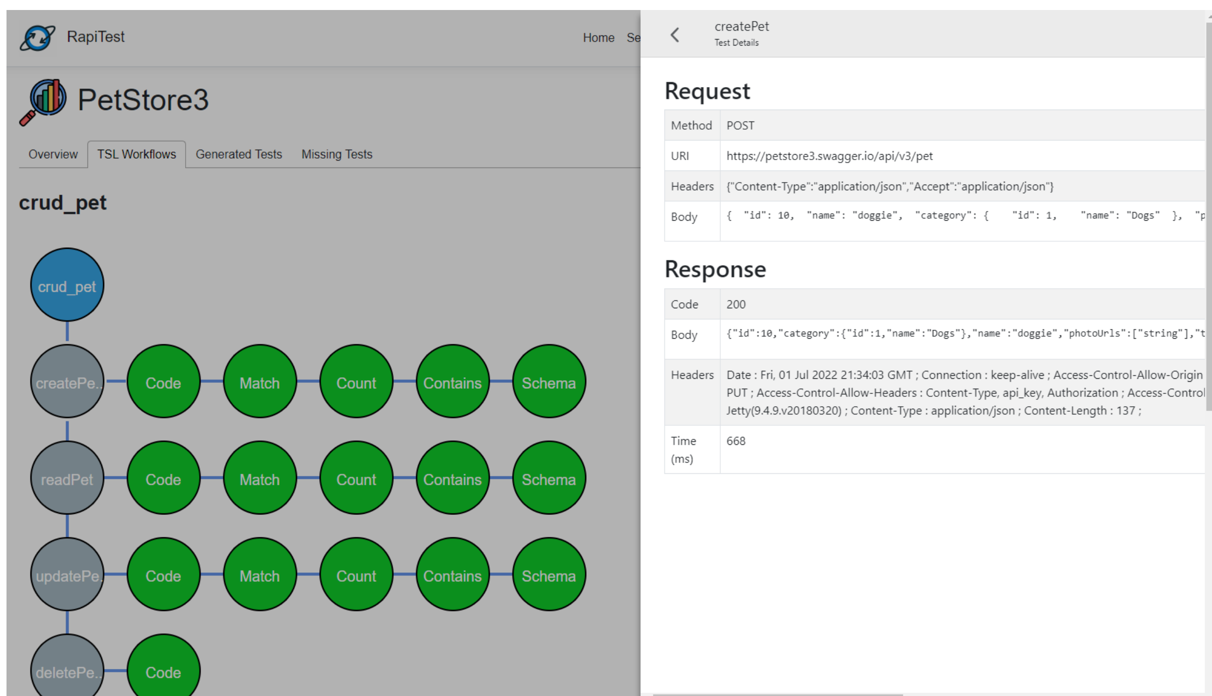


Figura 4.22: Resultados dos *workflows*.

Todas as verificações de um teste são representadas horizontalmente em verde ou vermelho, dependendo se foi ou não bem sucedida, respetivamente. Cada um dos círculos é *clickable*, sendo que ao clicar sobre ele é fornecido um certo tipo de informação através do painel que aparece à direita. Ao clicar sobre o *workflow* em azul um resumo dos testes e verificações é apresentado, se for clicado sobre um teste em cinzento é apresentado detalhes sobre o pedido `HTTP` e a sua resposta, e se for clicado sobre uma verificação detalhes sobre esta aparecem. Nesta caso, na Figura 4.22, está a ser apresentado os detalhes do teste *createPet*.

Caso o *workflow* contenha testes de carga os seus dados são também apresentados abaixo. A Figura 4.23 apresenta os resultados do teste de carga que foi realizado, como também alguns dados relevantes sobre tempos de resposta. Estes dados, apresentados na tabela, são relativos a cada teste, sendo estes o menor e maior tempo de resposta, o tempo médio, o tempo do último teste e o número de vezes que o pedido foi realizado. É também apresentado num gráfico os tempos individuais de resposta de todos os pedidos, sendo que cada entrada no eixo horizontal representa a execução do *workflow* na sua totalidade com os quatro testes, tendo no total 40 execuções do *workflow*.

Stress Test

Tests	Min (ms)	Max (ms)	Average (ms)	Last (ms)	Count
createPet	102	752	177.95	105	40
readPet	86	111	103.65	104	40
updatePet	102	134	111.7	106	40
deletePet	64	119	104.3	105	40

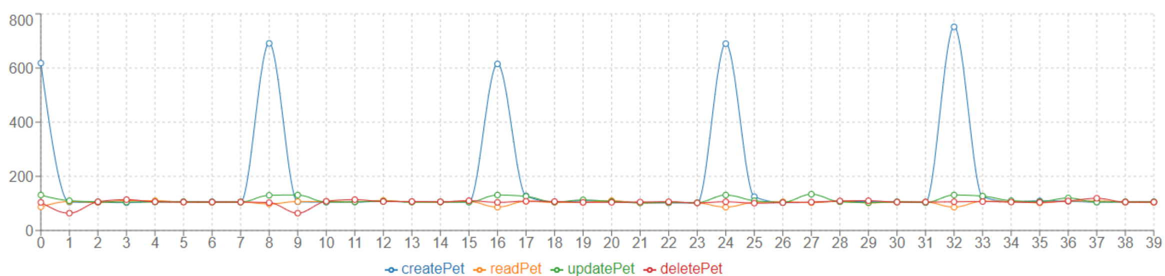


Figura 4.23: Resultados dos testes de carga.

Na secção *Generated Tests* são apresentados os resultados dos testes gerados de forma automática. Esta secção é bastante semelhante à secção `TSL` *Workflows*, focando apenas nos testes gerados. A Figura 4.24 apresenta um excerto dos testes gerados com algumas verificações bem sucedidas e outras falhadas.

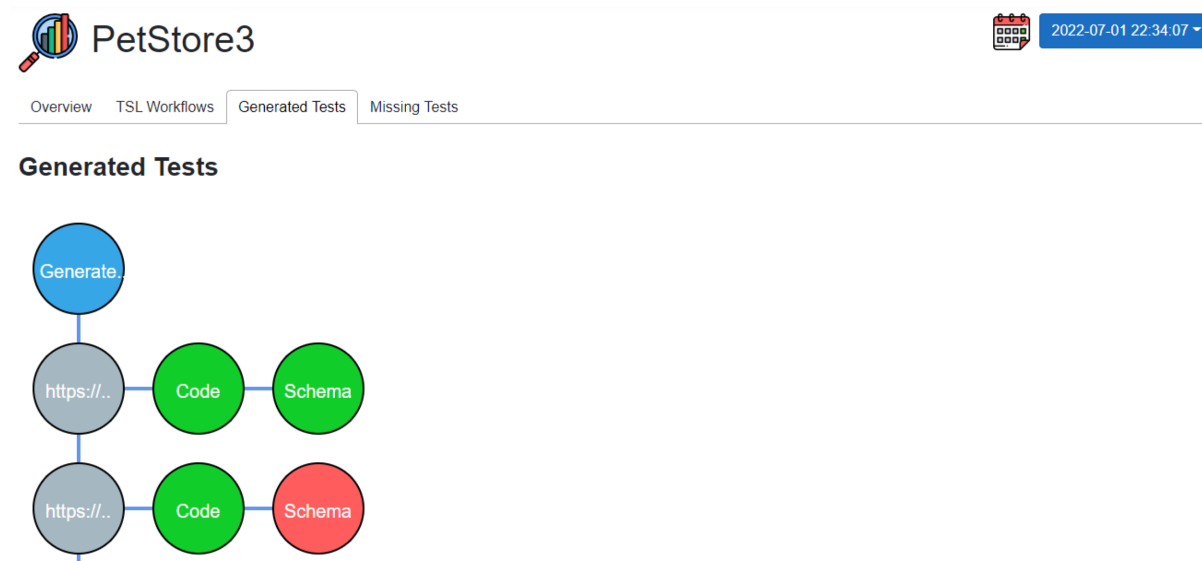


Figura 4.24: Resultados dos testes gerados.

Na secção *Missing Tests* são apresentadas todas as combinações de método, *Path* e *Headers* (diferentes *Content-Type* e *Accept*) que não foram diretamente testadas através dos ficheiros TSL. Esta página apresenta também uma funcionalidade bastante relevante, sendo esta a geração automática do ficheiro TSL para todos estes testes em falta. A Figura 4.25 apresenta a página da secção *Missing Tests*.

PetStore3

2022-07-01 22:34:07

Overview | TSL Workflows | Generated Tests | Missing Tests

Missing Tests

Generate TSL

Tests	Method	URI	Headers	Code
0	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/json","Accept":"application/xml"}	200
1	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/json","Accept":"application/json"}	200
2	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/xml","Accept":"application/xml"}	200
3	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/xml","Accept":"application/json"}	200
4	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/x-www-form-urlencoded","Accept":"application/xml"}	200
5	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/x-www-form-urlencoded","Accept":"application/json"}	200
6	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/json","Accept":null}	400
7	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/xml","Accept":null}	400
8	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/x-www-form-urlencoded","Accept":null}	400
9	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/json","Accept":null}	404
10	Put	https://petstore3.swagger.io/api/v3/pet	{"Content-Type":"application/xml","Accept":null}	404

Figura 4.25: Testes em falta.

A Listagem 4.5 apresenta um excerto do ficheiro TSL gerado automaticamente.

Listagem 4.5: Excerto de ficheiro TSL gerado automaticamente.

```
1  - WorkflowID: "MissingTestsTSL"
2    Stress: null
3    Tests:
4      - TestID: "https://petstore3.swagger.io/api/v3/petPutapplication/
   xmlapplication/json200"
5        Server: "https://petstore3.swagger.io/api/v3"
6        Path: "/pet"
7        Method: "Put"
8        Headers:
9          - "Content-Type:application/json"
10         - "Accept:application/xml"
11       Body: "{\r\n  \"id\": 1,\r\n  \"name\": \"sample\",...}"
12       Retain: null
13       Query: null
14       Verifications:
15         - Code: 200
16           Schema: null
17           Match: null
18           Contains: null
19           Count: null
20           Custom: null
21     - TestID: "https://petstore3.swagger.io/api/v3/petPutapplication/
   jsonapplication/json200"
22       Server: "https://petstore3.swagger.io/api/v3"
23       ...
```

Este consiste em apenas um *workflow* composto por todos os testes em falta. O ficheiro contém o mínimo necessário para ser válido e pode ser diretamente utilizado numa configuração de um teste. Este contém diversos campos com valor *null*, sendo que estes não são obrigatórios e relativamente complicados (ou mesmo impossíveis) de gerar de forma automática. São no entanto mantidos no ficheiro para notificar o utilizador da sua existência, sendo que a sua utilização enriquece significativamente a qualidade dos testes.

Todos os resultados de todos os testes são mantidos na base de dados incluindo testes anteriores. Uma das funcionalidades consiste na observação dos resultados de testes anteriores através do calendário no canto superior direito da página.

5

Deploy e Resultados

Neste capítulo, é especificado na secção 5.1, o processo para o *deploy* da aplicação e de todos os seus componentes, de forma a ser possível testar a aplicação em qualquer máquina, independentemente do sistema operativo. Na secção 5.2 é apresentado como a aplicação foi testada, incluindo os resultados obtidos desses testes como também do *feedback* dos utilizadores sobre a aplicação.

5.1 *Deploy*

O *deploy* de uma solução informática é onde se cruzam e surgem a maioria dos problemas transversais entre o desenvolvimento de software e as operações (*DevOps*) [33]. De forma a facilitar o processo de instalação da aplicação noutra máquina, usou-se a tecnologia *Docker* [16]. Esta funciona à base da tecnologia de contentores [29], que incluem apenas a aplicação, binários e bibliotecas que a aplicação possa depender, apresentado na Figura 5.1. A Figura 5.2 apresenta a mesma aplicação, recorrendo, no entanto, a máquinas virtuais para a instanciação. A diferença mais significativa entre as duas abordagens é a necessidade de instanciar por completo um novo sistema operativo na solução com máquinas virtuais. Já na solução com contentores, como estes tiram partido do sistema operativo da máquina, tornam-se mais rápidos de criar e executar.

A tecnologia de contentores fornece diversas vantagens, tais como:

- Portabilidade, onde após testar a aplicação dentro de um contentor, é possível

Figura 5.1: Estrutura *Docker*.

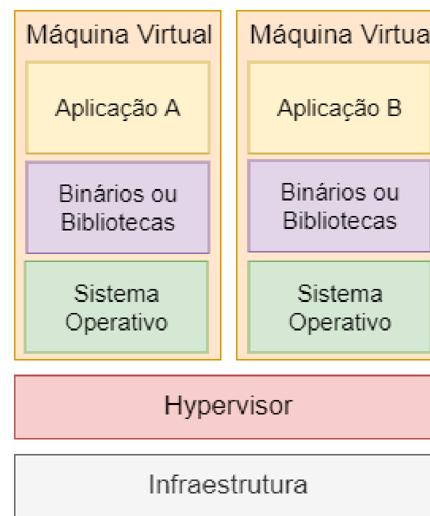


Figura 5.2: Estrutura máquinas virtuais.

fazer *deploy* em qualquer outra máquina com *Docker*, tendo a garantia que a aplicação irá correr exatamente da mesma maneira;

- Desempenho, onde comparando com a alternativa das máquinas virtuais, *Docker* tira partido do sistema operativo da própria máquina, resultando num tamanho de imagem significativamente menor, como também maior rapidez na sua criação e execução;

- Isolamento, contentores estão completamente isolados de outros contentores, útil para garantir que um certo contentor contém as versões de bibliotecas que necessita, e outro contentor mais atualizado, pode conter as mesmas com outra versão;

- Escalabilidade, sendo possível rapidamente adicionar contentores dependendo das necessidades, oferecendo opções para gerir diversos contentores.

Para criar um contentor *Docker*, a maneira mais simples é através de um *Dockerfile*. Este ficheiro de texto contém todos os comandos que um utilizador teria que executar numa linha de comandos para a criação da imagem. Foi criado um *Dockerfile* por cada aplicação, nomeadamente *RapiTest* (aplicação *web*), *SetupTestsWorkerService* (validar ficheiros de *input* e criar o ficheiro de teste) e *RunTestsWorkerService* (executar os testes e validações). A Listagem 5.1 apresenta o *Dockerfile* para a aplicação *web RapiTest*. Sendo esta uma aplicação *web* com servidor em *.NET Core*, é primeiro definida a imagem base onde esta se baseia, neste caso uma imagem que contém o *runtime .NET Core* e *ASP.NET* (linha 1). Nas linhas 3 e 4 é definido o porto exposto pelo servidor para a comunicação com o cliente, neste caso o porto 80.

Listagem 5.1: Ficheiro *Docker* para *RapiTest*.

```
FROM mcr.microsoft.com/dotnet/aspnet:3.1 AS base
WORKDIR /app
ENV ASPNETCORE_URLS=https://+:80
EXPOSE 80
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y libpng-dev libjpeg-dev curl libxi6 build-essential
    libgl1-mesa-glx
RUN curl -sL https://deb.nodesource.com/setup_lts.x | bash -
RUN apt-get install -y nodejs
FROM mcr.microsoft.com/dotnet/sdk:3.1 AS build
RUN apt-get update
RUN apt-get install -y curl
RUN apt-get install -y libpng-dev libjpeg-dev curl libxi6 build-essential
    libgl1-mesa-glx
RUN curl -sL https://deb.nodesource.com/setup_lts.x | bash -
RUN apt-get install -y nodejs
WORKDIR /src
COPY ["RAPITest/RAPITest.csproj", "RAPITest/"]
COPY ["ModelsLibrary/ModelsLibrary.csproj", "ModelsLibrary/"]
RUN dotnet restore "RAPITest/RAPITest.csproj"
COPY . .
WORKDIR "/src/RAPITest"
RUN dotnet build "RAPITest.csproj" -c Release -o /app/build
FROM build AS publish
RUN dotnet publish "RAPITest.csproj" -c Release -o /app/publish
FROM base AS final
WORKDIR /app
COPY RAPITest/certs/certificate.pfx ./certs/
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "RAPITest.dll"]
```

De seguida, pois esta aplicação tem uma componente cliente baseada em *NodeJS*, é necessário instalar o *NodeJS* no contentor (linhas 5 a 15) juntamente com o *sdk* do *dotnet*. O projeto e a biblioteca que o projeto depende são copiados para o contentor (linhas 17 e 18), para de seguida compilar o projeto. É também copiado o certificado necessário para suportar comunicação HTTPS (linha 27). Este certificado foi criado através da *toolkit* OpenSSL [43] utilizando o comando *openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365*.

O certificado tem o formato X.509 que é o *standard* para certificados utilizados no protocolo HTTPS. Contém uma chave pública, uma entidade e está ou não assinado por uma autoridade de certificação. Neste caso, o certificado gerado é *self-signed*, ou seja,

não está assinado através de uma autoridade de certificação, não sendo ideal mas, suficiente para os objetivos. A chave privada é criada através do algoritmo `RSA`, contém 4096 *bits* e é válida durante 365 dias. O *Dockerfile* finaliza com o *entrypoint*, que sinaliza que o contentor deve correr como um executável, neste caso, o ficheiro `DLL` gerado.

Um *Dockerfile* para as outras duas aplicações foi também criado, seguindo, de forma geral, o apresentado acima, tirando as partes relativas à componente cliente. Tendo as três imagens criadas estas foram colocados no *Dockerhub* [15] todas em repositórios públicos. Desta forma estas podem ser facilmente acedidas de qualquer máquina.

5.1.1 Docker Compose

A arquitetura total necessita de 5 componentes principais. O *RapiTest*, os dois *Worker Services*, a base de dados *SQL Server* e o serviço de mensagens *RabbitMQ*. O *RapiTest* e os *Worker Services* já foram colocados no *Dockerhub*, sendo que imagens da base de dados *SQL* e do serviço *RabbitMQ* estão já disponibilizadas no repositório, a Figura 5.3 apresenta a arquitetura da Figura 3.9 mas em formato de contentores *Docker*.

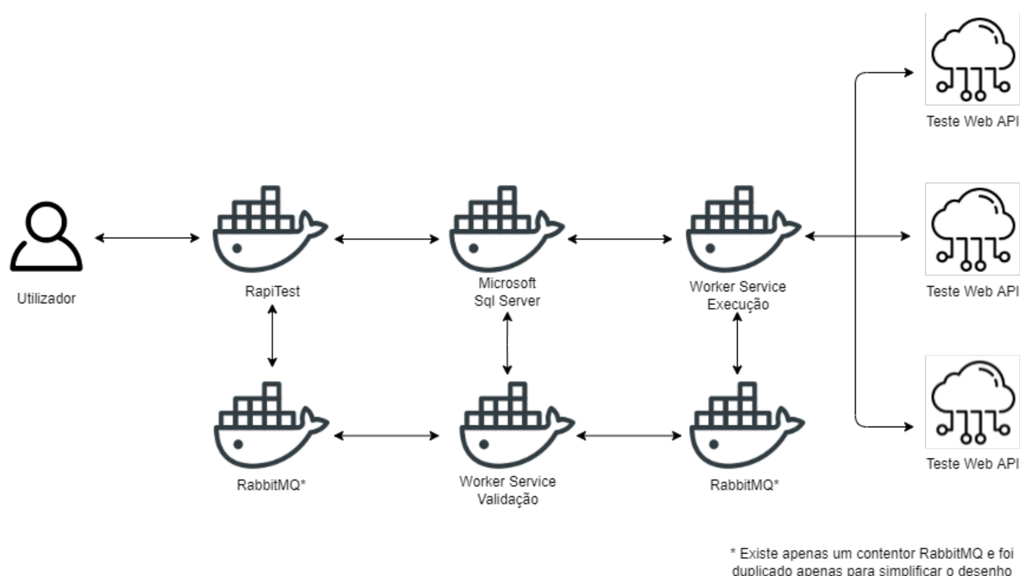


Figura 5.3: Arquitetura através de contentores *Docker*.

Para facilitar a criação e execução de todos estes serviços recorreu-se à funcionalidade *Docker Compose*. Esta funcionalidade permite a definição de serviços *multi-container*, simplificando toda a parte de execução. *Docker Compose* tira partido de um ficheiro `YAML` que descreve todos os contentores e detalhes sobre estes. A Listagem 5.2 apresenta um excerto do ficheiro *Docker Compose* criado para as cinco componentes.

Listagem 5.2: Ficheiro *Docker Compose*.

```
rabbitmq: # login guest:guest
image: rabbitmq:3-management
hostname: "rabbitmq"
labels:
    NAME: "rabbitmq"
ports:
    - "4369:4369"
    ...
restart: always
db:
image: "mcr.microsoft.com/mssql/server"
environment:
    SA_PASSWORD: "Your_password123"
    ACCEPT_EULA: "Y"
restart: always
rapitest:
image: duartefelicio/rapitest:latest
depends_on:
    - db
    - rabbitmq
ports:
    - "8080:80"
restart: always
runtestsworkerservice:
image: duartefelicio/runtestsworkerservice:latest
depends_on:
    - db
    - rabbitmq
restart: always
setuptestsworkerservice:
image: duartefelicio/setuptestsworkerservice:latest
depends_on:
    - db
    - rabbitmq
restart: always
```

Nas linhas 1 a 9, temos a descrição do contentor que corresponde ao serviço *RabbitMQ*. A linha 2, descreve a imagem que deve ser utilizada do *Dockerhub*, seguido de *metadata* sobre os nomes desta imagem (linhas 3 a 5). De seguida, temos a descrição dos portos, cada entrada consiste em dois portos separados por *":"*, onde o porto da esquerda representa o porto exposto para a máquina que está a hospedar estes contentores, e o porto da direita representa o porto interno do contentor. Na linha 9 é sinalizado que

que caso o contentor apresente algum erro que leva à sua terminação, este deve fazer *restart*.

De seguida, é possível observar a descrição do serviço da base de dados *SQL Server* (linhas 10 a 15), apresentando principalmente a definição da password para se poder conectar à base de dados.

Nas linhas 16 a 23, temos a descrição da aplicação *web RapiTest*. Esta apresenta o conceito de *depends_on* (linha 18), neste caso da base de dados e do serviço de mensagens, pois esta necessita destes serviços para o correto funcionamento. No entanto, esta funcionalidade não é totalmente útil, pois apenas garante que este serviço é começado depois dos outros dois, não garantindo que os outros dois já estão prontos, pois existe uma grande distinção entre "começou a execução" e "está pronto". Os outros dois serviços também apresentam esta necessidade e devido a este fator quando um serviço é inicializado, este fica em modo *stand-by* aguardando a completa inicialização da base de dados e do serviço de mensagens, e só depois é que realmente "estão prontos". Na linha 22 é apresentado os portos do *RapiTest*, como foi mencionado anteriormente no *Dockerfile* a aplicação expõe o seu conteúdo no porto 80, sendo este o porto interno do contentor, para de seguida se poder aceder à aplicação na máquina hospedeira é necessário utilizar o porto 8080. Após todos os serviços estarem prontos um utilizador que estiver a testar a aplicação localmente pode então aceder à aplicação através do URL: *https://localhost:8080*. O repositório público do projeto, juntamente com os ficheiro *Docker* e *DockerCompose* encontram-se disponíveis no *Github* [37].

5.2 Resultados

Como foi mencionado na introdução, a aplicação *RapiTest* foi desenvolvida numa parceria entre a CML e o ISEL. Desde 2017, a CML usa a plataforma de dados urbanos PGIL para gerir alguns verticais da cidade, com diversos serviços integrados, alguns dos quais são fluxos de dados em tempo real [56]. A PGIL conta hoje com mais de 900 processos de integração, muitos dos quais têm API *web* como fonte de dados. Assim, a validação e supervisão são necessários para que as fontes de dados sejam consistentes com a especificação.

A CML esteve diretamente envolvida ao longo do desenvolvimento da aplicação, que foi demonstrada várias vezes a vários funcionários da CML para fins de *feedback* e orientação. O *feedback* sobre a interface do utilizador foi extremamente positivo devido à ênfase na simplicidade, no entanto, em relação à funcionalidade, os utilizadores finais sentiram que era necessário um amplo conhecimento do domínio do problema,

incluindo os novos ficheiros `TSL`. Isso resultou no suporte da criação de tais ficheiros através da própria `GUI` a fim de simplificar a experiência do utilizador, exigindo menos conhecimento técnico para operar corretamente a aplicação *web*.

De forma a testar todos os casos de utilização suportados com todos os tipos diferentes de teste foi tirado partido de três `API`. Uma delas consiste na `API` pública para testes fornecida pela *Swagger*, *petstore*, apresentada anteriormente, e as outras duas `API` foram fornecidas pela `CML`, *PGIL-01* e *PGIL-02* para propósito de testar a aplicação com `API` reais que estão neste momento em pré-produção.

A Tabela 5.1 apresenta dados sobre a `API` da loja de animais, nomeadamente todos os *endpoints*, os métodos suportados nesses *endpoints*, como também o número total de combinações de testes possíveis para esse *endpoint* tendo em consideração o formato de *input*, *output* e os possíveis códigos de resposta.

Tabela 5.1: Dados sobre a `API` *petstore*.

Endpoint	Método HTTP	Combinações de pedidos
/pet	PUT	15
	POST	9
/pet/findByStatus	GET	3
/pet/findByTags	GET	3
/pet/{petId}	GET	4
	POST	1
	DELETE	1
/pet/{petId}/uploadImage	POST	1
/store/inventory	GET	1
/store/order	POST	6
/store/order/{orderId}	GET	4
	DELETE	2
/user/createWithList	POST	2
/user/login	GET	3
/user/{username}	GET	4
	DELETE	2

É possível de observar que esta `API` contém diversos *endpoints* diferentes, abrangendo os métodos `HTTP` principais e conta com um total de 61 combinações possíveis de pedidos, tornando esta `API` num bom candidato para testar a aplicação. É também relevante mencionar que alguns *endpoints* na secção que define as possíveis respostas

do servidor usufruem da *keyword: default*. Esta *keyword* quando utilizada significa que todos os códigos de resposta retornados pelo pedido que não se encontram explicitamente especificados devem respeitar a estrutura de resposta nesse campo. Ao utilizar este campo num *endpoint* está a ser implicitamente especificado que a resposta pode retornar qualquer código o que resultaria num teste para cada código, quando muito provavelmente apenas uma fração de todos os códigos poderia ser realmente retornada. Tendo isso em consideração este campo foi ignorado tanto na implementação como nos testes, no entanto a sua inclusão seria relevante como trabalho futuro.

A Tabela 5.2 apresenta os mesmos dados para uma das API da CML, neste caso uma API para integração entre o COI e a aplicação de Meios e Recursos.

Tabela 5.2: Dados sobre a API *PGIL-01*.

Endpoint	Método HTTP	Combinações de pedidos
/coi/entidadesCML/{de}/{ate}	GET	1
/coi/recursosHumanos/{de}/{ate}	GET	1
/coi/entidadesExternas/{de}/{ate}	GET	1
/coi/meiosMateriais/{de}/{ate}	GET	1

Esta API pode inicialmente parecer simples com apenas quatro *endpoints* sendo todos o método HTTP *Get* e apresentando no total quatro combinações diferentes. No entanto, devido às variáveis presentes em cada *endpoint* a criação de testes torna-se menos banal.

A Tabela 5.3 apresenta os mesmos dados para a última API da CML que trata com dados de sensores sobre a cidade de Lisboa.

Tabela 5.3: Dados sobre a API *PGIL-02*.

Endpoint	Método HTTP	Combinações de pedidos
/sensor/{sensorId}/info	GET	3
/{sensorType}/{avgSpan}	GET	3
/lastmeasurment/{locationId}	GET	3
/lastmeasurments	GET	3

Esta API, de forma semelhante à anterior, apesar de parecer simples apresenta grande complexidade devido aos campos presentes nos *endpoints*, dificultando significativamente a realização de testes sobre estas API.

Todos os testes apresentados nas secções seguintes foram realizados sobre a aplicação *deployed* em contentores numa máquina com as seguintes características relevantes apresentadas na Tabela 5.4.

Tabela 5.4: *Hardware* do computador para testes.

CPU	RAM	Memória
Intel(R) Core(TM) i7-4790K CPU @ 4.00GHz	16 GB	120 GB SSD

5.2.1 Testes sem `TSL`

Um dos suportes chave da aplicação é a geração automática de testes sem ser necessário fornecer um ficheiro `TSL`, para este efeito a aplicação foi avaliada na geração e execução para as três `API` apresentadas anteriormente. As métricas avaliadas consistem no número de testes gerados, comparados com o total de testes apresentado acima, o número de validações totais e o número total das bem sucedidas. A Tabela 5.5 apresenta os resultados dos testes sem ficheiro `TSL` para a `API` *petstore*.

Tabela 5.5: Resultados sem `TSL` - *petstore*.

Testes Gerados	Testes Possíveis	Validações Sucedidas
61	61	41.33%

Os objetivos foram alcançados em termos da geração de testes, gerando 100% dos testes possíveis. Em relação à qualidade dos testes gerados estes revelam alguns problemas, sendo que apenas cerca de 41% das validações foram bem sucedidas. Como foi mencionado anteriormente, isto deve-se ao facto da geração ser feita sem grande processamento, pois foi apenas realizada como prova de conceito. Fatores como pedidos com parâmetros serem sempre um valor inteiro aleatório, e os objetos gerados conterem apenas valores *default*, resulta nesta baixa taxa de sucesso.

A Tabela 5.6 apresenta os mesmos resultados para ambas as `API` da `CML`.

Tabela 5.6: Resultados sem `TSL` - *PGIL-01* e *PGIL-02*.

	Testes Gerados	Testes Possíveis	Validações Sucedidas
PGIL-01	4	4	0%
PGIL-02	12	12	0%

É possível observar que todas as validações falharam, isto deve-se ao facto que ambas as `API` exigem uma chave específica (*api key*) para ser possível de as aceder, o que

invalida imediatamente todas as abordagens que usam exclusivamente a OAS, pois essas chaves privadas não são fornecidas na especificação. Contudo, elas podem ser especificadas em ficheiros TSL para testar corretamente as API.

Nestes cenários foi também testada com sucesso a funcionalidade de geração automática do ficheiro TSL. Todas as API geraram um ficheiro TSL com o número total de testes possíveis, sendo que este necessita de ser editado manualmente pelo utilizador para criar testes relevantes.

Em conclusão estes testes gerados de forma automática são testes simples sendo que não suportam diretamente qualquer tipo de teste especifico como *workflow*, carga e segurança. Esta funcionalidade obteve resultados ótimos na geração dos diferentes testes, sendo que deve ainda ser melhorada em trabalho futuro na geração de dados de teste através de outras técnicas como *fuzzing*, análise de pedidos anteriores, entre outros.

5.2.2 Testes com TSL

Nesta secção, os testes realizados tiram partido da funcionalidade principal da aplicação, o suporte a ficheiros TSL. Esta secção encontra-se divida nos tipos principais de teste onde são apresentados os resultados e conclusões sobre a aptidão tanto da ferramenta como dos ficheiros TSL em cada tipo de teste.

Validação Para a API da CML *PGIL-01* foi criado um ficheiro TSL com oito *workflows* com um teste cada um. Quatro dos testes contêm a chave da API e os outros quatro não a contêm. Cada teste contém cinco validações, *Code*, *Match*, *Count*, *Contains* e *Schema*, um exemplo de como as descrever num ficheiro TSL pode ser observado na Listagem 3.2. Todos os *endpoints* contêm como parâmetros no *path* duas datas que representam o range de tempo de dados que o utilizador quer obter. Tendo isto em consideração todos os dados utilizados nas validações foram testados anteriormente com datas do passado de forma a garantir e gerar validações expectáveis de suceder. Em relação à validação do esquema foi tirado partido da funcionalidade dos ficheiros TSL de verificar o esquema com o que é fornecido na OAS. A Tabela 5.7 apresenta os resultados obtidos da execução deste ficheiro TSL.

Tabela 5.7: Resultados com TSL - Validação *PGIL-01*.

	Testes	Validações Sucedidas
Com api key	4	80%
Sem api key	4	0%

Em relação aos testes onde a *api key* se encontrava presente 80% das validações foram bem sucedidas, neste caso as validações que não forem bem sucedidas foram todas as validações de esquema. Nenhum dos esquemas se encontrava em conformidade com os objetos de resposta reais devido a vários problemas. Concretamente, certos campos da resposta apresentavam o valor *null*, mas não foram especificados como *nullable* no esquema. A Figura 5.4 apresenta os resultados de um teste onde a validação do esquema não foi bem sucedida, representada em vermelho. Também é possível visualizar o porquê da validação não ter sido bem sucedida no painel à direita analisando a descrição. Neste caso, vários campos tinham valor nulo enquanto era esperado algum outro tipo, sendo um problema do esquema descrito na especificação.

Figura 5.4: Descrição da razão da falha da verificação do esquema - *PGIL-01*.

Sobre os testes onde a *api key* não foi fornecida, naturalmente todos os testes falham, pois o servidor devolve código de erro (401) com mensagem de erro. Isto no entanto releva outra inconsistência na especificação, neste caso esta define em todos os pedidos apenas uma possível resposta do servidor, código 200 com um certo esquema, o que se encontra incompleto, revelado por este mesmo teste.

A `API` *PGIL-02* foi também testada com um respetivo ficheiro `TSL` criado manualmente e foram encontradas mais inconsistências na especificação. Nomeadamente o mesmo problema de esquemas incompletos ou com tipos errados, como pode ser observado na Figura 5.5. Outra inconsistência foi novamente o retorno do código 401 quando a *api key* não se encontra presente, sendo que essa resposta não está na especificação.

Figura 5.5: Descrição da razão da falha da verificação do esquema - *PGIL-02*.

A API *petstore* apresenta os mesmos problemas derivados de especificações incompletas, onde por exemplo um dos pedidos especifica que como resposta o servidor apenas retorna o código 405 de erro, o que não é o caso na presença de um *input* válido.

Workflow Para testar o correto funcionamento de testes de *workflow* é necessário que exista inter-dependência de diferentes pedidos, como uma simples lógica de criação, obtenção, modificação e remoção de um recurso (CRUD). Ambas as API da CML são compostas de apenas métodos GET, não existindo qualquer dependência entre eles, logo, este tipo de testes foi realizado tirando partido da API *petstore*. Esta API, como foi mencionado anteriormente, simula uma loja de animais, e contém *endpoints* CRUD para animais de estimação. Foi então realizado um ficheiro TSL com um *workflow* contendo quatro testes, primeiro a criação do *pet* guardando o seu *id* proveniente da resposta, seguido da obtenção, alteração e remoção em função do *id*, seguindo a lógica apresentada na Figura 3.1. Um exemplo deste *workflow*, nomeadamente criar e atualizar um *pet* pode ser observado na Listagem 3.2. Os resultados foram positivos e o *workflow* foi executado com sucesso, onde todos os testes foram executados em sequência e todas as validações bem sucedidas, como pode ser observado na Figura 5.6.

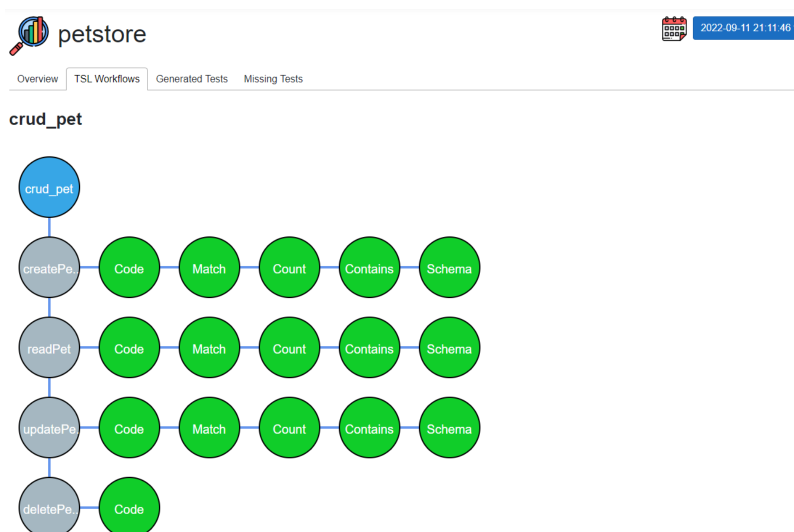


Figura 5.6: Resultados dos testes de *workflow* - *petstore*.

Carga Um dos elementos mais importantes de teste para API é o seu comportamento em situações de número elevado de pedidos em simultâneo. Foi testada a API da CML *PGIL-01* tirando partido do ficheiro TSL descrito anteriormente, nomeadamente quatro *workflows* com um teste cada, sendo que os testes sem *api key* foram removidos pois não acresentam valor a este teste. Foi de seguida acrescentada a configuração de testes de carga para realizar cada *workflow* 100 vezes, dividido em 10 fios de execução por

workflow, com um tempo de espera entre cada *workflow* do mesmo fio de execução de cinco segundos. Um exemplo de como descrever testes de carga pode ser observado na Listagem 3.2, neste exemplo o teste é descrito com 40 execuções, em cinco fios de execução sem nenhum tempo de espera. Os resultados são apresentados na Tabela 5.8.

Tabela 5.8: Resultados dos testes de Carga - *PGIL-01*.

Workflow	Repetições	Threads	Delay	Tempo Total de Execução
entidadesCOI	100	10	5 seg	7 minutos
recursosHumanosCOI	100	10	5 seg	9.3 minutos
entidadesExternasCOI	100	10	5 seg	8.2 minutos
meiosMateriaisCOI	100	10	5 seg	6.6 minutos

Somando os tempos totais de cada *workflow* obtém-se cerca de 31 minutos de execução, no entanto, devido ao paralelismo da execução, o tempo real observado para a conclusão dos testes foi de apenas 2 minutos para a realização de 400 pedidos. Este tempo de execução tão curto deve-se também ao facto da `API` conter um limite para o número de pedidos realizados pelo mesmo utilizador (ou fio de execução, neste caso), sendo uma funcionalidade utilizada em múltiplas `API` com o objetivo de se precaver contra este tipo de pedidos em barragem. Este fenómeno é possível de ser observado através dos dados apresentados na Tabela 5.9.

Tabela 5.9: Estatísticas do teste de carga - *PGIL-01*.

Tempo em ms	**entidades**	**recursosHum**	**entidadesExt**	**meiosMat**
Média	2222.1	1856.8	3851.7	2573.3
Valor Máximo	39013	33502	39077	37479
Valor Mínimo	34	29	42	41
Desvio Padrão	7348.3	5471.6	8975.5	7651.2
Número de dados	100	100	100	100

É possível de observar na tabela uma grande discrepância entre os valores de tempos mínimos e máximos de resposta, observando também um valor de desvio padrão elevado, o que demonstra a grande variância nos tempos de resposta. Grande parte dos pedidos obteve uma resposta quase instantânea (cerca de 40 ms), no entanto, nesses casos o servidor devolve o código de resposta 429 *Too Many Requests*. Esta funcionalidade deve ser melhorada em trabalho futuro, alterando dinamicamente o tempo entre pedidos sempre que o servidor devolve um 429, de forma a analisar o tempo de

atendimento de cada pedido como também o tempo de espera entre pedidos, conseguindo determinar o número de pedidos que o servidor suporta num dado tempo. É relevante mencionar que não é possível de observar o resultado em detalhe de cada pedido, sendo a única informação mostrada o tempo de resposta. Esta é uma limitação da aplicação e a sua melhoria seria um trabalho futuro relevante. A resposta de 429 em grande parte dos pedidos foi verificada manualmente durante a realização dos testes.

Outro aspeto relevante a testar é o desempenho da própria aplicação quando múltiplos testes são realizados simultaneamente. Para este efeito. foi configurado o mesmo teste à API *PGIL-01* duas vezes e foram corridos simultaneamente, tirando partido da funcionalidade *Run Now* para correr imediatamente os testes. Neste cenário ambos os testes completaram com sucesso, cerca de 3 minutos após o seu começo, comparado aos 2 minutos de apenas um teste. Foi feito novamente o teste mas desta vez para três em simultâneo, o que resultou no sucesso de todos os testes demorando novamente cerca de 3 minutos para completarem. Naturalmente, se forem realizados números muito elevados de testes o serviço que os executa pode sofrer problemas de desempenho ou mesmo *crash* total do serviço, no entanto, graças ao desacoplamento destas funcionalidades, torna-se relativamente simples acrescentar mais instâncias do mesmo serviço caso seja necessário num cenário real.

Segurança Na secção 3.1.4 foram apresentados os 10 principais riscos de segurança presentes em API, como foi mencionado todos os riscos que não requerem direto acesso ao código fonte para testar são possíveis de realizar através de ficheiros TSL, abaixo são apresentadas descrições de como o fazer.

1. *Broken Object Level Authorization* é possível de testar através de ficheiros TSL colocando manualmente no pedido um *id* aleatório diferente do *id* do objeto que o utilizador tem acesso, apresentado na Listagem 5.3 como exemplo.

Listagem 5.3: Exemplo de teste para *Broken Object Level Authorization*.

```
1  - WorkflowID: read_my_book
2    Tests:
3    - TestID: readBook
4      Server: "a_server"
5      Path: "/book/{id}"
6      Method: Get
7      Headers:
8        - Accept:application/xml
9      Verifications:
10       - Code: 200
```

Ao colocar manualmente um valor aleatório no campo *id* seria possível testar se se tem acesso aos livros de outro utilizador validando com resposta de sucesso. Este teste poderia ser suportado automaticamente pelo ficheiro `TSL` em trabalho futuro, por exemplo, acrescentando outro campo nas verificações que por estar presente, corre múltiplas vezes o pedido com valores aleatórios no *id* à procura de respostas válidas.

2. *Excessive Data Exposure* é também possível de testar com ficheiros `TSL`, neste caso é diretamente suportado através de uma validação do esquema do objeto que, no caso de existirem campos não especificados presentes no objeto da resposta, resultam na falha da verificação.

3. *Broken Function Level Authorization* é também possível de ser feito com ficheiros `TSL`, neste caso criando testes que realizam pedidos a certos *endpoints* que necessitam de autorizações especiais, verificando depois se devolvem resposta de sucesso.

4. *Broken User Authentication* poderia também ser testado criando um teste que tenta fazer *login* com credenciais aleatórias ou roubadas validando se o servidor devolveu resposta de sucesso, como apresentado na Listagem 5.4 para um exemplo de tentativa de *login* com credencias roubadas.

Listagem 5.4: Exemplo de teste para *Broken User Authentication*.

```
1  - WorkflowID: login-BUA
2   Tests:
3   - TestID: login
4       Server: "a_server"
5       Path: "/login"
6       Query:
7         - username=<stolen_username>
8         - password=<stole_password>
9       Method: Get
10      Headers:
11        - Accept:application/xml
12      Verifications:
13        - Code: 200
```

5. *Lack of Resources and Rate Limiting* é possível tirando partido da funcionalidade de testes de carga e foi inclusivamente testada como foi apresentado na secção anterior, onde grande parte das respostas devolvem código de erro devido ao número elevado de pedidos. A Listagem 5.5 apresenta o ficheiro `TSL` do teste de carga anterior.

Listagem 5.5: Exemplo de teste para *Lack of Resources and Rate Limiting*.

```
1  – WorkflowID: workflowEntidades
2     Stress:
3        Count: 100
4        Threads: 10
5        Delay: 5000
6     Tests:
7     - TestID: entidadesCOI
8        Server: "a_server"
9        Path: "a_path"
10       Method: Get
11       Headers:
12          - Accept:application/json
13          - Ocp–Apim–Subscription–Key:<key>
14       Verifications:
15          - Code: 200
```

6. *Mass Assignment* pode também ser testado colocando como exemplo campos como *isAdmin=true* em testes de criação de utilizadores, como é apresentado na Listagem 5.6, sendo de seguida necessário verificar se o utilizador ficou de facto com permissões de administrador.

Listagem 5.6: Exemplo de teste para *Mass Assignment*.

```
1  – WorkflowID: create_user
2     Tests:
3     - TestID: create_a_user
4        Server: "a_server"
5        Path: "/user"
6        Method: Post
7        Body: "$ref/dictionary/user"
8        Retain:
9           - petId#$.id
10       Verifications:
11          - Code: 200
12 //Ficheiro Dictionary
13 dictionaryID:user {
14    "username": "theUser",
15    "email": "john@email.com",
16    "isAdmin": true }
```

7. *Security Misconfiguration*, apesar de não ser possível de verificar diretamente com ficheiros `TSL`, é no entanto possível criar testes com campos intencionalmente

inválidos, sendo depois necessário verificar manualmente a resposta do servidor para esses testes de forma a observar se este devolveu detalhes sensíveis da aplicação.

8. *Improper Assets Management* é também possível de testar com ficheiros `TSL`, manualmente colocando servidores distintos do real, tentando encontrar outro de versão anterior, um exemplo pode ser observado na Listagem 5.7.

Listagem 5.7: Exemplo de teste para *Improper Assets Management*.

```
- WorkflowID: improper_assets_managemet
  Tests:
  - TestID: readPet
    Server: <random_similiar_server>
    Path: "a_path"
    Method: Get
    Headers:
      - Accept:application/xml
    Verifications:
      - Code: 200
```

Esta funcionalidade podia também ser implementada como trabalho futuro de forma semelhante à *Broken Object Level Authorization*, onde a sua presença nas verificações provoca diversos pedidos a servidores com valores semi-aleatórios em função do real.

9. *Injection* são possíveis, tentando injetar comandos através de parâmetros, como é possível observar na Listagem 5.8, onde é tentado injetar o comando para apagar a tabela *Users* através dos parâmetros do pedido, sendo também possível colocar outro teste para verificar se foi realmente removida.

Listagem 5.8: Exemplo de teste para *Injection*.

```
- WorkflowID: read_my_pet
  Tests:
    - TestID: readPet
      Server: "https://petstore3.swagger.io/api/v3"
      Path: "/pet?petId=10; DROP TABLE Users"
      Method: Get
      Headers:
        - Accept:application/xml
      Verifications:
        - Code: 200
```

10. *Insufficient Logging and Monitoring* é a única que não é possível de ser testada com ficheiros TSL, pois necessita de acesso aos *logs* gerados pelo servidor, o que naturalmente não está disponível.

Custom A funcionalidade e utilidade de verificações externas *custom* foram também testadas, para este efeito, foi criada uma validação *custom*, apresentada na Listagem 5.9, onde foi de seguida gerado o ficheiro DLL.

Listagem 5.9: Validação *Custom*.

```
public class CustomVerification : Verification{
    public Task<Result> Verify(HttpResponseMessage Response){
        Result res = new Result();
        res.TestName = "Custom";
        if ((int)Response.StatusCode == 200) {
            res.Success = true;
            res.Description = "Custom Success";}
        else {
            res.Success = false;
            res.Description = "Custom Failure";}
        return Task.FromResult(res); }}
```

A validação é notavelmente simples, servindo apenas como demonstração do seu funcionamento. O utilizador, tendo acesso à resposta do pedido, pode realizar qualquer tipo de validação que desejar dentro do método. A Figura 5.7 apresenta o resultado de sucesso da validação externa.

Figura 5.7: Resultados da validação *custom*.

6

Conclusões e Trabalho Futuro

6.1 Conclusões

No início deste projeto foi identificado um problema: "Dificuldade em devidamente testar API tendo acesso apenas à OAS". Foi identificado que muitas das vezes estas OAS se encontram incompletas, o que resulta em testes gerados incompletos. Testes gerados de forma automática também apresentam diversas limitações, como na identificação de *workflows*, no uso de dados corretos em pedidos com variáveis, como em pedidos com corpo.

O objetivo deste projeto foi desenvolver uma aplicação *web* que tomando partido da OAS, mas, também de uma nova DSL chamada *Test Specification Language* (TSL), suporte casos de teste personalizados. A aplicação *web* permite a configuração de várias verificações nativas, relativas à segurança e exatidão das respostas, enquanto executa os testes em intervalos regulares, como a cada 24 horas. Dessa forma, a API pode ser supervisionada continuamente para garantir o seu correto funcionamento.

No início do trabalho, foram avaliadas algumas ferramentas já desenvolvidas no sentido de identificar funcionalidades já existentes e por forma a contribuir com algo inovador. Depois deste estudo, foram escolhidas as tecnologias apresentadas atrás. Seguidamente foram implementadas as funcionalidades pretendidas, nomeadamente 1) desenvolvimento de uma DSL que permita a especificação de casos de teste personalizados, 2) a parte de autenticação de um utilizador utilizando a *framework IdentityServer*, 3) configuração de testes para um API, suportando o *upload* direto de ficheiros TSL, ou

então da sua criação diretamente na aplicação, e, por fim 4) a página onde é possível monitorizar os testes configurados, como também analisar os resultados em detalhe.

Concluída a aplicação, foi possível colocá-la em modo de produção através de imagens *Docker*, onde foi necessário tirar partido do *Docker-Compose* para instanciar as diferentes imagens e garantir a sua comunicação. Este cenário de produção foi testado sobre as suas diferentes funcionalidades comprovando o funcionamento correto da aplicação.

6.2 Trabalho Futuro

Melhorar a geração automática de testes A geração automática de testes é, neste momento, muito limitada, estando maioritariamente focado na total geração aleatória ou *default* de valores. Seria relevante incorporar técnicas apresentadas nos artigos de geração mais complexas, como a análise de respostas anteriores, análise dos *endpoints* para tentar encontrar dependências e *fuzzing* mais complexo. Outro aspeto relevante seria suportar o valor *default* na `OAS` na geração dos testes, tomando alguma liberdade como gerar testes apenas para os códigos mais expectados serem devolvidos dependendo do pedido.

Melhorar os testes de carga Um dos problemas com os testes de carga são as respostas de 429, alterando dinamicamente o tempo entre pedidos sempre que o servidor recebe um 429 estes podem ser evitados de forma mais eficaz, com o objetivo de conseguir determinar o número de pedidos num dado tempo que a `API` suporta. Outro aspeto é também melhorar a visualização dos resultados de carga, de forma a ser possível analisar individualmente cada pedido/resposta.

Expandir as verificações fornecidas nativamente Como apresentado nos resultados, suportar diretamente mais verificações seria outro aspeto relevante a adicionar, nomeadamente, validações de *SQL Injection*, *Cross Site Scripting*, *Boundary Scan*, *Broken Object Level Authorization*, *Improper Assets Management*, entre outros.

Sistema de notificações Quando um teste é executado e concluído, o utilizador não é notificado desse acontecimento, a interface simplesmente atualiza-se, no entanto, seria relevante ter um sistema onde o utilizador é notificado destes acontecimentos.

Email de confirmação Quando o utilizador se regista este deve fornecer o seu *email*, no entanto, nenhum *email* a confirmar é enviado, sendo esta outra funcionalidade relevante a acrescentar.

Possibilidade de melhor editar um teste já configurado Neste momento, o utilizador pode apenas editar o nome de um teste configurado, no entanto seria relevante possibilitar a edição de outros dados, como a `OAS` caso tenha sido atualizada ou os ficheiro `TSL` com alguma alteração.

Expansão da criação de ficheiros `TSL` através da `GUI` Neste momento, a criação de ficheiros `TSL` pela aplicação não suporta todas as funcionalidades, sendo relevante expandir a criação para funcionalidades como o *Retain* e outras validações nativas.

Referências

[1] Andy Neumann, Nuno Laranjeiro & Jorge Bernardino, "An Analysis of Public REST Web Service APIs", *IEEE Transactions on Services Computing*, vol. 14, n.º 4, páginas 957–970, 2021. DOI: 10.1109/TSC.2018.2847344.

[2] Alberto Martin-Lopez, Sergio Segura, Carlos Muller & Antonio Ruiz-Cortes, "Specification and Automated Analysis of Inter-Parameter Dependencies in Web APIs", *IEEE Transactions on Services Computing*, páginas 1–1, 2021. DOI: 10.1109/TSC.2021.3050610.

[3] Josué Alejandro Díaz-Rojas, Jorge Octavio Ocharán-Hernández, Juan Carlos Pérez-Arriaga & Xavier Limón, "Web API Security Vulnerabilities and Mitigation Mechanisms: A Systematic Mapping Study", em *2021 9th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 2021, páginas 207–218. DOI: 10.1109/CONISOFT52520.2021.00036.

[4] OASIS. "Advanced Message Queuing Protocol". (2012), URL: https://web.archive.org/web/20141010172100/http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html. Accessed: 10-12-2021.

[5] Laranjeiro, Nuno and Agnelo, João and Bernardino, Jorge. "bBOXRT". (2022), URL: https://git.dei.uc.pt/cnl/bBOXRT/tree/e5d329133d51aa75cd39209590cac7046d0640b1/src/main/java/pt/uc/dei/rest_api_robustness_tester. Accessed: 24-06-2022.

[6] Andrea Arcuri, "Automated black-and white-box testing of restful apis with evomaster", *IEEE Software*, vol. 38, n.º 3, páginas 72–78, 2020.

[7] Vaggelis Atlidakis, Patrice Godefroid & Marina Polishchuk, "Restler: Stateful rest api fuzzing", em *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, páginas 748–758.

[8] Kaspersky Lab. "What's a Brute Force Attack?" (2020), URL: https://www.kaspersky.com/resource-center/definitions/brute-force-attack. Accessed: 01-05-2022.

[9] CENTERIS. (2022), URL: http://centeris.scika.org/?page=home. Accessed: 30-09-2022.

[10] "Dredd — HTTP API Testing Framework". (2021), URL: https://dredd.org/en/latest/index.html. Accessed: 09-12-2021.

[11] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo & Jordi Cabot, "Automatic generation of test cases for REST APIs: A specification-based approach", em *2018 IEEE 22nd international enterprise distributed object computing conference (EDOC)*, IEEE, 2018, páginas 181–190.

[12] Amazon. "Amazon Web Services". (2022), URL: https://aws.amazon.com/pt/. Accessed: 30-06-2022.

[13] Google. "AngularJS". (2021), URL: https://angularjs.org/. Accessed: 27-06-2022.

[14] Api Blueprint. "api blueprint". (2021), URL: https://apiblueprint.org/. Accessed: 24-06-2022.

[15] Dockerhub. "Dockerhub". (2022), URL: https://hub.docker.com/. Accessed: 05-07-2022.

[16] Docker. "Docker". (2022), URL: https://www.docker.com/. Accessed: 05-07-2022.

[17] Tilde Inc. "Ember". (2022), URL: https://emberjs.com/. Accessed: 27-06-2022.

[18] Gluster. (2022), URL: https://www.gluster.org/. Accessed: 25-09-2022.

[19] Microsoft. "System.Net.Http Namespace". (2022), URL: https://docs.microsoft.com/en-us/dotnet/api/system.net.http?view=net-6.0. Accessed: 02-07-2022.

[20] Knockoutjs. "Knockout". (2022), URL: https://knockoutjs.com/. Accessed: 27-06-2022.

[21] Meteor Software. "Meteor". (2022), URL: https://www.meteor.com/. Accessed: 27-06-2022.

[22] Microsoft. “OpenAPI.NET”. (2022), URL: https://github.com/microsoft/OpenAPI.NET. Accessed: 28-06-2022.

[23] Meta Platforms, Inc. “React”. (2022), URL: https://reactjs.org/. Accessed: 26-06-2022.

[24] aaubry. (2022), URL: https://www.nuget.org/packages/YamlDotNet. Accessed: 18-09-2022.

[25] Roy Fielding & Architectural Styles, “The design of network-based software architectures”, *Doctoral Dissertation, School of Information and Computer Science*, 2000.

[26] Patrick Lambert. “How to filter user input: An overview”. (2012), URL: https://resources.infosecinstitute.com/topic/how-to-filter-user-input/. Accessed: 24-02-2022.

[27] Gil Fink & Ido Flatow, *Pro Single Page Application Development*. Springer, 2014.

[28] Brock Allen & Dominick Baier. “Identity Server”. (2020), URL: https://identityserver4.readthedocs.io/en/latest/. Accessed: 07-06-2022.

[29] Shashank Mohan Jain, “Linux Containers and Virtualization”, *A Kernel Perspective*, 2020.

[30] Tony Tam. “Swagger Core”. (2022), URL: https://github.com/swagger-api/swagger-core. Accessed: 24-06-2022.

[31] Nuno Laranjeiro, João Agnelo & Jorge Bernardino, “A Black Box Tool for Robustness Testing of REST Services”, *IEEE Access*, vol. 9, páginas 24 738–24 754, 2021.

[32] LoadView. “loadview”. (2022), URL: https://www.loadview-testing.com/. Accessed: 24-06-2022.

[33] Lucy Ellen Lwakatare, Terhi Kilamo, Teemu Karvonen, Tanja Sauvola, Ville Heikkilä, Juha Itkonen, Pasi Kuvaja, Tommi Mikkonen, Markku Oivo & Casper Lassenius, “DevOps in practice: A multiple case study of five companies”, *Information and Software Technology*, vol. 114, páginas 217–230, 2019, ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2019.06.010. URL: https://www.sciencedirect.com/science/article/pii/S0950584917302793.

[34] Alberto Martin-Lopez, Sergio Segura & Antonio Ruiz-Cortés, “A catalogue of inter-parameter dependencies in RESTful web APIs”, em *International Conference on Service-Oriented Computing*, Springer, 2019, páginas 399–414.

[35] ——, "RESTest: automated black-box testing of RESTful web APIs", em *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, páginas 682–685.

[36] Snehal Mumbaikar, Puja Padiya et al., "Web services based on soap and rest principles", *International Journal of Scientific and Research Publications*, vol. 3, n.º 5, páginas 1–4, 2013.

[37] Duarte Felício. "RapiTest". (2022), URL: https://github.com/ISEL-DEETC/RAPITest. Accessed: 30-09-2022.

[38] Microsoft. ".NET". (2022), URL: https://dotnet.microsoft.com/. Accessed: 26-06-2022.

[39] "5 Examples of APIs We Use in Our Everyday Lives". (2021), URL: https://nordicapis.com/5-examples-of-apis-we-use-in-our-everyday-lives/. Accessed: 25-11-2021.

[40] IETF OAuth Working Group. "OAuth 2.0". (2020), URL: https://oauth.net/2/. Accessed: 01-05-2022.

[41] Peter Oehlert, "Violating assumptions with fuzzing", *IEEE Security & Privacy*, vol. 3, n.º 2, páginas 58–62, 2005.

[42] "What Is OpenAPI?" (2021), URL: https://swagger.io/docs/specification/about/. Accessed: 09-12-2021.

[43] OpenSSL. "OpenSSL Cryptography and SSL/TLS". (2022), URL: https://www.openssl.org/. Accessed: 05-07-2022.

[44] OWASP. "API Security Top 10 2019". (2019), URL: https://owasp.org/www-project-api-security/. Accessed: 25-06-2022.

[45] Claus Pahl & Pooyan Jamshidi, "Microservices: A Systematic Mapping Study.", em *CLOSER (1)*, 2016, páginas 137–146.

[46] OpenAPI Specification. "Swagger Petstore". (2022), URL: https://petstore.swagger.io/. Accessed: 14-01-2022.

[47] "Postman". (2021), URL: https://www.postman.com/. Accessed: 09-12-2021.

[48] "Search the Largest API Directory on the Web". (2021), URL: https://www.programmableweb.com/apis/directory. Accessed: 25-11-2021.

[49] Cristi Vîjdea. "drf-yasg". (2022), URL: https://github.com/axnsan12/drf-yasg. Accessed: 24-06-2022.

[50] RabbitMQ. "AMQP 0-9-1 Model Explained". (2020), URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html. Accessed: 04-05-2022.

[51] Sam Corcos. “HTML5 drag-drop zone with React.js: dropzone”. (2020), URL: https://github.com/react-dropzone/react-dropzone. Accessed: 01-05-2022.

[52] Martin-Lopez, Alberto and Segura, Sergio and Ruiz-Cortés, Antonio. “RESTest”. (2021), URL: https://github.com/isa-group/RESTest/. Accessed: 24-06-2022.

[53] Atlidakis, Vaggelis and Godefroid, Patrice and Polishchuk, Marina. “RESTler”. (2019), URL: https://github.com/microsoft/restler-fuzzer. Accessed: 24-06-2022.

[54] Viglianisi, Emanuele and Dallago, Michael and Ceccato, Mariano. “RestTestGen”. (2020), URL: https://github.com/resttestgenicst2020/submission_icst2020. Accessed: 24-06-2022.

[55] António Serrador, João Tremoceiro, Nuno Cota, Nuno Cruz & Nuno Datia, “iLX - A Success Case in Public Tender Methodology”, em *CENTERIS 2018 - Conference on ENTERprise Information Systems / ProjMAN 2018 - International Conference on Project MANagement / HCIST 2018 - International Conference on Health and Social Care Information Systems and Technologies*, 2018.

[56] António Serrador, João Tremoceiro, Nuno Cota, Nuno Cruz & Nuno Datia, “iLX- A Success Case in Public Tender Methodology”, em *ProjMAN 2018-International Conference on Project MANagement*, 2018.

[57] “SoapUI”. (2021), URL: https://www.soapui.org/tools/soapui/. Accessed: 09-12-2021.

[58] Neoteric in Medium. “Single-page application vs. multiple-page application”. (2020), URL: https://medium.com/@NeotericEU/single-page-application-vs-multiple-page-application-2591588efe58. Accessed: 01-05-2022.

[59] Microsoft. “SQL Server”. (2022), URL: https://www.microsoft.com/en-us/sql-server. Accessed: 26-06-2022.

[60] Uku Tomikas in Messente. “What Are the Benefits of Two-Factor Authentication?” (2019), URL: https://messente.com/blog/most-recent/benefits-of-two-factor-authentication. Accessed: 01-05-2022.

[61] Emanuele Viglianisi, Michael Dallago & Mariano Ceccato, “RestTestGen: automated black-box testing of RESTful APIs”, em *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, páginas 142–152.

[62] YAML. “YAML Ain’t Markup Language”. (2021), URL: https://yaml.org/. Accessed: 25-06-2022.