Editor para testes semi-automáticos de Web API

ALEXANDRE SANTOS ROCHA

Licenciado

Trabalho de Projeto para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

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

Júri:

Presidente: Prof. Doutor Nuno Miguel Machado Cruz
Vogais: Prof. Doutor Filipe Bastos de Freitas
Prof. Doutor Nuno Miguel Soares Datia

dezembro 2024

Editor para testes semi-automáticos de Web API

ALEXANDRE SANTOS ROCHA

Licenciado

Trabalho de Projeto para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Prof. Doutor José Manuel de Campos Lages Garcia Simão, ISEL
Prof. Doutor Nuno Miguel Soares Datia, ISEL

Júri:

Presidente: Prof. Doutor Nuno Miguel Machado Cruz, ISEL
Vogais: Prof. Doutor Filipe Bastos de Freitas, ISEL
Prof. Doutor Nuno Miguel Soares Datia, ISEL

dezembro 2024

Agradecimentos

Agradeço aos meus orientadores José Simão e Nuno Datia, pela ajuda e disponibilidade ao longo do projeto, mesmo em momentos de menor disponibilidade da minha parte. Tive o prazer de os ter como professores ao longo do meu percurso no ISEL e, sem tirar mérito aos restantes professores, foram sem dúvida dos melhores professores que tive e confesso que foram um fator que tive em conta na escolha do tema para o meu projeto final.

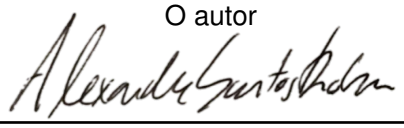
Agradeço à minha família o apoio e motivação que me deram ao longo do projeto, especialmente em momentos de maior carga de trabalho.

Agradeço aos meus colegas que me acompanharam ao longo do mestrado, em especial ao meu colega Filipe Mendes, com quem fiz todo o meu percurso no ISEL desde o inicio da licenciatura até ao fim do mestrado.

Declaração de integridade

Declaro que este trabalho de projeto é o resultado da minha investigação pessoal e independente. O seu conteúdo é original e todas as fontes listadas nas referências bibliográficas foram consultadas e estão devidamente mencionadas no texto. Mais declaro que todas as referências científicas e técnicas relevantes para o desenvolvimento do trabalho estão devidamente citadas e constam das referências bibliográficas.

O autor



Lisboa, 23 de outubro de 2024

Resumo

Nos dias que correm, as Web APIs são uma das formas mais comuns de realizar comunicação entre diversos serviços na Web. Embora existam diferentes estilos arquiteturais para o desenho de uma Web API, o padrão mais amplamente utilizado é o REST. Considerando o seu uso extensivo em inúmeras áreas e a sua importância, torna-se cada vez mais uma necessidade garantir o correto funcionamento das Web APIs, nomeadamente através da testagem contínua das mesmas. Num trabalho previamente realizado, foi desenvolvida a ferramenta *RapiTest*, com o objetivo de fornecer uma solução para o problema da testagem de RESTful APIs, seguindo uma abordagem caixa-preta. Esta ferramenta gera testes a partir da especificação da API, e, adicionalmente, faz uso da TSL, uma linguagem criada para permitir o desenvolvimento de casos de testes customizados. Este trabalho dá continuidade ao projeto anterior, focando-se em solucionar algumas das lacunas que a *RapiTest* apresentava, sendo a principal a falta de um editor gráfico para a elaboração de testes usando TSL. Com este objetivo, foi desenvolvido de raíz um editor interativo, utilizando uma abordagem *node-based* (ou *flow-based*), que visa tornar o processo de criação de testes fácil e intuitivo para qualquer utilizador, independentemente do seu nível de conhecimento técnico. Este permite não só a criação de configurações de teste como a edição de configurações previamente feitas, sendo capaz de carregar um ficheiro TSL e preencher o editor a partir do mesmo. Esta e outras funcionalidades desenvolvidas contribuem para acrescentar valor à *RapiTest*, de forma a solidificar a mesma como uma solução simples mas viável para a testagem de RESTful APIs.

Palavras-chave: Web API; RESTful API; Interface de utilizador; Editor gráfico de testes; Interface baseada em nós.

Abstract

Nowadays, Web APIs are one of the most common ways of communicating between various services on the Web. Even though there are many different architectural styles for designing a Web API, the REST pattern is the most widely used across the world. Considering its extensive use in numerous areas and its importance, it is becoming increasingly necessary to guarantee the quality and correct functioning of Web APIs, namely through continuous testing. In a previous work, the *RapiTest* tool was developed to provide a solution to the problem of testing RESTful APIs, following a black-box approach. This tool generates tests from the API specification, and, additionally, makes use of TSL, a language created to allow the creation of customized test cases. The present work continues the development of this tool, focusing on addressing some of the flaws that *RapiTest* presented, the main one being the lack of a dedicated graphical editor for test configurations using TSL. With this in mind, a new interactive editor was developed from scratch, using a *node-based* (or *flow-based*) approach, with the goal of making test creation an easy and intuitive process for any user, regardless of their technical knowledge. The editor allows not only the creation of new test configurations, but also the editing of previously made configurations, by loading an existing TSL file and populating the editor with its data. This and other developed functionalities contribute to add value to the *RapiTest* tool, solidifying it as a simple but reliable solution for testing RESTful APIs.

Keywords: Web API; RESTful API; User Interface; Graphical test editor; Node-based interface.

Índice

Índice de Figuras

Índice de Tabelas

Índice de Listagens

Glossário

Black-box	Método de testagem em que os detalhes de implementação não são conhecidos 2
Fuzzing	Técnica de testagem que envolve a inserção de dados aleatórios em inúmeros pedidos de forma a tentar encontrar falhas 5
RESTful	Que segue o estilo arquitetural REST 10
Single Page Application	Aplicação que carrega apenas uma página HTML e atualiza a mesma dinamicamente ao longo da utilização 21
White-box	Método de testagem em que os detalhes de implementação são conhecidos 2

Siglas

AMQP	Advanced Message Queuing Protocol 19
API	Application Programming Interface 1
CLS	Cummulative Layout Shift 45
CRUD	Create, Read, Update, Delete 1
HTTP	Hypertext Transfer Protocol 1
INP	Interaction to Next Paint 45
IoT	Internet of Things 7
JSON	JavaScript Object Notation 10
LCP	Largest Contentful Paint 45
NPM	Node Package Manager 38
OAS	OpenAPI Specification 2
REST	Representational State Transfer 1
SPA	Single Page Application 21
SQL	Structured Query Language 21
TSL	Test Specification Language 2
TTFB	Time to First Byte 45
URL	Uniform Resource Locator 1
YAML	YAML Ain't Markup Language 10

1 Introdução

Este capítulo introduz o tema do trabalho, nomeadamente o tópico da testagem de web APIs, e apresenta alguns conceitos importantes. São apresentadas as necessidades e desafios que constituem o problema a resolver, e, naturalmente, é apresentado o objetivo a que se propõe o trabalho. É apresentada a ferramenta desenvolvida previamente, algumas das suas características e quais os passos tomados para atingir os objetivos definidos. No fim do capítulo é também detalhada a estrutura do restante documento.

1.1 Problema

Atualmente, uma das formas mais comuns de realizar comunicação entre serviços, aplicações e outros componentes na web é através de web APIs [1]. Uma API (*Application Programming Interface*) é, como o nome indica, uma interface ou um contrato que permite a interação entre programas, sendo utilizada muitas vezes, por exemplo, como uma ponte entre aplicações cliente e servidores. Por exemplo, uma aplicação web a executar num browser faz muitas vezes uso de web APIs disponibilizadas por um servidor de forma a obter os dados necessários para apresentar ao utilizador.

Apesar de existirem várias formas de implementar uma web API, o estilo arquitetural mais comum na web é o padrão REST (*Representational State Transfer*) [2]. Este estilo define uma abordagem orientada aos recursos, sendo estes identificados através de URLs. Estes recursos são depois manipulados através de métodos HTTP como GET, POST, entre outros, muitas vezes representando as operações CRUD (*Create, Read, Update, Delete*) - por exemplo, GET corresponde a *Read*, POST corresponde a *Create*, etc. Devido à sua lógica, flexibilidade e simplicidade, o padrão REST é assim amplamente utilizado para o desenho de APIs, contribuindo para o seu uso cada vez mais omnipresente na web.

Posto isto, não é então surpreendente que haja uma necessidade cada vez mais crescente de garantir a qualidade das APIs desenvolvidas, nomeadamente no que toca a aspetos como a segurança, o desempenho, e até mesmo a conformidade com a sua própria documentação e especificação. Aspetos como a deteção de vulnerabilidades comuns, testes de carga e desempenho, ou testes automáticos gerados a partir da especificação tornam-se assim essenciais no que toca à testagem de web APIs.

1.2 Objetivo

A aplicação *RapiTest* [3] foi desenvolvida num projeto anterior com o intuito de fornecer uma solução para este problema, oferecendo uma abordagem *Black-box* para a testagem de APIs REST. Este tipo de abordagem é usado quando não são conhecidos os detalhes de implementação das APIs, sendo o foco principal nos *inputs* e *outputs* esperados. Existem também abordagens *White-box*, onde são conhecidos os detalhes de implementação (neste caso, das APIs). A ferramenta *RapiTest* aproveita a especificação da API (a especificação deve estar de acordo com a OAS - *OpenAPI Specification*) para gerar inúmeros testes automaticamente com base na mesma, bem como suporta a criação manual de *workflows* (no fundo, conjuntos de testes) através da TSL (*Test Specification Language*), uma nova linguagem desenvolvida especificamente para permitir este objetivo. Outro dos grandes objetivos da *RapiTest* é também o de fornecer uma experiência de utilizador simples e intuitiva, permitindo o uso da aplicação por parte de utilizadores com fraco conhecimento técnico.

Porém, a *RapiTest* não foi totalmente concluída no projeto anterior. Apesar de já ser um protótipo funcional, capaz de processar as especificações OAS e ficheiros TSL para produzir e executar configurações de teste, a *RapiTest* apresentava também diversas lacunas, nomeadamente no que toca à interface de utilizador e às funcionalidades suportadas. Relativamente às funcionalidades, alguns dos pontos de melhoria identificados incluem, por exemplo, um melhoramento no processo de geração automática de testes ou a extensão das verificações suportadas nativamente pela TSL.

No entanto, o principal problema era sem dúvida relativo à interface de utilizador, mais especificamente a interface de criação de testes. Como mencionado anteriormente, um dos objetivos chave da *RapiTest* é o de proporcionar uma experiência de utilizador simples e intuitiva independentemente do conhecimento técnico do utilizador. Desta forma, é imperativo que a interface de utilizador da aplicação siga estes princípios, o que não era de todo o caso no início deste trabalho. Apesar de existir uma interface para criação de testes, esta era bastante primitiva, sendo baseada em formulários, e requerendo ainda algum nível de conhecimento técnico para o seu preenchimento. Entre outras falhas, não existia inclusive forma de editar os ficheiros TSL na aplicação, tornando o cenário em que um utilizador possua um ficheiro TSL que pretende alterar, mas não possua o conhecimento técnico para o fazer manualmente, uma verdadeira catástrofe. Estes e outros aspetos impedem a ferramenta *RapiTest* de cumprir o seu objetivo, pelo que se torna óbvio que esta situação seria invariavelmente de prioridade máxima no que diz respeito ao trabalho a realizar.

Assim, o objetivo definido para o trabalho a realizar passou precisamente por colmatar as principais lacunas de forma continuar o desenvolvimento da *RapiTest*, concentrando os esforços na criação de um novo editor gráfico de testes, que irá inovar e melhorar a experiência de criação de *workflows* e testes dentro da aplicação, bem como possibilitar a edição de ficheiros já existentes. Isto, juntamente com os restantes requisitos, irão assim contribuir para atingir o objetivo de garantir que a *RapiTest* possa ser uma solução simples e acessível para a testagem *black-box* de web APIs Rest.

1.3 Organização do documento

Este documento está divido em 6 capítulos.

- O capítulo presente, o 1, trata de introduzir o problema em questão e descrever o objetivo do trabalho a realizar;
- O capítulo 2 descreve o trabalho relacionado, apresentando artigos, ferramentas e considerações acerca dos temas em questão;
- O capítulo 3 define a abordagem, nomeadamente relativamente aos tópicos da testagem de Web APIs, linguagem TSL e editor de testes, para além de apresentar a arquitetura da solução;
- O capítulo 4 descreve a implementação, descrevendo a aplicação na sua globalidade mas focando-se maioritariamente no editor de testes;
- O capítulo 5 apresenta uma análise dos resultados alcançados, tanto de forma qualitativa como quantitativa;
- O capítulo 6 é reservado para as conclusões e considerações acerca de trabalho futuro.

2 Trabalho Relacionado

Neste capítulo é apresentado um resumo da investigação feita relativamente ao trabalho relacionado com este projeto. Primeiro, numa ótica mais geral sobre testagem de web APIs, são apresentadas ferramentas semelhantes à *RapiTest*, provenientes de artigos científicos. São também mencionadas duas conhecidas ferramentas industriais. A segunda secção, mais centrada no objetivo concreto deste trabalho, tem como foco o editor gráfico para os testes. É feito um breve resumo da abordagem escolhida para o editor e das suas vantagens, fundamentado a escolha e apresentando alguns artigos científicos sobre o tema. São também apresentados alguns exemplos da utilização deste tipo de interfaces atualmente em ferramentas amplamente usadas em diferentes indústrias.

2.1 Testagem de Web APIs

Tendo em conta o vasto uso de web APIs atualmente, como mencionado na introdução, não é surpresa existirem diversas ferramentas e artigos que procurem oferecer algum tipo de solução no que toca à testagem de APIs. Acerca da literatura sobre este tópico em particular, foram analisados vários artigos que propõem ferramentas que apresentam algumas semelhanças com a aplicação *RapiTest*. Três das ferramentas que mais se destacam são a *RestTestGen*, a *bBOXRT* e a *RESTest*:

RestTestGen [4], tal como *RapiTest*, segue uma abordagem *black-box* em que são gerados testes automaticamente a partir da especificação OAS. O aspeto mais interessante, porém, é abordagem usada para a construção dos testes, tentando estabelecer dependências entre as várias operações, de forma a ter estas em conta para aumentar a robustez dos testes criados.

bBOXRT [5] é também uma ferramenta baseada em *black-box* e também esta gera testes a partir da especificação OAS. Especializada em testes do tipo *Fuzzing*, é capaz de realizar um grande numero de mutações sobre os valores de teste, sendo que as respostas aos pedidos são tidas em conta para os testes futuros.

RESTest [6], à semelhança das duas anteriores, segue também a abordagem *black-box* e tira partido da especificação OAS para a geração de testes. A particularidade desta ferramenta

prende-se com a atenção a pedidos em que um parâmetro de input condiciona outro parâmetro de input. Os autores referem-se a isto como *inter-parameter dependencies*. Ao fazer a análise destas dependências, é possível aumentar a robustez e sucesso dos testes realizados. Por exemplo, testes de *fuzzing*, muitas vezes gerados com valores aleatórios, normalmente não respeitariam estas dependências, causando um número elevado de falhas.

Existem também ferramentas mais industriais, amplamente utilizadas hoje em dia. Duas das mais populares são o *Postman* [7] e a *SoapUI* [8]. Estas duas ferramentas oferecem inúmeras funcionalidades no que toca à testagem de APIs, incluindo de padrões que não o REST: geração de testes automaticamente, criação de testes manualmente, criação de coleções de testes, resultados detalhados acerca da execução dos testes, integração com outras ferramentas, entre outros. No entanto, apresentam também algumas lacunas. O *Postman* apresenta menos capacidade no que toca ao tópico da automação de testes, e possui menos verificações nativamente, tais como verificações relativas a testes de carga e a segurança. A *SoapUI*, consequência de ser uma ferramenta mais completa e robusta, apresenta uma curva de dificuldade acentuada, com uma interface de utilizador inferior. Para além disso, requer mais recursos computacionais, prejudicando a sua performance.

2.2 Editor gráfico para os testes

Relativamente à implementação do editor gráfico para a criação e edição de testes, objetivo principal para a melhoria da ferramenta *RapiTest*, foram também recolhidas informações acerca deste tópico. Foi decidido que a implementação do editor seria feita seguindo uma interface *node-based*. Este tipo de abordagem, também conhecida como *flow-based*, *dataflow*, entre outros, é essencialmente um tipo de interface de utilizador em que o utilizador interage com o sistema através da inserção, remoção e manipulação de nós e fios (*nodes* e *flows*). Isto acontece por exemplo através de sistemas *drag-and-drop*, em que o utilizador arrasta os elementos da interface com o cursor do rato. Estes nós são depois interligados entre si, através de *flows* (normalmente representados por setas, ou linhas), produzindo assim estruturas que se assemelham a *workflows*. Este tipo de interfaces traz grandes vantagens quando comparadas com uma abordagem mais clássica ou textual, nomeadamente a sua simplicidade, flexibilidade, interatividade e modularidade. Os utilizadores conseguem observar em tempo real as relações e fluxos entre os nós, e através de um simples arraste ou seleção conseguem produzir alterações aos *workflows*, tornando todo o processo de criação e modificação de *workflows* fácil de entender. Por exemplo, alterações complexas que poderiam de uma outra forma necessitar de alterações manuais em código, podem agora ser feitas através da interface de forma simples. Isto permite que qualquer tipo de utilizador, mesmo utilizadores sem conhecimento técnico, consigam utilizar o sistema. Este aspeto vai diretamente de encontro ao pretendido com o presente trabalho, afirmando assim a escolha deste tipo de interfaces.

Este tipo de interfaces é alvo de estudo no campo da programação há várias décadas. Eric Hosick, arquiteto de software e fundador do blogue *InterfaceVision*, mantém um repositório de *snapshots* de interfaces deste tipo [9], relacionadas com programação visual, em que o primeiro registo de uma interface deste tipo surge em 1963. Ainda nos anos 60, foram

publicados dois textos sobre o tópico. Em 1966, William Robert Sutherland analisou na sua tese “Online Graphical Specification of Procedures” [10] questões relacionadas com fluxo de dados, afirmando que o fluxo de dados de um programa pode ser suficiente para determinar as suas operações, não sendo necessário controlar o comportamento através de instruções explícitas de controlo de fluxo, o que de facto é uma das vantagens das interfaces *node-based*. Em 1969, foi publicado o texto “The GRAIL Project: An experiment in man-machine communications” [11], onde a linguagem apresentada GRAIL (*Graphical Input Language*) era uma linguagem gráfica baseada em diagramas de fluxo. Os autores afirmavam que os diagramas de fluxo ajudam na compreensão das opções de controlo e nas relações entre os diferentes processos, devido ao facto de ser possível observar estes aspetos em duas dimensões. Desde então o tópico das interfaces visuais, especialmente relacionadas com a programação visual, tem sido alvo de debate, sendo que este tipo de interfaces vai sendo cada vez mais uma opção escolhida para sistemas que possam tirar proveito das mesmas.

As interfaces *node-based* são assim hoje em dia amplamente usadas em aplicações e sistemas que envolvam o estabelecimento de *workflows* ou o processamento sequencial de dados, entre outros. Alguns exemplos conhecidos que utilizam este tipo de interface são a *Unreal Engine* [12], um motor de jogo, utilizado com grande sucesso em alguns dos jogos mais jogados no mundo. Esta oferece como uma das suas funcionalidades uma linguagem *node-based* para o desenvolvimento de jogos. “The Blueprint Visual Scripting system in Unreal Engine is a complete gameplay scripting system based on the concept of using a node-based interface to create gameplay elements” [13]. Outra ferramenta que faz uso deste tipo de interfaces é o *Blender* [14], um software para a criação de animações, modelagem, entre outros. “Blender contains different node-based editors with different purposes” [15]. Existem muitas mais ferramentas que usam este tipo de interface, como *Houdini* [16], *DaVinciResolve* [17], entre outras.

Para além de software de “utilidade geral”, as interfaces *node-based* têm também uma aplicação específica no campo da programação. Nomeadamente, em tópicos como a programação visual e as abordagens *low-code*, cada vez mais populares nos dias que correm. Empresas como a *OutSystems* [18] seguem esta abordagem nos seus serviços, oferecendo plataformas *low-code*, com interfaces de utilizador deste tipo. *Node-RED* [19] é uma ferramenta *low-code* para aplicações baseadas em eventos, normalmente usada no campo da IoT (*Internet of Things*), que oferece também uma interface à base de nós e fluxos. “Node-RED provides a browser-based flow editor that makes it easy to wire together flows using the wide range of nodes in the palette” [20]. O próprio *Postman*, já mencionado anteriormente, possui uma interface visual para a criação de *workflows* de APIs (porém, esta funcionalidade não tem como objetivo fornecer uma solução para a testagem de APIs, sendo mais direcionada para construção de aplicações simples). Denominada Postman Flows [21], é também uma interface *node-based* (sendo que a terminologia utilizada pelo Postman para os nós é *Block*, ou *Blocks*).

2.3 Resumo

Existem muitas mais ferramentas e artigos do que os mencionados. Porém, considerando todos estes aspetos e toda a informação recolhida, é possível observar que existem várias ferramentas com um objetivo semelhante à *RapiTest*, contudo todas com as suas diferenças, especialidades e também lacunas. É possível reafirmar também os méritos da escolha de uma interface *node-based* para o objetivo de providenciar um editor simples mas eficaz para a criação de testes, ajudando assim a solidificar a *RapiTest* como uma ferramenta acessível a qualquer tipo de utilizador, independentemente do seu nível de conhecimento técnico.

3 Abordagem

Este capítulo trata de apresentar alguns aspetos acerca da abordagem tomada no que toca ao desenvolvimento da *RapiTest*. Primeiramente, são tidas algumas considerações relativamente aos diferentes tipos de teste, ao papel da especificação OAS e à necessidade de encontrar uma solução que permita ir de encontro aos objetivos pretendidos. De seguida, é introduzida a linguagem TSL, e é feita uma explicação sobre as suas características e capacidades. Posteriormente, é detalhada a abordagem tomada para o planeamento do editor de testes, incluindo os objetivos que este se propõe a resolver, os problemas do editor antigo, as capacidades pretendidas, entre outros. Finalmente, é apresentada a arquitetura e os diferentes componentes da aplicação *RapiTest*.

3.1 Testagem de Web APIs

3.1.1 Tipos de teste

Considerando os objetivos já identificados, e tendo em conta o tópico em questão, um passo necessário a tomar na abordagem ao problema é a identificação dos diferentes tipos de teste e cenários de teste a suportar.

No caso concreto das APIs, existem inúmeros tipos de teste possíveis, desde testes de validação de dados e esquemas, tratamento de erros, testes de carga/*stress*, compatibilidade, segurança, fluxo, entre outros. Muitos destes dividem-se ainda em sub-categorias, pelo que as possibilidades de testagem são imensas.

No que diz respeito a satisfazer os objetivos da aplicação *RapiTest*, foram originalmente identificados três principais tipos de teste [3]:

- Testes de Validação
- Testes de Carga
- Testes de Fluxo

Os testes de validação englobam todos os testes cujo objetivo seja validar que a resposta a um pedido está conforme o esperado. Isto inclui por exemplo a verificação de *status code* ou a análise do corpo da resposta (seja para validar o esquema ou o conteúdo concreto).

Os testes de carga têm como objetivo testar a capacidade da API de responder aos pedidos em situações em que está sob *stress* devido a um número extremamente alto de pedidos simultâneos. Isto por sua vez permite perceber a capacidade limite da API, e aferir a necessidade de alocar mais recursos computacionais para garantir um desempenho normal em situações de tráfego elevado.

Os testes de fluxo são aqueles em que se pretende testar uma sequencia de ações com uma ordem definida, simulando por exemplo um fluxo típico que um utilizador possa realizar. Muitas vezes, estes testes acabam por ser compostos por uma sequência de testes de validação cuja ordem de execução é determinante para obter um resultado que faça sentido.

A forma como estes tipos de teste são suportados na prática será descrita na secção 3.2 relativa à TSL.

3.1.2 Geração automática *vs* elaboração manual

Uma forma conveniente de produzir testes é através da sua geração automática tendo como base a especificação da API. A especificação de uma RESTful API é normalmente descrita usando o padrão OAS - *OpenAPI Specification*. A OpenAPI [22] é uma iniciativa que tem como principal objetivo atingir a estandardização das descrições de RESTful APIs. É uma iniciativa pertencente à Linux Foundation, contando com apoio de gigantes como a Google, Microsoft, entre outros. A especificação propriamente dita foi evoluindo ao longo dos anos, sendo baseada na antiga especificação Swagger, que foi doada a este projeto.

Este é, portanto, um padrão amplamente utilizado na indústria, estabelecendo um formato definido para a descrição da API, nomeadamente os *endpoints* disponibilizados, métodos suportados, respostas esperadas, parâmetros, entre outras.

Uma especificação é, no fundo, uma descrição das características, capacidades e comportamento da API em questão. No caso da OAS, as especificações são normalmente descritas em YAML ou JSON, sendo facilmente compreendidas tanto por pessoas como por computadores. Uma especificação é também ponto de partida para permitir a elaboração de outras ferramentas de forma automática, nomeadamente geração de documentação, geração de aplicações cliente e servidor para a API e, como já mencionado, a geração de testes.

Apesar da sua utilidade e facilidade relativamente à elaboração manual, a geração automática de testes a partir da especificação não é de todo suficiente para cobrir todos os tipos de teste desejados.

Em primeiro lugar, existe o risco de a especificação conter falhas, erros e outros problemas, seja por falta de manutenção, falhas na geração, ou por qualquer outra razão, comprometendo imediatamente os testes gerados. Frequentemente, as especificações não consideram todos os cenários possíveis que podem ocorrer a partir de um pedido, por exemplo casos em que aconteça determinado tipo de erro inesperado ou uma situação incomum. Isto por sua vez

traduz-se em testes que não contemplam estes cenários.

É também bastante mais difícil, ou até impossível, gerar automaticamente cenários de teste personalizados complexos. Por exemplo, testes de fluxo, em que é necessário estabelecer uma relação de ordem entre pedidos a diferentes *endpoints*; testes de validação mais complexos, nos quais se pretende fazer uma validação concreta do conteúdo da resposta; testes de carga e stress de forma a averiguar a performance e escalabilidade da API sob condições de elevado tráfego; e até mesmo relativamente à segurança, sendo que existem vulnerabilidades que não são possíveis de testar apenas com base na especificação.

Por todas estas razões, foi desenvolvida no trabalho anterior a linguagem TSL, com vista a suportar a realização de testes mais específicos e personalizados. Esta linguagem será apresentada de seguida.

3.2 TSL - Test Specification Language

A Test Specification Language, TSL, foi então desenvolvida com o propósito de permitir a elaboração manual e personalizada de testes, tendo em conta os principais tipos de teste identificados anteriormente.

A Figura 3.1 apresenta o esquema para a estrutura de uma configuração de teste na TSL:

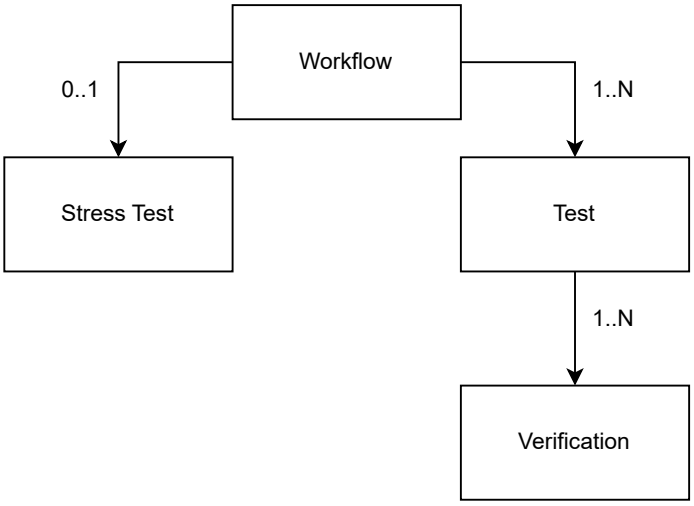


Figura 3.1: Esquema da lógica de testagem suportada pela TSL

Esta nova linguagem suporta a especificação de *workflows*, de forma a testar uma sequência especifica de testes em determinada ordem (por outras palavras, testes de fluxo mencionados anteriormente); testes de carga (*stress test*), sendo possível especificar o número de execuções, o número de *threads* e o *delay* entre cada pedido; e testes personalizados, oferecendo várias validações nativas. O utilizador inicialmente especifica os dados necessários para a execução do teste, nomeadamente o servidor a que o teste deve aceder, que método HTTP a utilizar, corpo do pedido, entre outros. Ainda na definição do teste, o utilizador especifica de seguida as verificações que pretende que sejam realizadas. De momento, existem 5 verificações suportadas nativamente pela TSL, que serão descritas em detalhe nesta secção.

Caso nenhuma das verificações satisfaça o pretendido pelo utilizador, existe também a possibilidade de fornecer uma verificação personalizada, através do fornecimento de um ficheiro `.dll` com a implementação da verificação.

3.2.1 Ficheiro TSL

Um ficheiro TSL é escrito utilizando a linguagem YAML [23], sendo assim possível a sua fácil interpretação por pessoas e computadores, à semelhança do que acontece com as especificações OAS. É composto da seguinte forma:

```
- WorkflowID: workflow_1
  Stress:
    Count: 100
    Threads: 10
    Delay: 0
  Tests:
  - TestID: test_1
    Server: "https://example.api.com"
    Path: "/examplePath"
    Method: Post
    Query:
        - name=exampleName
    Headers:
      - Content-Type: application/json
    Body: "$ref/dictionary/exampleBody"
    Retain:
      - retainedId#$.responseId
    Verifications:
      - Code: 200
        Contains: exampleString
        Count: exampleString#1
        Schema: "$ref/definitions/exampleSchema"
        Match: $.examplePath#exampleString
        Custom: ["CustomVerification.dll"]
  - TestID: test_2
#...
- WorkflowID: workflow_2
#...
```

Listagem 3.1: Exemplo da estrutura de um ficheiro TSL

A Listagem 3.1 apresenta um ficheiro TSL com todas as funcionalidades oferecidas pela linguagem. Serve apenas como exemplo, sendo que os dados e alguma combinação de atributos poderão não fazer sentido num caso prático. Tendo em conta o formato do ficheiro e as capacidades do TSL, é possível afirmar que para um utilizador com algum nível de conhecimento técnico, produzir um ficheiro TSL com as verificações desejadas não será uma tarefa muito difícil. No entanto, o mesmo não se verifica no caso de um utilizador que não possua tanto conhecimento técnico, que terá dificuldades a criar um ficheiro TSL manualmente. Esta é uma das principais razões que levou à decisão da implementação de um editor gráfico

para a criação dos testes, objetivo já mencionado previamente e que será descrito em maior detalhe posteriormente neste documento.

De seguida serão explicados em mais detalhe todas as funcionalidades suportadas pela TSL.

Workflow:

```
- WorkflowID: my sample workflow
  Stress:
    ...
  Tests:
    ...
```

Listagem 3.2: Estrutura de um *Workflow*

Um *workflow* é composto por um ou mais testes e, opcionalmente, por um *stress test*. Em termos de informação específica do *workflow*, apenas importa fornecer um nome (único) para o mesmo. Os testes serão executados em sequência, na ordem pela qual são definidos no ficheiro.

Stress Test:

```
Stress:
  Count: 10
  Threads: 2
  Delay: 50
```

Listagem 3.3: Estrutura de um *Stress Test*

Um *stress test* tem três campos: `Count` representa o número total de execuções pretendidas para o *workflow*; `Threads`, como o nome indica, o número de *threads* que serão usadas para a execução; e `Delay`, que representa o tempo de espera entre cada execução de um *workflow*, em milissegundos.

No exemplo acima, o *workflow* será executado 10 vezes. Estas 10 execuções serão divididas por 2 *threads* (pelo que cada *thread* fará 5 execuções). Entre cada execução haverá um intervalo de 50 milissegundos.

Test:

```
1  Tests:
2  - TestID: createPet
3    Server: "https://petstore3.swagger.io/api/v3"
4    Path: "/pet"
5    Method: Post
6    Query:
7      - status=available
8    Headers:
9      - Content-Type: application/json
10   Body: "$ref/dictionary/petBody"
11   Retain:
12     - petId#$.id
13   Verifications:
14     ...
```

Listagem 3.4: Estrutura de um *Test*

Um teste, à semelhança de um *workflow*, contém um nome identificador que deve ser único. Ao contrário do *workflow*, possui uma série de campos onde é possível fornecer a informação relevante para a execução do teste, alguns destes opcionais.

É sempre necessário fornecer pelo menos 3 campos: `Server`, `Path` e `Method`. Estes correspondem respetivamente ao endereço do servidor (ou seja, a "base" da API), ao caminho concreto para o teste (que é concatenado ao `Server` para formar o *endpoint* completo) e ao método HTTP usado no teste.

Há também a possibilidade de fornecer informação adicional sobre o pedido, através de 4 campos opcionais: `Query`, `Headers`, `Body`, `Retain`. Os três primeiros são fáceis de compreender, correspondendo essencialmente às funcionalidades indicadas pelo próprio nome: fornecer parâmetros de *query*, *headers* a ser utilizados no pedido e conteúdo a ser incluído no corpo do pedido. Já o campo `Retain` não corresponde a nenhuma funcionalidade comum nos pedidos HTTP; este campo permite especificar um valor para "reter" de forma a poder ser utilizado nos testes seguintes do mesmo *workflow*. Esta funcionalidade é bastante útil no contexto de um *workflow*, pois permite que um utilizador possa utilizar o mesmo valor em todo o *workflow* sem o saber *a priori*.

O exemplo pretende criar um *pet*, utilizando a API Swagger Petstore. O campo `Body` inclui uma referência a um ficheiro auxiliar, onde o conteúdo usado no pedido está definido com a chave `petBody` (linha 10). Os ficheiros auxiliares serão descritos em mais detalhe posteriormente.

O exemplo inclui o campo `Retain`, onde é especificado que o valor `id` presente na resposta ao pedido será guardado na variável `petId` (linhas 11 e 12). Assim, o utilizador poderá usar esta variável nos testes subsequentes, por exemplo usando o *path* `pet/petId`, permitindo assim operar sobre o recurso criado no teste anterior.

Verifications:

```
Verifications:
  - Code: 200
    Contains: id
    Count: doggie#1
    Schema: "$ref/definitions/Pet"
    Match: $.name#doggie
    Custom: ["CustomVerificationTry1.dll"]
```

Listagem 3.5: Estrutura de *Verifications*

Cada teste contém verificações, sendo que a primeira, `Code`, é a única obrigatória (e representa o *status code* HTTP). `Contains` verifica se o corpo da resposta contém a *string* indicada; `Count` é semelhante, mas verifica também quantas vezes está presente a *string*; `Schema` verifica se o corpo da resposta satisfaz o esquema fornecido; `Match` verifica se o corpo da resposta inclui no *path* indicado a *string* indicada; `Custom` permite fornecer uma qualquer verificação implementada pelo utilizador através de um ficheiro `.dll`.

3.2.2 Ficheiros Auxiliares

A TSL permite que o utilizador tire proveito de 2 tipos diferentes de ficheiros auxiliares: ficheiros dicionário e ficheiros de verificação personalizados.

O ficheiro dicionário tem como principal função facilitar o preenchimento do ficheiro TSL ao permitir definir separadamente dados como objetos a usar no corpo de um pedido ou esquemas para usar nas validações.

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

Listagem 3.6: Exemplo de um ficheiro dicionário

Os ficheiros `.dll` permitem que o utilizador forneça uma verificação personalizada implementada em código, caso este pretenda realizar alguma verificação que não seja suportada nativamente pela TSL.

É apenas necessário implementar um método que respeite a interface `Verification`:

```
1 public interface Verification
2 {
3     Task<Result> Verify(HttpResponseMessage Response);
4 }
```

Listagem 3.7: Interface Verification

`Result` está definido da seguinte forma:

```
1 public class Result
2 {
3     public string TestName { get; set; }
4
5     public bool Success { get; set; }
6
7     public string Description { get; set; }
8 }
```

Listagem 3.8: Classe Result

3.3 Editor de testes

De forma a poder tirar partido ao máximo da linguagem TSL, é necessário que a aplicação *RapiTest* esteja preparada para suportar a mesma, nomeadamente oferecendo um editor de testes que permita aos utilizadores criar e editar configurações de teste que usem TSL.

3.3.1 Requisitos identificados

Como mandam as boas práticas do desenvolvimento em software, após estabelecido o objetivo foi então necessário proceder à identificação dos requisitos que o editor deve suportar.

Relativamente à criação de configurações de teste, estes foram os requisitos identificados:

- Criar uma nova configuração de teste, fornecendo um nome para a mesma;
- Fornecer a especificação OAS, ao fazer upload de um ficheiro ou fornecendo um URL;
- Configurar definições de temporização para a execução dos testes;
- Editar configurações de teste previamente realizadas.

Relativamente aos ficheiros TSL, estes foram os requisitos identificados:

- Permitir a criação de um ficheiro TSL de raíz a partir do editor através da interface gráfica *node-based*;
- Permitir o carregamento e edição de um ficheiro TSL previamente elaborado através da interface gráfica *node-based*;
- Fazer carregamento de ficheiros auxiliares (dicionário e de verificação personalizada) a serem usados no ficheiro TSL (e na interface gráfica).

Mais concretamente sobre a criação/edição do ficheiro TSL através da inteface gráfica, estes foram os requisitos identificados:

- Criar, editar e apagar um *workflow*;
- Criar, editar e apagar um *test*;
- Criar, editar e apagar um *stress test*;
- Criar, editar e apagar todos os componentes opcionais de teste oferecidos pela TSL: *body*, *headers*, *query*, *retain* (mencionados na secção anterior relativa à TSL);
- Criar, editar e apagar todas as verificações oferecidas pela TSL: *status*, *schema*, *contains*, *count*, *match*, *custom* (mencionadas na secção anterior relativa à TSL).

Finalmente, foram tidos em conta outros requisitos diversos mais relacionados com a interface e experiência de utilizador no geral:

- Permitir fácil alteração da ordem dos testes e *workflows* após a sua criação;
- Facilitar preenchimento dos campos com base na informação da especificação;
- Alertar imediatamente para erros de preenchimento e destacar campos obrigatórios;
- Mecanismos de prevenção de erros;
- Disponibilizar ajuda/informação adicional sob a forma de *tooltips* com dicas de utilização.

3.3.2 *Layout* do editor

Com base em todos os requisitos identificados e também tendo em consideração toda a investigação feita sobre este tipo de editores (como mencionado na secção 2), foram identificadas três áreas principais para o editor:

- Área de configuração
- Área de trabalho
- Área de controlos e definições

3.3.2.1 Área de configuração

Esta área tem essencialmente dois propósitos: o primeiro é permitir ao utilizador inserir a informação necessária para a configuração de teste que este vai realizar, nomeadamente o nome escolhido, a especificação OAS e configurações de temporização. Haverá também a possibilidade de fornecer os ficheiros auxiliares previamente mencionados. O segundo objetivo é disponibilizar uma interface que permita a inserção dos nós na área de trabalho, por exemplo através da presença de botões que representam os nós - para que o utilizador possa criar um novo *workflow*, um teste, inserir uma verificação, etc.

3.3.2.2 Área de trabalho

A área de trabalho é como o nome indica a área em que o utilizador irá construir os seus testes e *workflows*. Esta área deverá ser um espaço amplo em que o utilizar consiga inserir nós e operar sobre estes, seja arrastá-los, compactá-los, ligá-los a outros nós, preencher os seus dados, entre outros.

3.3.2.3 Área de controlos e definições

Esta área deve ser de pequenas dimensões, e deve conter controlos para o editor tais como botões para alterar nível de *zoom*, compactar ou expandir nós, organizar os nós de forma visualmente apelativa, e outras utilidades relativas ao editor. Deverá também permitir abrir uma janela para configurar quaisquer definições relativas ao editor que se entendam úteis.

Na Figura 3.2 é possível observar um *mockup* do *layout* pretendido para o editor:

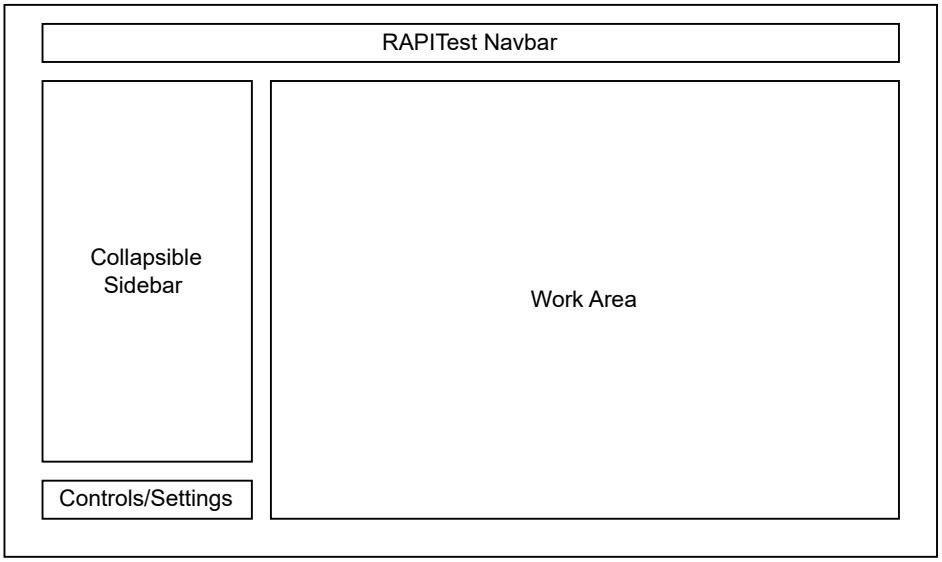


Figura 3.2: *Mockup* do *layout* pretendido para o editor

Neste *mockup*, a Collapsible Sidebar (barra lateral colapsável) representa a área de configuração, Work Area a área de trabalho e Controls/Settings a área de controlos e definições do editor.

Com o intuito de melhorar ao máximo a experiência de utilizador, uma consideração importante que foi tida em conta foi a preocupação de aproveitar ao máximo o espaço útil no ecrã do editor, de forma a proporcionar uma área de trabalho com o maior tamanho possível. Esta é também a razão para a ênfase na barra colapsável, pois permite ao utilizador "esconder" elementos da página que não são necessários de momento de forma a expandir a área de trabalho para uma melhor organização e experiência de utilização.

3.4 Arquitetura

A arquitetura do sistema *RapiTest* pode ser observada na Figura 3.3. É essencialmente composta por quatro componentes distintos: a aplicação Web; o servidor; e dois serviços *Worker*, um para o *setup* dos testes e o outro para a execução dos mesmos.

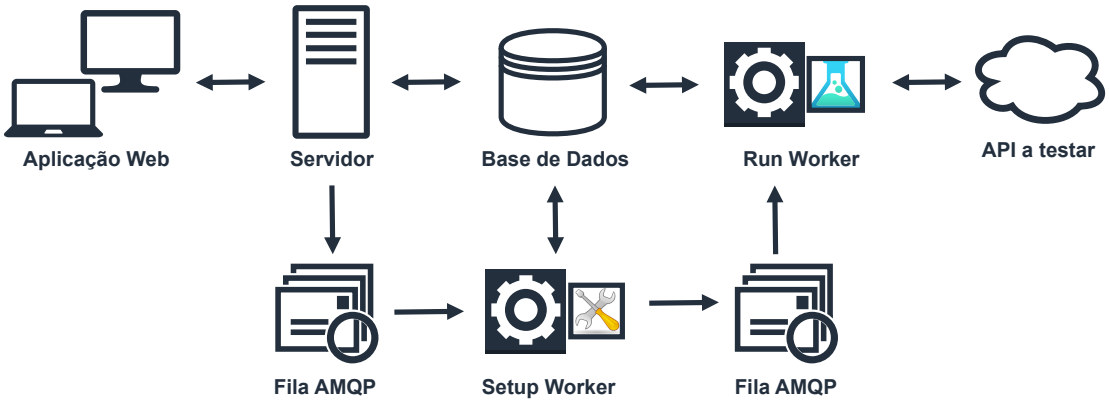


Figura 3.3: Diagrama da arquitetura do sistema *RapiTest*

Esta é composta por uma aplicação Web, à qual naturalmente os utilizadores acederão para usar as funcionalidades do sistema. A aplicação Web fornece as interfaces necessárias para a configuração dos testes, *workflows*, e outras funções relacionadas. Esta comunica com o servidor através de uma Web API exposta pelo mesmo. O servidor, para além da óbvia função de atender aos pedidos da aplicação Web, é responsável pela comunicação com a base de dados, guardando os dados relevantes do utilizador, como por exemplo as configurações realizadas pelo mesmo. É também responsável por enviar uma mensagem para uma fila AMQP [24], sinalizando que os testes se encontram prontos a realizar. O primeiro serviço *Worker*, responsável pelo *setup* dos testes, atende à mensagem colocada na fila e realiza os procedimentos necessários para este *setup*, tais como validação e serialização dos ficheiros. Após isto, este serviço guarda na base de dados o ficheiro serializado e envia por sua vez uma mensagem para outra fila AMQP, sinalizando que o *setup* dos testes está concluído. O segundo serviço *Worker*, responsável por executar os testes, atende a esta mensagem e procede então à execução dos mesmos, comunicando com a base de dados primeiro para obter o ficheiro de testes serializado, e depois, após a execução dos testes concluída, para guardar os resultados na base de dados.

É importante mencionar que esta arquitetura foi desenvolvida previamente, e no decorrer deste trabalho não foram feitas alterações estruturais à mesma. A grande maioria dos desenvolvimentos incidiu sobre a aplicação Web, o que não é surpreendente tendo em conta que o principal objetivo do trabalho era a criação de um novo editor de testes, que, naturalmente, faz parte da aplicação Web. Porém, foram também feitas alterações à API do servidor, nomeadamente acrescentados *endpoints* para retorno de informação sobre configurações de teste. Estas alterações serão detalhadas na secção da implementação.

4 Implementação

Neste capítulo é detalhada a implementação dos diferentes componentes da aplicação *RapiTest*. A primeira secção é focada nos desenvolvimentos já feitos previamente, sendo que a implementação será descrita de forma mais geral uma vez que não foi realizada durante este trabalho. A segunda secção foca-se no editor de testes, e apresenta em maior detalhe a implementação do mesmo, nomeadamente como foi inserido na aplicação, as diferentes funcionalidades, bibliotecas utilizadas, exemplos de utilização, entre outros.

4.1 Desenvolvimentos prévios

Uma vez que a maior parte da aplicação *RapiTest* foi desenvolvida num trabalho prévio, as escolhas relativamente à implementação dos diferentes componentes do sistema já se encontravam definidas *a priori*. No entanto, apesar de não ser o âmbito deste trabalho, importa apresentar alguns destes aspetos.

Em primeiro lugar, relativamente à escolha de tecnologias usadas, houve a preocupação de priorizar tecnologias *cross-platform* e, se possível, de código aberto. Para o servidor, foi utilizada a *framework* .NET CORE, sendo a base de dados em SQL Server. A aplicação cliente foi desenvolvida como uma SPA (Single Page Application) em JavaScript utilizando a biblioteca React. Os serviços *Worker*, também implementados em .NET CORE, realizam a comunicação com o servidor e entre si usando o protocolo AMQP, através do *message broker* RabbitMQ.

Para a API do servidor, ponto de comunicação com a aplicação Web, foram disponibilizados os seguintes *endpoints*, divididos entre três controladores.

Para a configuração de testes:

- SetupTestController/GetSpecificationDetails
- SetupTestController/GetSpecificationDetailsURL
- SetupTestController/UploadFile
- SetupTestController/RemoveUnfinishedSetup

Para a gestão de testes:

- MonitorTestController/GetUserAPIs
- MonitorTestController/DownloadReport
- MonitorTestController/ReturnReport
- MonitorTestController/ReturnReportSpecific
- MonitorTestController/RunNow
- MonitorTestController/GenerateMissingTestsTSL
- MonitorTestController/ChangeApiTitle
- MonitorTestController/ChangeApi
- MonitorTestController/RemoveApi
- MonitorTestController/ReturnSpec
- MonitorTestController/ReturnTsl
- MonitorTestController/ReturnDictionary
- MonitorTestController/ReturnDll

Para detalhes do utilizador:

- HomeController/GetUserDetails

Os *endpoints* MonitorTestController/ReturnSpec, MonitorTestController/ReturnTsl, MonitorTestController/ReturnDictionary, MonitorTestController/ReturnDll e MonitorTestController/ChangeApi foram criados no decorrer deste projeto. Os primeiros quatro permitem obter do servidor informação sobre a especificação, TSL e ficheiros auxiliares associados a uma configuração de teste, informação necessária para algumas das funcionalidades desenvolvidas como a edição de uma configuração de teste existente. O último é precisamente para suportar a funcionalidade de edição, seguindo uma lógica semelhante ao *endpoint* para a criação de uma nova configuração de teste.

A aplicação possui uma barra de navegação no topo da página, com ligações para as diferentes páginas que esta oferece, nomeadamente *homepage*, configuração de testes, gestão de testes, registo, *login* e uma página de informações.

A aplicação inicia na *homepage*, que difere no seu conteúdo dependendo se o utilizador está ou não autenticado. Em caso afirmativo apresenta alguns dados específicos do utilizador. Um utilizador autenticado pode aceder às duas principais páginas da aplicação, a configuração de testes e a gestão de testes.

Relativamente à configuração de testes, a interface antiga divide o processo em 4 partes:

1. Escolha do nome
2. Upload da especificação OAS
3. Upload ou criação de TSL
4. Configuração das opções de temporização/execução

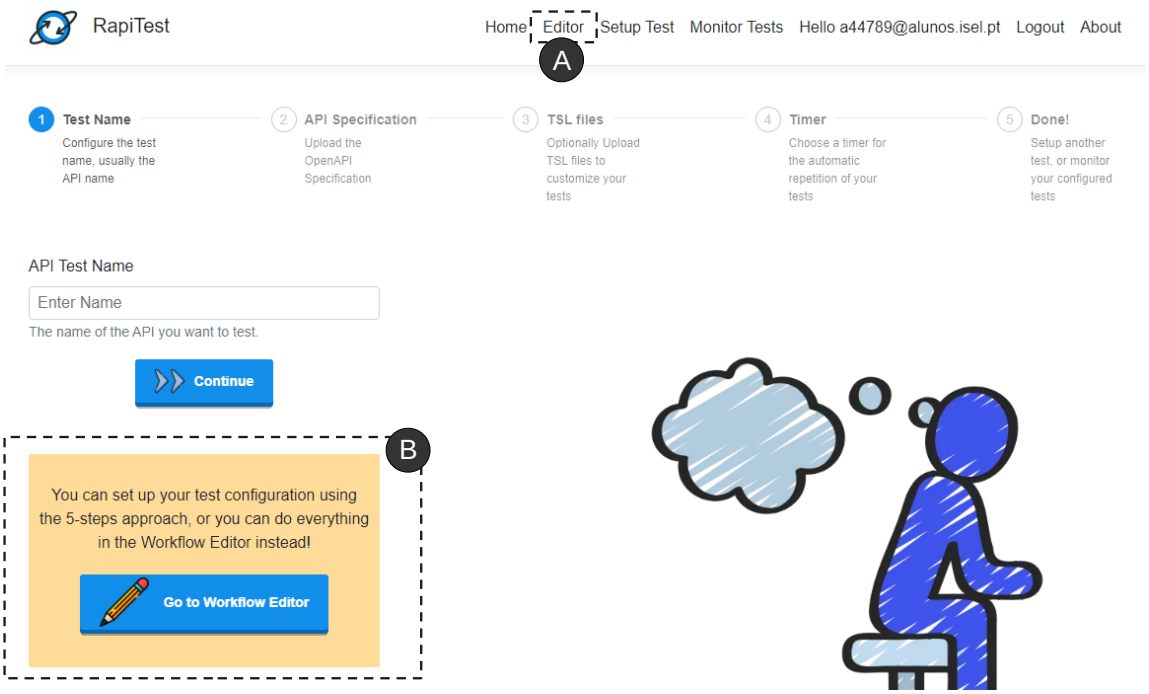


Figura 4.1: Interface antiga para o *setup* de uma configuração de teste.

A Figura 4.1 apresenta a vista do ecrã inicial de configuração de testes, com as alterações realizadas neste trabalho assinaladas nas secções A e B; a primeira foi a introdução de um *link* para o editor na barra de navegação (disponível em todas as vistas da aplicação); e a segunda o destaque dado ao editor na página de configuração, novamente com uma ligação direta para o editor.

Na página de gestão de testes, é apresentada uma lista de todas as configurações de teste realizadas pelo utilizador. Ao selecionar uma configuração, é possível realizar as seguintes ações:

- Editar o nome da configuração
- Executar os testes imediatamente
- Observar os resultados da última execução
- Descarregar num ficheiro os resultados da última execução
- Eliminar a configuração

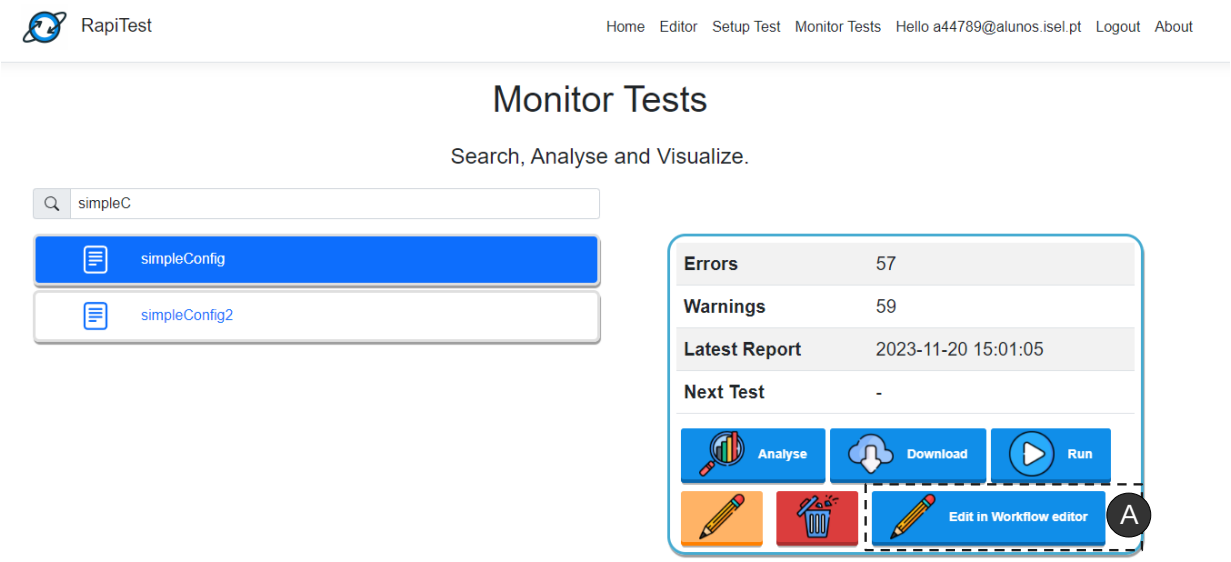


Figura 4.2: Vista das configurações de teste previamente realizadas.

A Figura 4.2 apresenta esta vista, já modificada para incluir o botão que permite editar a configuração no editor, delineado na secção A. Ao carregar neste botão, o utilizador é levado para o editor, que recria o estado com as informações da configuração (isto inclui, entre outras ações, a criação e preenchimento dos nós no editor).

A lógica de *setup* e execução dos testes é tratada pelos dois serviços *Workers*, da seguinte forma:

O *Setup Worker* ao receber mensagem do servidor, começa o processo de validação dos testes. Este processo envolve os seguintes passos:

- Análise da especificação OAS
- Análise do ficheiro TSL
- Análise do ficheiro dicionário
- Análise dos ficheiros DLL
- Deserialização para objetos de negócio
- Verificação de erros
- Geração automática de testes adicionais
- Serialização final dos testes

O *Run Worker* recebe mensagem do *Setup Worker* após este terminar o processo de validação dos testes, e inicia o processo de execução dos testes, que envolve os seguintes passos:

- Carregamento das bibliotecas externas
- Deserialização do ficheiro de testes
- Execução dos testes

4.2 Implementação do editor de testes

4.2.1 *Layout*

4.2.1.1 Vista Geral

A Figura 4.3 apresenta o estado inicial do editor. É possível observar que o *layout* final vai de encontro aos requisitos apresentados na secção 3, nomeadamente com uma interface que se assemelha bastante ao *mockup* idealizado na Figura 3.2.

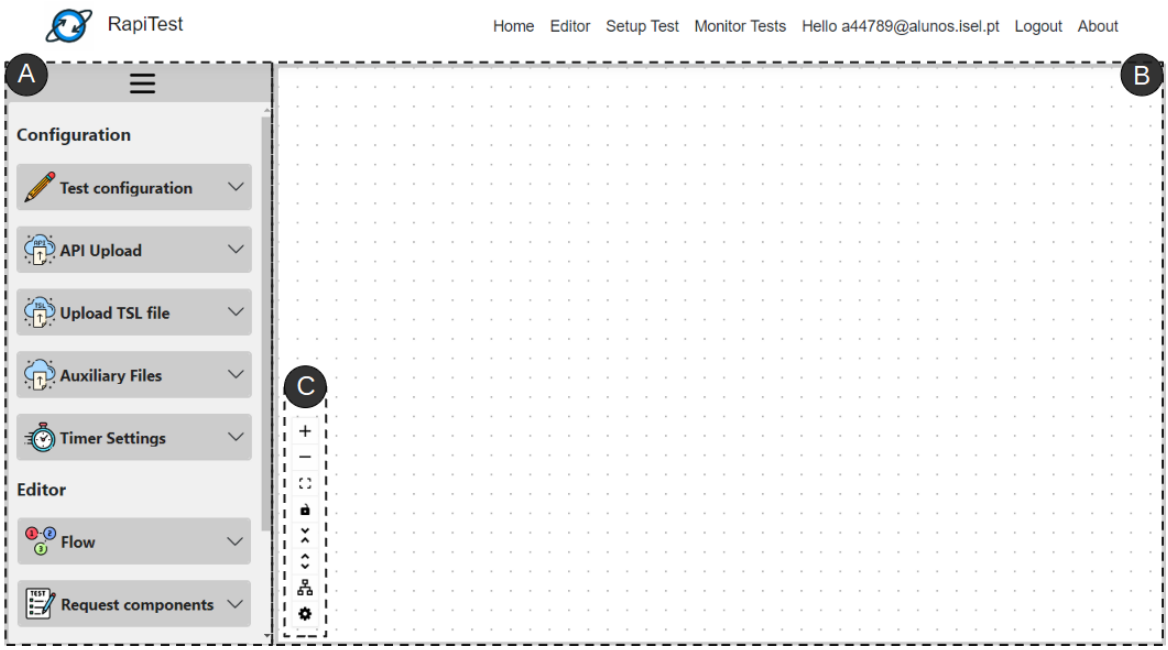


Figura 4.3: Vista geral do editor.

Do lado esquerdo (secção A) encontra-se a barra lateral, na qual o utilizador poderá fazer as configurações necessárias e escolher os nós a inserir na área de trabalho.

A área de trabalho surge imediatamente a seguir à barra lateral (secção B), ocupando todo o restante espaço do ecrã. Isto permite aproveitar ao máximo o espaço útil para providenciar uma melhor experiência de utilização. Na área de trabalho o utilizador irá então inserir, preencher e ligar os diferentes tipos de nós para construir os seus testes e *workflows*.

Finalmente, é possível também observar uma pequena área de controlos, a barra vertical no canto inferior esquerdo da área de trabalho (secção C). Esta está inserida no próprio editor tirando partido de um componente oferecido pela biblioteca React Flow.

Estas três áreas que em conjunto formam o editor na sua globalidade serão detalhadas de seguida.

4.2.1.2 Sidebar - Configurações

A primeira parte da barra lateral é dedicada à zona de configuração. Aqui, o utilizador poderá fazer o seguinte:

- Escolher o nome para a configuração de teste
- Fazer carregamento da especificação OAS
- Configurar as definições de temporização
- Fazer carregamento de um ficheiro TSL (opcional)
- Fazer carregamento de ficheiros auxiliares (opcional)

Uma vez que é necessário escolher um nome e fornecer a especificação OAS, estas serão as primeiras duas tarefas que o utilizador deverá realizar quando usa o editor (à semelhança do que acontecia na interface antiga). Na verdade, o próprio editor impede que sejam feitas outras ações (como inserir nós) antes da inserção do nome e da especificação. A Figura 4.4 mostra estes dois campos. Tal como requisitado anteriormente, é possível fazer o carregamento da especificação de duas formas: o utilizador pode fazer upload de um ficheiro (clicando ou arrastando o mesmo para a *dropzone*) ou pode inserir um URL que aponte para a especificação.

Após o carregamento estar feito, a vista muda e é possível ver o ficheiro inserido (Figura 4.5). É possível também apagá-lo de forma a poder fazer upload de outra especificação (por exemplo, em caso de engano do utilizador ao fornecer o ficheiro errado).

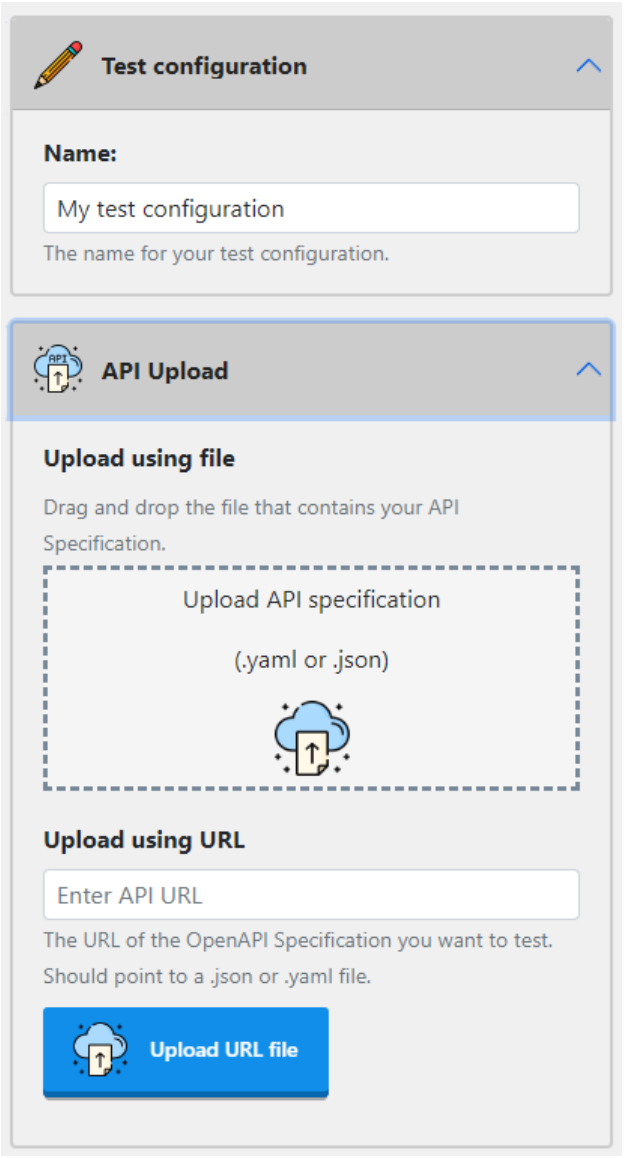


Figura 4.4: Configuração inicial: nome e carregamento da especificação *OAS*.

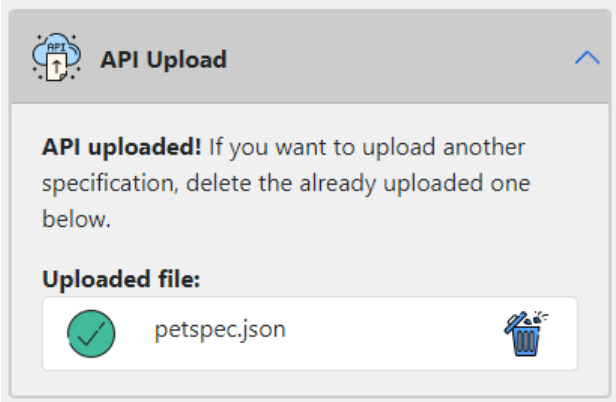


Figura 4.5: Especificação carregada pelo utilizador.

O editor fornece também a opção de fazer carregamento de um ficheiro TSL (Figura 4.6). Ao receber um ficheiro TSL válido, todo o estado do editor é automaticamente recriado - isto

significa, entre outras coisas, a inserção, preenchimento e ligação dos nós que correspondem ao conteúdo do ficheiro TSL. Estes poderão depois ser alterados pelo utilizador, que pode removê-los, alterar os seus dados ou acrescentar mais nós. Naturalmente, esta funcionalidade é de uma tremenda utilidade ao utilizador, permitindo efetivamente editar um ficheiro TSL previamente criado através da interface gráfica, um dos requisitos chave identificados para o editor.

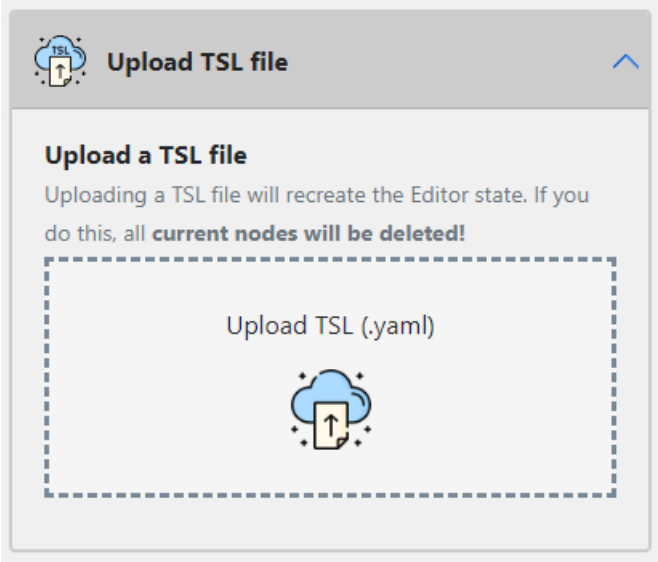


Figura 4.6: Possibilidade de carregamento de um ficheiro TSL.

Na Figura 4.7 é possível observar o editor logo a seguir ao upload de um ficheiro TSL. O editor apresenta *feedback* ao utilizador que o ficheiro foi carregado com sucesso (caso seja o caso) e de seguida preenche a área de trabalho de acordo com o mesmo, aplicando automaticamente o algoritmo de *layout* para organizar os nós (que será detalhado posteriormente neste documento). Verifica-se que o ficheiro descrito na Listagem 4.1 corresponde ao estado recriado na Figura. Como se observa, os nós e ligações possuem cores distintas, algo que também será detalhado posteriormente neste documento.

```
1  - WorkflowID: my simple workflow
2    Stress:
3      Count: 10
4      Threads: 2
5      Delay: 0
6    Tests:
7    - TestID: readPet
8      Server: "https://petstore3.swagger.io/api/v3"
9      Path: "/pet/10"
10     Method: Get
11     Headers:
12       - Accept:application/json
13     Verifications:
14       - Code: 200
15         Contains: id
```

Listagem 4.1: Exemplo de um ficheiro TSL simples.

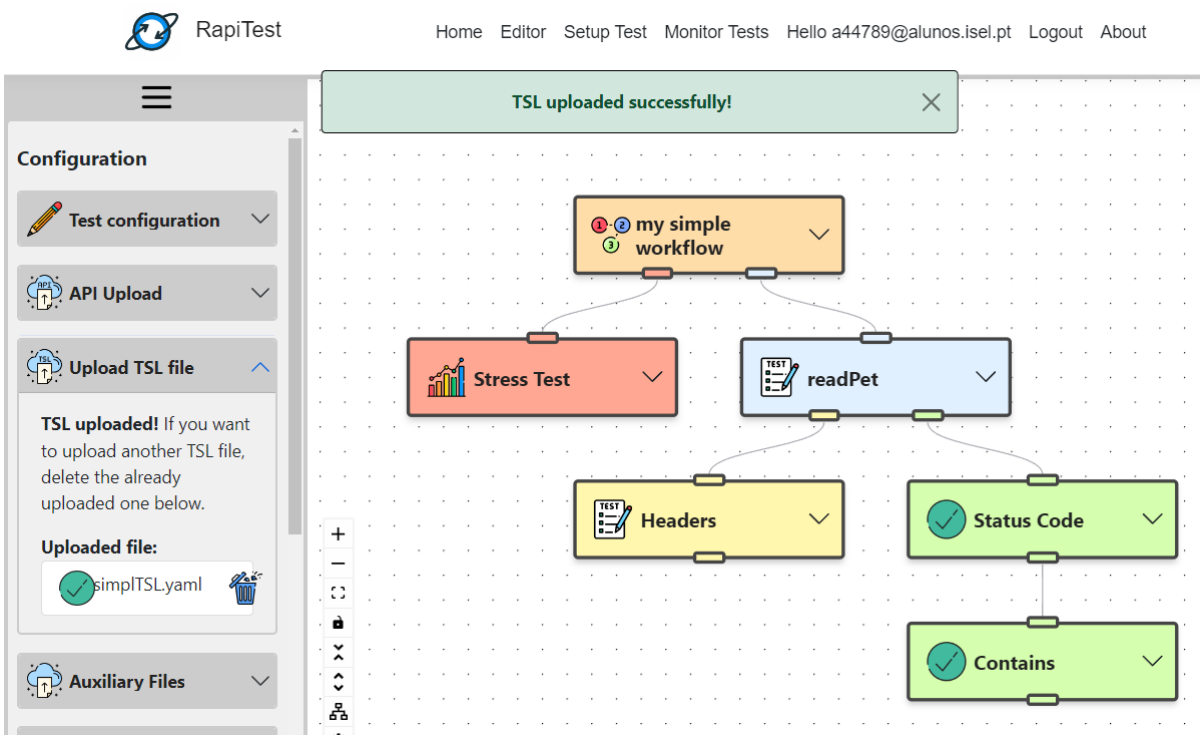


Figura 4.7: Estado do editor imediatamente após o upload do ficheiro TSL.

À semelhança do que acontece com a especificação e com os ficheiros TSL, é também possível fazer o upload de ficheiros auxiliares (Figura 4.8), nomeadamente ficheiros dicionário e ficheiros de verificação personalizada, cujos propósitos já foram mencionados anteriormente. É permito carregar um ficheiro dicionário (`.txt`) e múltiplos ficheiros de verificação personalizada (`.dll`). O dicionário pode ser usado nos nós `Body` e `Schema`, enquanto que as verificações personalizadas podem ser usadas no nó `Custom`. Ao fazer upload dos ficheiros, o conteúdo é processado de forma a ficar imediatamente disponível nos nós, mesmo que tenham sido inseridos na área de trabalho antes do carregamento dos ficheiros.

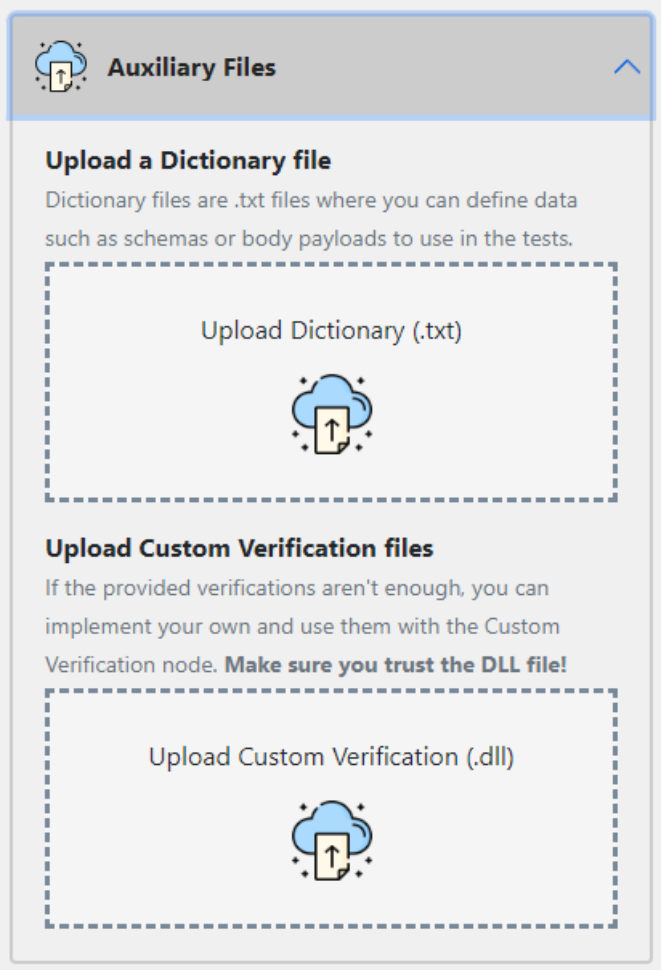


Figura 4.8: Possibilidade de carregamento de ficheiros auxiliares.

Finalmente, encerrando a zona de configuração (Figura 4.9), encontra-se a área dedicada às configurações de temporização. Existem três definições a ter em conta:

- Run Generated - se o utilizador pretende que sejam executados os testes gerados automaticamente a partir da especificação (em adição aos testes manualmente definidos pelo utilizador);
- Run Immediately - se os testes devem ser executados imediatamente após o utilizador terminar e guardar a configuração;
- Run Interval - qual o intervalo em que os testes devem ser realizados (sendo as opções disponíveis a cada hora, a cada 12 horas, a cada 24 horas, a cada semana ou sem execução contínua)

É obrigatório definir um valor para cada uma destas três definições. O editor seleciona previamente valores por omissão, pelo que o utilizador pode ou não mudar as definições.

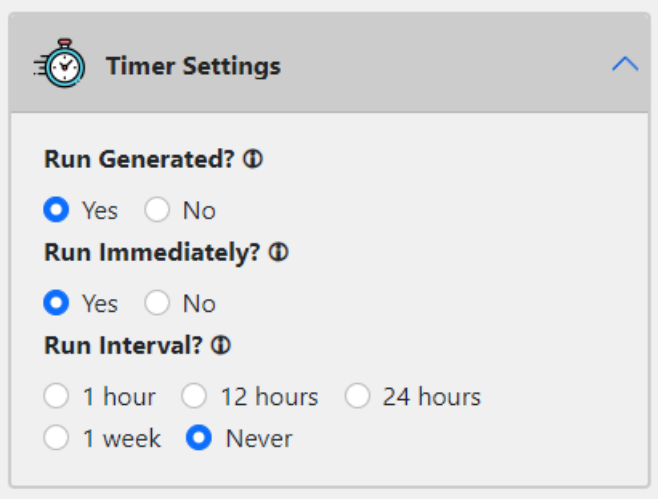


Figura 4.9: Configurações de temporização.

4.2.1.3 Sidebar - Editor

A segunda parte da barra lateral é dedicada ao editor em si, principalmente à inserção dos nós, estando estes divididos por três categorias: Flow, Request components e Verifications. Existe também uma área dedicada ao *setup*.

A primeira categoria, Flow, contém os três principais nós, nomeadamente Workflow, Test e Stress Test (Figura 4.10). Os dois primeiros em particular são essenciais para qualquer que seja a configuração de teste que o utilizador pretenda criar.

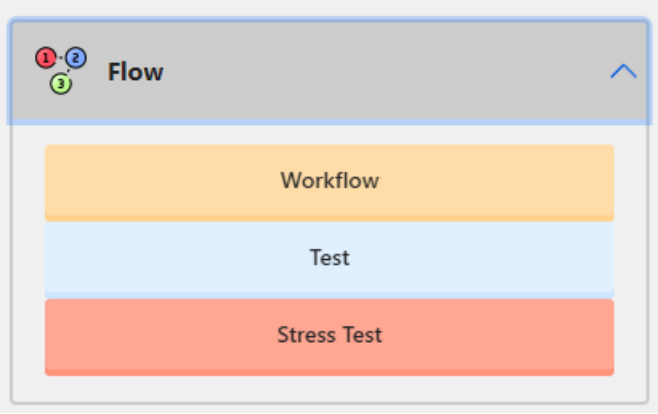


Figura 4.10: Nós do tipo Flow.

A categoria de Request components contém, como o nome indica, os nós que podem fazer parte do pedido executado num teste (Figura 4.11). São estes Body, Headers, Query e Retain, tal como observado anteriormente na secção relativa à linguagem TSL.

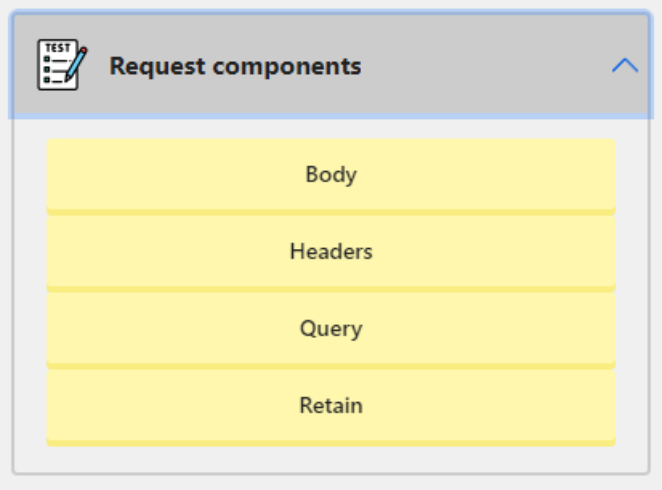


Figura 4.11: Nós do tipo Request components

A última categoria de nós é a das verificações, onde naturalmente se encontram os nós que representam as verificações suportadas pela TSL (Figura 4.12).

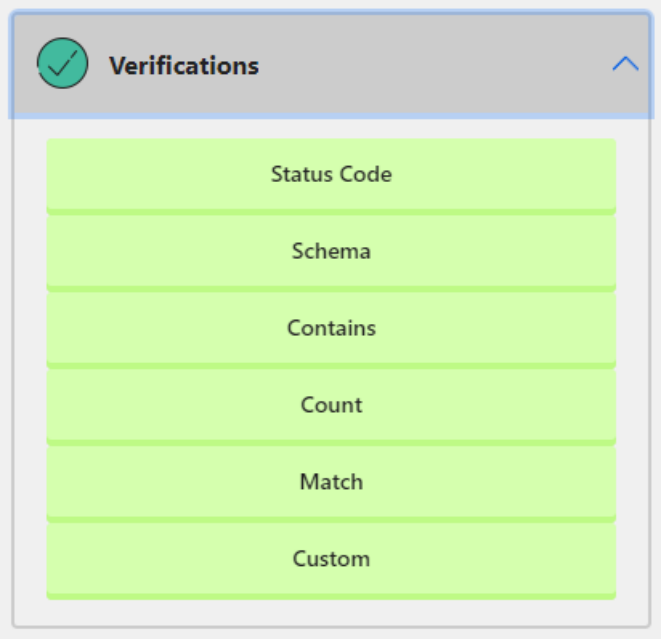


Figura 4.12: Nós do tipo Verifications

Finalmente, existe uma área relativa ao *setup*, onde o utilizador pode guardar e finalizar a configuração de teste (Figura 4.13).

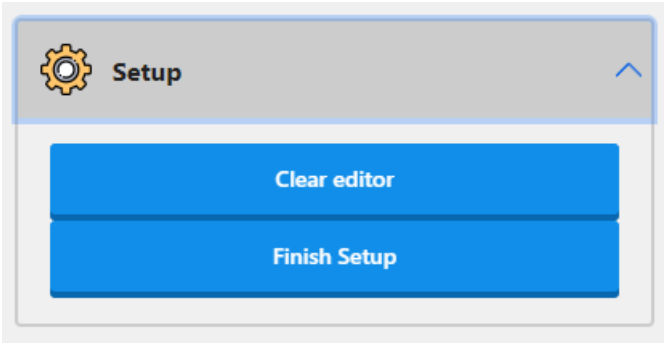


Figura 4.13: Área de *setup*.

4.2.1.4 Área de trabalho

A área de trabalho é, como o nome indica, onde o utilizador irá efetivamente construir os seus testes. Ao carregar nos botões da *sidebar*, o utilizador insere os respetivos nós na área de trabalho. Estes podem ser arrastados de forma a ficarem posicionados ao gosto do utilizador e também são colapsáveis/compactáveis, pelo que é possível esconder os detalhes de cada nó de forma a economizar o espaço disponível e promover uma melhor organização, como é observável na Figura 4.14.

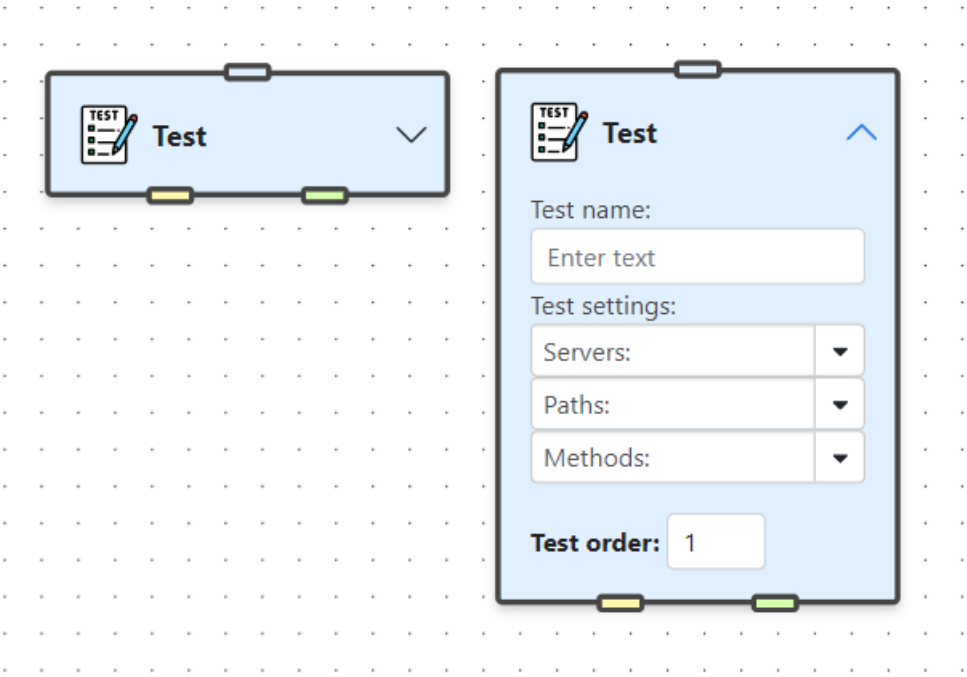


Figura 4.14: Exemplo de um nó compactado vs um nó expandido.

Como seria de esperar neste tipo de interfaces, é também possível conectar os nós a outros nós, pois só desta forma realmente se formam os testes e *workflows*. Existem, no entanto, regras para o fazer; nomeadamente, o editor certifica-se que as regras da TSL são respeitadas. Por exemplo, o nó Workflow pode apenas ligar-se com nós Test e Stress Test, e não por exemplo a um nó de verificação. Um nó de verificação por sua vez, apenas pode ligar-se a um Test ou a outras verificações. Essencialmente, têm que ser respeitadas as regras estruturantes da TSL, detalhadas anteriormente neste documento. Para enfatizar isto, tornando ao mesmo

tempo o processo mais intuitivo para o utilizador, cada nó possui conectores (ou docas de ligação) da mesma cor dos nós compatíveis. Um exemplo que ilustra isto, com vários nós conectados, está disponível na Figura 4.15.

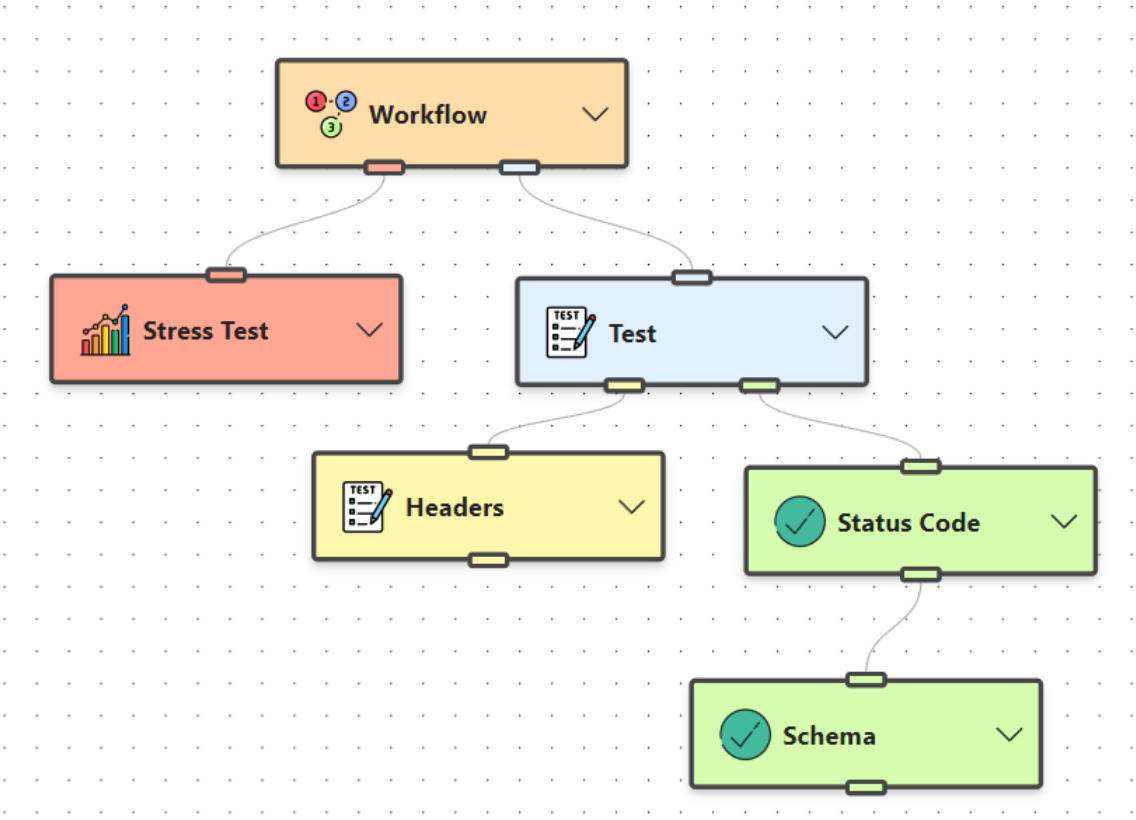


Figura 4.15: Nós conectados de acordo com a linguagem TSL.

4.2.1.5 Barra de Controlos

Como mencionado previamente, o editor inclui uma barra de controlos com várias utilidades, como é possível observar na Figura 4.16. Os primeiros 4 botões são funcionalidades oferecidas pela biblioteca usada para o editor, enquanto que os últimos 4 foram implementados de raiz (o que será detalhado na secção seguinte relativa aos detalhes de implementação).

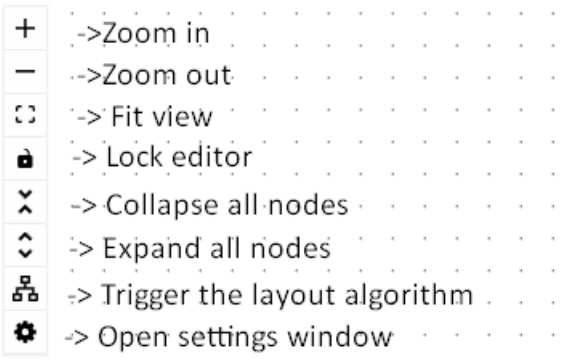


Figura 4.16: Barra de controlos do editor, com a funcionalidade correspondente anotada à frente de cada botão.

4.2.1.6 Guias de utilização

Como referido anteriormente na secção dedicada à analise de requisitos, uma das considerações tidas em conta relativamente ao editor era torná-lo intuitivo e de fácil utilização mesmo para utilizadores pouco experientes ou com pouco conhecimento técnico. Enquanto que isto é parcialmente alcançado pela abordagem tomada, houve a preocupação de ir mais além e procurar explicitamente guiar o utilizador na sua utilização do editor. Desta forma, é possível encontrar ao longo do editor algumas indicações e dicas de utilização, por exemplo em forma de avisos, texto informativo ou *tooltips*. As Figuras 4.17, 4.18, 4.19 e 4.20 apresentam alguns exemplos disto.

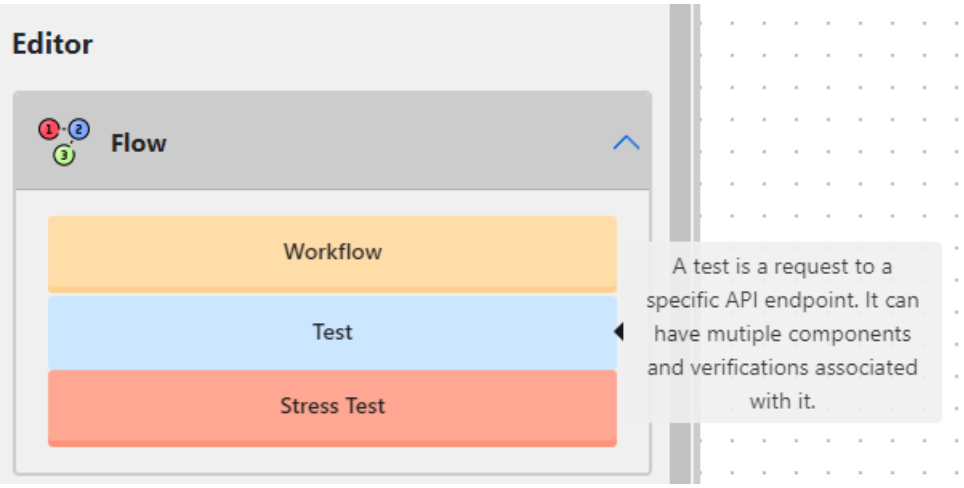


Figura 4.17: Exemplo de uma *tooltip*, mostrada ao utilizador ao fazer *hover* sobre o botão Test.

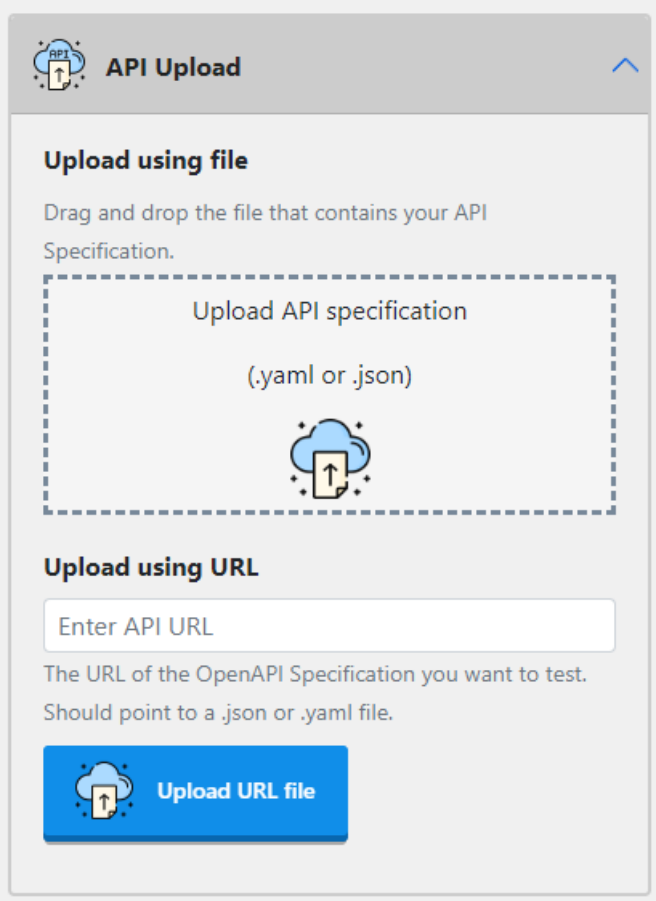


Figura 4.18: Exemplo da utilização de texto informativo, a cinzento claro. Contém informação útil e dicas para o utilizador.

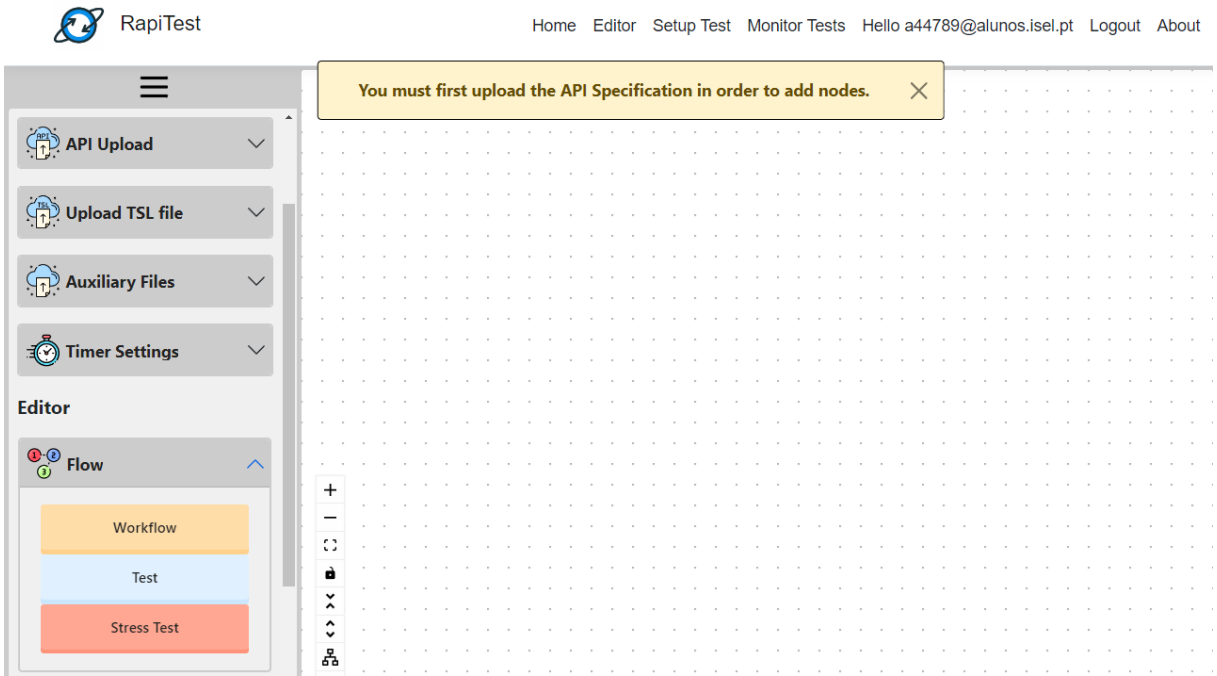


Figura 4.19: Exemplo de uma proteção do editor para prevenir erros. O utilizador é informado de que a ação que pretende realizar não é possível, e o editor indica como proceder.

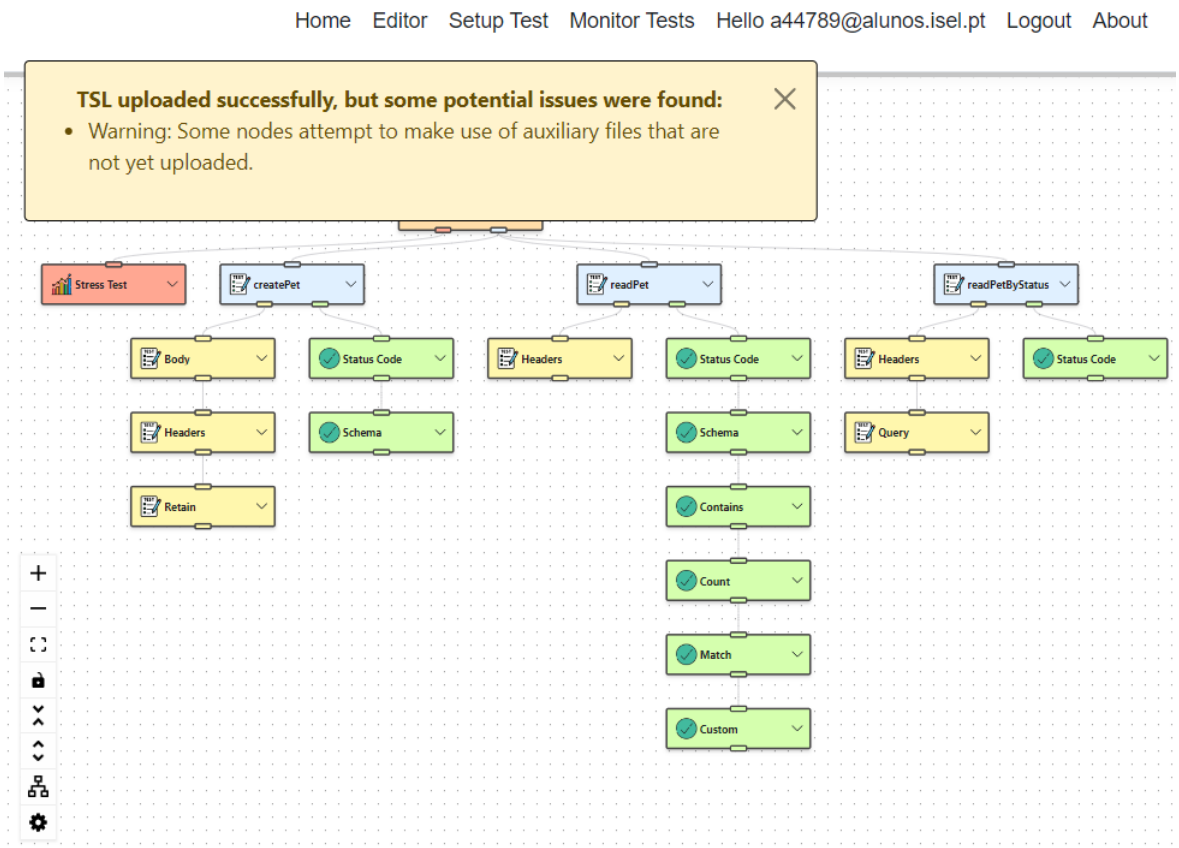


Figura 4.20: Exemplo de um aviso informativo para o utilizador.

4.2.2 Detalhes de implementação

4.2.2.1 Tecnologias utilizadas

Tendo em conta a abordagem escolhida para o editor, nomeadamente a escolha de uma interface *node-based*, foi necessário delinear a melhor estratégia para a implementação do mesmo. Uma vez que a aplicação Web está implementada em JavaScript usando React, a solução passou por procurar bibliotecas já existentes especializadas na construção destas interfaces, e, de preferência, que integrassem facilmente com a base de código existente, particularmente com a biblioteca React.

Apesar de existirem algumas opções diferentes, sem dúvida a biblioteca que reuniu as melhores condições foi a biblioteca React Flow, por uma variedade de razões:

- Como o próprio nome indica, esta biblioteca foi desenvolvida de forma a permitir uma fácil integração com React, pelo que a sua integração na aplicação Web existente torna-se simples.
- A documentação disponível no site é de qualidade, sendo clara, fácil de compreender e concisa. Possui vários exemplos com utilidade prática, o que facilita bastante a compreensão e a aprendizagem da biblioteca.
- É da mais conhecidas e utilizadas biblioteca de React no que toca à construção deste tipo de interfaces, contando atualmente com mais de 500 mil transferências semanais no repositório NPM [25], sendo que por esta razão apresenta uma comunidade ativa, o que normalmente significa maior suporte; por exemplo, torna-se mais fácil encontrar soluções a problemas ao pesquisar por dúvidas ou tutoriais na internet.

Assim, a área de trabalho do editor foi implementada com recurso a esta biblioteca. A barra lateral foi implementada de raiz, sem recurso a outras bibliotecas.

4.2.2.2 Implementação da área de trabalho

A biblioteca React Flow oferece a base do editor pronta a usar - com a base do editor entende-se a tela infinita onde eventualmente se irão inserir os nós, como exemplificado na Figura 4.21.

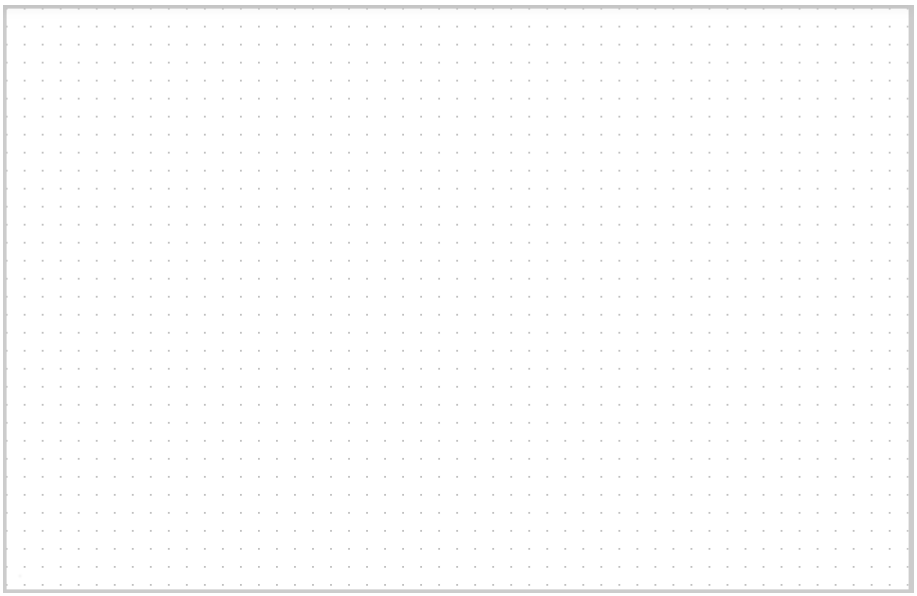


Figura 4.21: Base da área de trabalho, onde serão inseridos os nós.

Esta base é inserida na página através do componente `<ReactFlow/>` cuja funcionalidade é estendida através do fornecimento de várias *props*, como é habitual no ecossistema React.

As *props* (abreviatura de *properties*) são essencialmente uma forma de passar informação entre componentes [26].

Este componente é então personalizado, sendo que recebe, entre outras, as seguintes *props*:

- Lista de nós
- Lista de ligações
- Função de ligação

Como seria de esperar, o editor mostra os nós e ligações com base nas respetivas listas. Quando estas listas são atualizadas, o editor reflete essas alterações na interface.

A função de ligação é utilizada pelo React Flow sempre que o utilizador tenta conectar um nó a outro. Esta função recebe como parâmetro a informação da ligação e, com base nesta informação, deve decidir o que acontece, sendo a decisão mais relevante a de aceitar ou não a ligação. Assim, na implementação realizada, a função verifica qual o tipo de nó na qual origina a ligação e qual o tipo de nó na qual esta termina; desta forma é possível averiguar se a ligação tentada respeita ou não as regras da TSL, e em função disso permitir ou rejeitar a ligação.

Em relação aos nós, apesar de haver vários tipos diferentes consoante a funcionalidade respetiva, todos os nós possuem funcionalidades em comum, pelo que a abordagem passou por primeiro implementar um nó "geral", `<GeneralNode/>`, que servisse como esqueleto a partir do qual os outros nós seriam construídos.

Para a implementação, um nó é essencialmente um simples componente React, que deve obedecer a duas exigências:

- deve receber uma *prop* `data`, usada pelo React Flow;
- deve conter (pelo menos) um outro componente, `<Handle/>`, que é oferecido pela biblioteca e representa um conector para iniciar ou receber uma ligação.

O `<GeneralNode/>` implementa a funcionalidade de colapso uma vez que será comum a todos os nós. Implementa também os `<Handle/>`, sendo que a posição de cada conector é decidida com base nas *props* fornecidas pelos nós mais específicos.

Existe então uma implementação para cada nó oferecido pelo editor, por exemplo `<WorkflowNode/>`, `<BodyNode/>`, etc, que utilizam o `<GeneralNode/>` e apenas implementam o conteúdo específico à sua respetiva funcionalidade.

Como mencionado anteriormente, um aspeto importante que o editor deve conter é um algoritmo de *layout*, de forma a que seja possível dispor os nós e ligações de forma organizada e visualmente apelativa. Infelizmente, a biblioteca React Flow não oferece esta funcionalidade, pelo que foi necessário procurar outras soluções. Felizmente, existem bibliotecas com este objetivo, que inclusivamente integram com a biblioteca React Flow (como aliás o próprio site da biblioteca indica [27]).

A biblioteca utilizada foi a Dagre [28], uma das bibliotecas mencionadas na documentação da React Flow. Esta biblioteca permite implementar um algoritmo de *layout* em árvore de forma relativamente simples. É primeiro necessário criar um objeto grafo disponibilizado pela biblioteca. Depois, quando for para aplicar o algoritmo, a informação sobre todos os nós e ligações é fornecida ao grafo da biblioteca, atualiza internamente as posições dos nós e ligações de forma a produzir uma disposição em árvore. Finalmente, as listas de nós e ligações do editor são substituídas de forma a usarem as novas posições calculadas pela Dagre, e ocorre a consequente "renderização"da interface, que passa a dispor os nós organizados em árvore.

5 Avaliação da solução

Neste capítulo é feita uma análise à solução implementada, nomeadamente relativamente ao editor de testes. Primeiramente é feito um resumo das funcionalidades implementadas, comparando não só com o editor antigo mas também com todos os requisitos identificados inicialmente. Seguidamente são apresentados alguns dados concretos sobre o editor, nomeadamente medições realizadas relativamente a *performance*, uso de memória, entre outros.

5.1 *Deploy*

A solução foi disponibilizada numa máquina virtual e colocada em execução com recurso ao Docker [29]. Os diferentes componentes da solução foram disponibilizados em imagens Docker, das quais o ficheiro `docker-compose.yml` faz uso para colocar toda a aplicação em execução. Todo o código fonte da solução está disponível no repositório público alojado no Github [30].

5.2 Capacidades do editor

De forma a compreender os resultados obtidos, é importante fazer a comparação entre os requisitos definidos antes da implementação e as funcionalidades implementadas, para perceber se o produto implementado vai de encontro aos objetivos definidos e cumpre o seu propósito.

Fazendo este exercício, verifica-se que o editor desenvolvido cumpre os objetivos a que se propôs:

- Permite não só a criação de um ficheiro TSL de raiz, mas também a edição de um ficheiro existente - funcionalidade essencial que a interface antiga não possuía;
- Suporta todas as funcionalidades da TSL, permitindo assim uma compatibilidade total entre a interface e qualquer ficheiro, algo que também não era possível na interface antiga;
- Oferece todas as outras funcionalidades que junto com a TSL forma uma configuração de teste, nomeadamente o upload e processamento da especificação OAS, as configurações de temporização e a possibilidade de usar ficheiros auxiliares;
- Procura sempre que possível ajudar o utilizador para tornar a utilização do editor o mais intuitiva possível, através de texto informativo, *tooltips*, campos pré-preenchidos, entre outros. Porém, importa referir que, apesar de estas funcionalidades terem sido de facto implementadas, se cumprem ou não o seu propósito é algo que apenas o *feedback* dos utilizadores poderá confirmar. Isto será mencionado na secção de Trabalho Futuro.

5.3 Medições

Nesta secção são apresentas algumas medições concretas que se entendam relevantes para avaliar o desempenho do editor. Como referência, foram usados 3 ficheiros TSL diferentes para os testes: um de tamanho pequeno, contendo um *workflow* simples com apenas 6 nós; um de tamanho médio, com 20 nós, e um de tamanho muito grande, com 200 nós (irrealista num cenário real, mas útil para testes).

As medições foram realizadas num computador com as seguintes especificações:

- Modelo: HP Laptop 15s-eq2xxx
- Sistema Operativo: Microsoft Windows 11 Home v. 10.0.22631
- Processador: AMD Ryzen 7 5700U with Radeon Graphics, 1801 Mhz, 8 Cores, 16 Threads
- Memória: 16 GB RAM

5.3.1 Tempos de carregamento

Uma das medições que foi feita foi o tempo que demora o carregamento e processamento de um ficheiro TSL, bem como a criação do estado e preenchimento do editor em consequência do carregamento do ficheiro. Assim, na tabela 5.1 são apresentados em milissegundos os tempos que englobam este processo - desde imediatamente antes do carregamento do ficheiro até imediatamente a seguir da criação dos nós correspondentes. Para recolher estes valores foi usada a API Performance [31]. A Figura 5.1 apresenta os dados em forma de gráfico, facilitando a visualização.

	Pequeno	Médio	Grande
#1:	317.9 ms	697.7 ms	5834.7 ms
#2:	217.8 ms	602.6 ms	4977.1 ms
#3:	225.7 ms	650.7 ms	6007.9 ms
#4:	248.2 ms	564.4 ms	4830.7 ms
#5:	279.8 ms	661.7 ms	4918.7 ms
Média:	257.9 ms	635.4 ms	5313.8 ms

Tabela 5.1: Valores obtidos relativos aos tempos de carregamento de um ficheiro TSL.

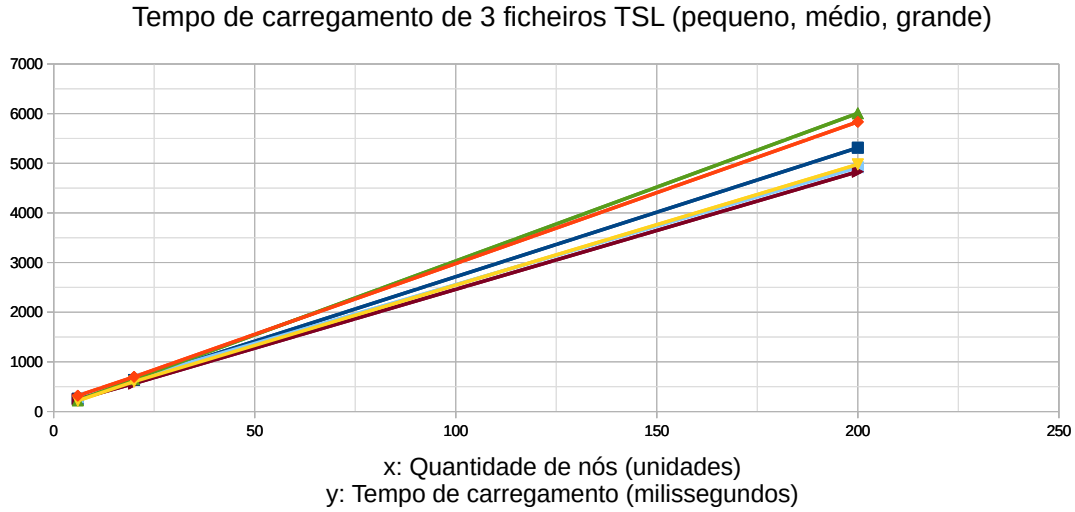


Figura 5.1: Visualização em gráfico das medidas apresentadas na tabela 5.1; é possível observar o aumento linear do tempo.

Ao analisar os resultados, observa-se que os tempos para os ficheiros Pequeno e Médio são bastante satisfatórios. Primeiramente, são tempos inferiores a 1 segundo, o que, considerando todo o processamento, se traduz numa experiência de utilização satisfatória, pois o utilizador praticamente não sente a espera desde que carrega o ficheiro até que os nós são criados. Mesmo no caso do ficheiro Grande, considerando a quantidade enorme de nós, o tempo médio

é aceitável ficando à volta dos 5 segundos. Embora não ideal, este é um caso excecional e para efeitos de exemplo, pois raramente um utilizador trabalhará com um ficheiro tão grande. É possível observar que com o acréscimo dos nós, o aumento do tempo é praticamente linear, e não exponencial, o que também é um indicador positivo.

5.3.2 Memória

Outra medida relevante para avaliar o desempenho do editor é o uso de memória. A Performance API também permite obter dados relativamente à memória, pelo que foi usada novamente. Essencialmente foram medidos 2 valores: o *heap* total alocado e o *heap* atualmente usado. De forma a perceber qual o impacto resultante do acréscimo de muitos nós, as medidas foram tiradas em 4 momentos diferentes: após o carregamento da página e depois após o carregamento de cada um dos 3 ficheiros em questão. Os valores em bytes estão disponíveis na tabela 5.2.

	Total alocado	Total usado
Início:	96057515 b	90243923 b
Pequeno:	112587105 b	94045813 b
Médio:	116788180 b	98793340 b
Grande:	170728033 b	136039477 b

Tabela 5.2: Valores obtidos relativos ao uso de memória por parte do editor.

Com base nestes valores, observa-se que a memória usada cresce proporcionalmente à quantidade de nós, havendo um acréscimo reduzido no carregamento dos ficheiros Pequeno e Médio, e um acréscimo maior após o carregamento do ficheiro Grande.

5.3.3 Responsividade

Relativamente à responsividade do editor, foram usadas duas ferramentas distintas para obter medidas concretas: a biblioteca web-vitals [32] e a ferramenta Google Lighthouse [33]. Na tabela 5.3 são apresentadas 4 medidas:

- INP (Interaction to Next Paint) - mede o tempo que a página demora a responder a interações do utilizador e fazer as atualizações necessárias na interface; deseja-se um valor o mais baixo possível, idealmente abaixo dos 200ms [34];
- CLS (Cummulative Layout Shift) - índice que mede a estabilidade da página no que toca a mudanças no layout da página; deseja-se um valor o mais baixo possível, idealmente abaixo dos 0.1 [35];
- TTFB (Time to First Byte) - mede o tempo entre o pedido do browser e a chegada da resposta do servidor (inclui desta forma a latência e o tempo de processamento); deseja-se um valor o mais baixo possível, idealmente abaixo dos 800ms [36];
- LCP (Largest Contentful Paint) - tempo que demora até o maior elemento da página carregar por completo; deseja-se um valor o mais baixo possível, idealmente abaixo dos 2500ms [37].

	web-vitals	Google Lighthouse
INP:	80 ms	170 ms
CLS:	0.003	0.007
TTFB:	2051 ms	2070 ms
LCP:	4996 ms	2100 ms

Tabela 5.3: Valores obtidos relativos à capacidade de responsividade do editor.

Ao analisar estes resultados, observa-se que INP e CLS apresentam excelentes valores, bastante reduzidos. Isto significa que a página apresenta um elevado grau de estabilidade e responsividade, indicadores bastante positivos. Pelo contrário, TTFB e LCP apresentam valores mais elevados do que desejável, indicando que poderá haver espaço para melhoria no que toca ao processamento do servidor e ao carregamento inicial da página.

6 Conclusão

Neste capítulo são apresentadas as conclusões obtidas após a realização do trabalho. É feito um resumo do trabalho desenvolvido, como este vai ou não de encontro aos problemas identificados, e se o resultado final cumpre os objetivos propostos. Para concluir, são feitas considerações acerca de aspetos em falta ou a melhorar e são apresentadas sugestões para trabalho futuro.

6.1 Conclusões

Concluído o projeto, é então possível fazer um resumo de todo o trabalho realizado e tirar as devidas conclusões. O objetivo principal deste trabalho, que inclusive dá nome ao título da tese, era a elaboração de um Editor para testes semi-automáticos de Web API. Este editor teria como propósito principal fornecer uma interface gráfica que permitisse a qualquer utilizador tirar partido da linguagem TSL, desenvolvida num projeto anterior, para realizar testes personalizados a Web APIs.

O editor desenvolvido foi implementado diretamente na aplicação existente, integrando-se na mesma de forma natural. Houve uma preocupação e foco em providenciar uma interface o mais simples e intuitiva possível de forma a alargar o alcance da aplicação a utilizadores com pouco conhecimento técnico. Considerando a complexidade e todas as funcionalidades da linguagem TSL, esta tarefa não foi trivial, porém a abordagem *node-based* provou ser uma mais-valia e, apesar de certamente haver espaço para melhorias, o editor cumpre os seus objetivos.

Assim, é seguro afirmar que o objetivo principal deste trabalho foi concluído, pois a aplicação *RapiTest* possui agora um editor *node-based* que suporta por completo todos os requisitos identificados. É possível realizar qualquer espécie de configuração de teste a partir do editor, sem perder nenhum tipo de capacidade ou flexibilidade relativamente ao que seria a elaboração manual de um ficheiro TSL. Adicionalmente, os resultados obtidos mostram um editor com elevado grau de responsividade e interatividade, capaz de suportar ficheiros TSL de grandes dimensões.

No entanto, isto não significa que o trabalho está concluído permanentemente. A próxima

secção irá indicar pontos de melhoria e trabalho futuro de forma a que a aplicação *RapiTest* possa cada vez mais ser uma opção viável e de qualidade para a realização de testes a Web APIs.

6.2 Trabalho Futuro

Como é habitual, especialmente num contexto de trabalho de universidade, o trabalho raramente fica verdadeiramente concluído, havendo sempre espaço a melhorias e novos desenvolvimentos para tornar a solução desenvolvida cada vez melhor. Naturalmente este é o caso da aplicação *RapiTest* no geral, e do editor desenvolvido em particular.

Relativamente ao editor implementado, a principal falha identificada não é na verdade uma questão de uma implementação menos conseguida ou uma funcionalidade que ficou em falta; é no entanto a falta de *feedback* de utilizadores relativamente ao produto desenvolvido. Como já dito anteriormente, este editor é uma ferramenta para ser usada por diferentes tipos de utilizadores, inclusive aqueles com pouco conhecimento técnico. Tendo em conta a ênfase na usabilidade e simplicidade, seria bastante importante disponibilizar a ferramenta para que vários utilizadores pudessem usá-la e testá-la, fornecendo o seu *feedback* sobre o editor. Apesar de a ferramenta ter sido *deployed* numa máquina virtual disponibilizada pelo ISEL, isto foi feito num momento tardio, pelo que não houve possibilidade de contactar utilizadores de teste de forma a obter *feedback* durante o desenvolvimento do produto. Assim, este é um aspeto que deve ser corrigido no futuro, pois pode trazer inúmeras mais-valias e novas perspetivas sobre os resultados conseguidos.

Existem também algumas funcionalidades que podem ser melhoradas no sentido de tornar o editor ainda mais intuitivo e "user-friendly". Talvez a mais evidente prende-se com a questão da ordem relativa dos *workflows* e testes. Esta funcionalidade permite ao utilizador alterar a ordem dos seus *wokflows* e testes mesmo após a sua criação. Implementar esta funcionalidade num editor *node-based* de forma intuitiva não é trivial e existem algumas formas diferentes de o fazer. A forma atual é através de um campo em cada um dos nós em que o utilizador especifica a ordem relativa que o nó deve ter. Por exemplo, se um nó Workflow tiver a ordem 1 será o primeiro *workflow* a ser executado; se outro tiver a ordem 2 será o segundo *workflow* a ser executado. Esta forma funciona, mas existem alternativas que poderão ser melhores. Por exemplo, uma solução pensada foi a possibilidade de ligar de forma direcionada os nós Workflow e Teste uns aos outros. Assim, a ordem seria decidida consoante as ligações feitas pelo utilizador. Porém esta abordagem poderia tornar a área de trabalho demasiado complexa, introduzindo demasiadas e diferentes ligações. Poderia também existir uma nova vista no editor dedicada apenas à alteração da ordem, apresentando os *workflows* e testes numa lista em que o utilizador poderia fazer "drag-and-drop"para alterar a ordem. Porém, a introdução desta nova vista poderia também causar confusão ao utilizador. Existem outras pequenas melhorias a fazer, tal como o aperfeiçoamento da interface de alguns dos nós, a possibilidade de customização de aspetos como o tamanho dos conectores ou a espessura das conexões ou a introdução de uma funcionalidade para ver ou até mesmo editar manualmente os ficheiros carregados.

Numa ótica mais geral, relativamente à aplicação *RapiTest* no seu todo, existem também alguns pontos relevantes para trabalho futuro. Por exemplo, um aspeto que surgiu em consideração na realização deste trabalho foi a melhoria da vista dos testes realizados, nomeadamente a página que apresenta os resultados, que não é muito intuitiva. Inclusivamente, poderá estudar-se a possibilidade de incorporar esta funcionalidade dentro do próprio editor, para que o utilizador possa realizar as suas configurações de teste e ter acesso imediato aos resultados sem sair da mesma página. Em termos das capacidades de testagem, dois pontos importantes que foram identificados foram a melhoria dos testes gerados automaticamente pela ferramenta e a expansão das verificações fornecidas nativamente, com foco em verificações relacionadas com segurança, como proteção contra Cross-Site Scripting ou SQL Injection. Estes dois aspetos foram inclusive identificados e propostos no início do trabalho como objetivos secundários; porém acabaram por não ser trabalhados em detrimento do objetivo principal que era a construção do editor de testes.

Bibliografia

[1] *Postman State of the API Report*. URL: https://www.postman.com/state-of-api/. (acedido em: 12-10-2024) (ver p. 1).

[2] R. T. Fielding e R. N. Taylor. "Principled design of the modern web architecture". Em: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), pp. 115–150 (ver p. 1).

[3] D. Felício, J. Simão e N. Datia. "Rapitest: Continuous black-box testing of restful web apis". Em: *Procedia Computer Science* 219 (2023), pp. 537–545 (ver pp. 2, 9).

[4] E. Viglianisi, M. Dallago e M. Ceccato. "RestTestGen: Automated Black-Box Testing of RESTful APIs". Em: *IEEE* (2020) (ver p. 5).

[5] N. Laranjeiro, J. Agnelo e J. Bernardino. "A Black Box Tool for Robustness Testing of REST Services". Em: *IEEE* (2021) (ver p. 5).

[6] A. Martin-Lopez, S. Segura e A. Ruiz-Cortés. "RESTest: Automated Black-Box Testing of RESTful Web APIs". Em: (2021) (ver p. 5).

[7] *Postman*. URL: https://www.postman.com/. (acedido em: 10-03-2023) (ver p. 6).

[8] *SoapUI*. URL: https://www.soapui.org/. (acedido em: 10-03-2023) (ver p. 6).

[9] E. Hosick. *Visual Programming Languages - Snapshots*. Fev. de 2014. URL: http://blog.interfacevision.com/design/design-visual-progarmming-languages-snapshots/. (acedido em: 10-03-2023) (ver p. 6).

[10] W. R. Sutherland. "The on-line graphical specification of computer procedures". Em: (1966) (ver p. 7).

[11] T. O. Ellis, J. F. Heafner e W. L. Sibley. "The GRAIL Project: An experiment in man-machine communications". Em: (1969) (ver p. 7).

[12] *Unreal Engine*. URL: https://www.unrealengine.com/en-US. (acedido em: 10-03-2023) (ver p. 7).

[13] *Blueprints Visual Scripting*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/blueprints-visual-scripting-in-unreal-engine. (acedido em: 12-10-2024) (ver p. 7).

[14] *User Interface / Nodes / Introduction*. URL: https://docs.blender.org/manual/en/latest/interface/controls/nodes/introduction.html. (acedido em: 10-03-2023) (ver p. 7).

[15] *Introduction - Blender*. URL: https://docs.blender.org/manual/en/latest/interface/controls/nodes/introduction.html. (acedido em: 12-10-2024) (ver p. 7).

[16] *Houdini*. URL: https://www.sidefx.com/products/houdini/. (acedido em: 10-03-2023) (ver p. 7).

[17] *DaVinci Resolve*. URL: https://www.blackmagicdesign.com/pt/products/davinciresolve. (acedido em: 10-03-2023) (ver p. 7).

[18] *OutSystems*. URL: https://www.outsystems.com/. (acedido em: 10-03-2023) (ver p. 7).

[19] *Node-RED*. URL: https://nodered.org/. (acedido em: 10-03-2023) (ver p. 7).

[20] *Node-RED - Browser-based flow editing*. URL: https://nodered.org/#:~:text=Node%2DRED%20provides%20a%20browser,runtime%20in%20a%20single%2Dclick.. (acedido em: 12-10-2024) (ver p. 7).

[21] *Postman Flows*. URL: https://learning.postman.com/docs/postman-flows/gs/flows-overview/. (acedido em: 10-03-2023) (ver p. 7).

[22] *OpenAPI Specification*. URL: https://spec.openapis.org/oas/latest.html. (acedido em: 12-10-2024) (ver p. 10).

[23] *YAML*. URL: https://yaml.org/. (acedido em: 10-03-2023) (ver p. 12).

[24] *AMQP*. URL: https://www.amqp.org/. (acedido em: 10-03-2023) (ver p. 19).

[25] *NPM - React Flow*. URL: https://www.npmjs.com/package/reactflow. (acedido em: 12-10-2024) (ver p. 38).

[26] *React - Passing Props to a Component*. URL: https://react.dev/learn/passing-props-to-a-component. (acedido em: 12-10-2024) (ver p. 39).

[27] *React Flow - Layouting*. URL: https://reactflow.dev/learn/layouting/layouting. (acedido em: 12-10-2024) (ver p. 39).

[28] *Dagre*. URL: https://github.com/dagrejs/dagre. (acedido em: 12-10-2024) (ver p. 40).

[29] *Docker*. URL: https://www.docker.com/. (acedido em: 12-10-2024) (ver p. 41).

[30] *Github | RAPITest*. URL: https://github.com/Alexandre-Rocha/RAPITest. (acedido em: 12-10-2024) (ver p. 41).

[31] *Performance - Web APIs | MDN*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance. (acedido em: 12-10-2024) (ver p. 43).

[32] *web-vitals*. URL: https://github.com/GoogleChrome/web-vitals. (acedido em: 12-10-2024) (ver p. 45).

[33] *Google Lighthouse*. URL: https://developer.chrome.com/docs/lighthouse/overview. (acedido em: 12-10-2024) (ver p. 45).

[34] *Interaction to Next Paint (INP)*. URL: https://web.dev/articles/inp. (acedido em: 12-10-2024) (ver p. 45).

[35] *Google Lighthouse*. URL: https://web.dev/articles/cls. (acedido em: 12-10-2024) (ver p. 45).

[36] *Time to First Byte (TTFB)*. URL: https://web.dev/articles/ttfb. (acedido em: 12-10-2024) (ver p. 45).

[37] *Largest Contentful Paint (LCP)*. URL: https://web.dev/articles/lcp. (acedido em: 12-10-2024) (ver p. 45).