NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTAMENTO DE INFORMÁTICA

RAFAEL DE ALBUQUERQUE RIVEIRO

Licenciado em Engenharia Informática

JEPREST 2.0

TESTING REST APPLICATIONS WITH CUSTOMIZED SEMANTICS

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa

Setembro, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTAMENTO DE
INFORMÁTICA

JEPREST 2.0

TESTING REST APPLICATIONS WITH CUSTOMIZED SEMANTICS

RAFAEL DE ALBUQUERQUE RIVEIRO

Licenciado em Engenharia Informática

Orientador: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA University Lisbon

Coorientador: Filipe Freitas
Assistant Professor, ISEL

Júri

Presidente: Dr. João Miguel da Costa Magalhães
Prof. Catedrático, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Vogais: Dr. Jácome Miguel Costa da Cunha
Prof. Associado, Faculdade de Engenharia da Universidade do Porto

Dr. Nuno Manuel Ribeiro Preguiça
Prof. Catedrático, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

MESTRADO EM ENGENHARIA INFORMÁTICA

Universidade NOVA de Lisboa
Setembro, 2024

JepREST 2.0
Testing REST Applications With Customized Semantics

Este documento foi gerado com o processador (pdf/Xe/Lua)LaTeX e o modelo NOVAthesis (v7.1.11) [46].

Agradecimentos

Gostaria de começar por expressar a minha gratidão aos meus orientadores, os Professores Nuno Preguiça e Filipe Freitas. Quero agradecer-lhes o apoio e orientação constantes, que me permitiram realizar um trabalho de que me orgulho.

Agradeço também à nossa instituição, a Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, por todos estes anos de aprendizagem e crescimento, tanto académico como pessoal.

Gostaria de agradecer a todos os amigos e colegas que conheci ao longo destes anos na universidade, pela amizade, companheirismo e momentos partilhados, que tornaram este percurso mais interessante e enriquecedor. Em especial, obrigado Duarte por tornares as idas à faculdade tão divertidas.

Por último, agradeço aos meus pais e ao meu irmão por estarem sempre ao meu lado, em todas as circunstâncias. Terminar a faculdade sem o vosso apoio teria sido impossível. A minha gratidão é imensa.

”

«Thinking doesn't guarantee that we won't make mistakes. But not thinking guarantees that we will.»

— Leslie Lamport

Abstract

Nowadays, many applications offer REST APIs to facilitate communication between systems and allow scalable and flexible service integration. However, the implementation of these applications presents a challenge for programmers due to the complexity involved. The JepREST tool assists programmers in detecting errors in applications, simplifying the development process.

Nevertheless, this tool is unable to test applications that do not follow REST semantics, which leaves many real-world applications unaddressed.

In this dissertation, we extended JepREST to enable the tool to test the correctness of REST applications with customized functionalities. This involved expanding the model defined in the tool to allow for the verification of such applications. Since the model needs to understand the behavior of the operations, a language was designed and implemented to specify the internal behavior of API operations. The use of this language allows operations to be described as sequences of simple REST methods. Thus, JepREST uses the model to verify the correctness of each operation present in these sequences.

To generate the history of invocations and results of the operations that JepREST analyzes based on the defined model, a generator of operations was developed. This generator associates each client with an operation to be executed at each moment, depending on the result of the previous operation. To determine the operation that a client will perform, we implemented a tool that generates an operation transition graph from the application's logs, thereby simulating a realistic execution of the application.

To evaluate the ability of the new version of JepREST to identify correctness issues, we used the tool to test an application implemented by us and another developed by other programmers. The presented results demonstrate that this tool can detect various errors, including logical failures, inconsistencies in responses, and compliance issues with the API specifications.

Keywords: Distributed Systems, REST, Jepsen, JepREST, Test Generation, Functional testing, Testing REST APIs

Resumo

Atualmente, muitas aplicações utilizam o estilo arquitetural REST como base para a criação de APIs, facilitando a comunicação entre sistemas e permitindo a integração de serviços de forma escalável e flexível. No entanto, a implementação destas aplicações representa um desafio para os programadores, devido à complexidade envolvida. A ferramenta JepREST auxilia os programadores na deteção de erros nas aplicações, simplificando o processo de desenvolvimento.

Contudo, esta ferramenta não é capaz de testar aplicações que não seguem a semântica REST, o que deixa de lado muitas aplicações do mundo real.

Nesta dissertação, estendemos o JepREST para que seja possível utilizar a ferramenta para testar a correção de aplicações REST com funcionalidades personalizadas. Isto envolveu a extensão do modelo definido na ferramenta para que este permitisse a verificação deste tipo de aplicações. Como o modelo precisa de compreender o funcionamento das operações, foi desenhada e implementada uma linguagem que permite especificar o comportamento interno das operações da API. O uso desta linguagem permite descrever as operações como sequências de operações REST simples. Desta forma, o JepREST utiliza o modelo para verificar a correção de cada operação presente nessas sequências.

Para gerar o histórico de invocações e resultados das operações que o JepREST analisa com base no modelo definido, foi desenvolvido um gerador de operações que associa a cada cliente as operações a serem executadas em cada momento, dependendo do resultado da operação anterior. Para determinar a operação que o cliente deve realizar, foi implementada uma ferramenta que gera um grafo de transições de operações a partir dos *logs* da aplicação, simulando assim uma execução realista da mesma.

Para avaliar a capacidade que a nova versão do JepREST tem em identificar problemas de correção, utilizámos a ferramenta para testar uma aplicação implementada por nós e uma desenvolvida por outros programadores. Os resultados apresentados demonstram que esta ferramenta consegue detetar variados erros, incluindo falhas de lógica, incoerências nas respostas e problemas de conformidade com as especificações da API.

Palavras-chave: Sistemas Distribuídos, REST, Jepsen, JepREST, Geração de Testes, Teste

funcional, Teste de APIs REST

Índice

Índice de Figuras

Índice de Listagens

1

Introdução

Este capítulo descreve o contexto, a motivação, o problema e a abordagem tomada para resolvê-lo. Também apresenta as contribuições do trabalho e a estrutura do documento.

1.1 Contexto

Atualmente, a maioria de aplicações *web* que usamos no nosso dia a dia são sistemas distribuídos. Um sistema distribuído é um sistema onde vários computadores trabalham em conjunto para fornecer um grupo de funcionalidades. Cada tarefa é dividida entre os diferentes componentes do sistema que se coordenam para, no fim, gerar um resultado que possa ser entregue a um cliente. Numa aplicação bem desenvolvida, isto pode supor uma maior eficiência e rapidez na execução de operações comparando com um sistema a correr num único dispositivo. Um sistema distribuído também tem a possibilidade de ser mais tolerante a falhas, pois se um componente falhar os outros coordenam-se para executar as operações que lhe correspondiam. Estes benefícios, entre muitos outros, são a razão pela qual muitas aplicações *web* são sistemas distribuídos.

Durante vários anos, a interação cliente-servidor, ou servidor-servidor, variava conforme o sistema utilizado. Ao não haver nenhum consenso sobre como deveriam ser feitas as interações dos serviços *web*, cada sistema usava o protocolo que os desenvolvedores achassem mais conveniente, dificultando a integração entre sistemas distintos. No ano 2000, Roy Fielding [26] apresentou um estilo arquitetural chamado REST, que visa resolver estes problemas de interação na *web*. O REST define uma API que indica como devem ser invocadas as operações de um sistema e quais são os possíveis resultados de cada uma. Isto simplifica bastante a parte da interação na *web*, de modo que um sistema, ou aplicação, pode definir um só modelo de interação para interagir com qualquer sistema REST, em vez de ter que definir um modelo diferente por cada sistema, dependendo do protocolo que este último use.

Embora o REST facilitasse a interação entre sistemas *web*, a documentação das APIs podia divergir de sistema a sistema, complicando a interoperabilidade. Então, em 2016, a OpenAPI Specification [53], que é uma forma de documentar APIs HTTP, tornou-se a

especificação padrão para documentar APIs REST. Esta documentação indica a informação necessária para executar pedidos a uma API. Entre estas informações sobre as operações estão: a descrição, o método HTTP que deve ser usado, que parâmetros são necessários e quais são as respostas possíveis.

1.2 Motivação

Os humanos são conhecidos por cometer erros em qualquer atividade que façam, mas também por ter a capacidade de reconhecê-los e resolvê-los. Na área da construção de software, isto não muda. Os desenvolvedores, até os mais experientes, acabam por cometer sempre algum erro. Em grande parte dos casos, os compiladores notificam os desenvolvedores sobre a existência de erros para que eles possam corrigi-los antes de publicar a aplicação. Apesar disto, existem muitos erros que não são detetados até a utilização da aplicação, e alguns até só são descobertos bastante tempo após esta estar a correr.

Um sistema distribuído é bastante complexo e difícil de desenhar e implementar. Os desafios associados à gestão da concorrência entre os diferentes processos complicam a deteção de possíveis erros. Um erro num sistema distribuído pode ter consequências más, como o desempenho do sistema piorar; muito más, como o resultado de algumas operações acabar errado; ou, até, catastróficas, como o sistema ficar inabilitado e inacessível. Todos estes problemas são indesejáveis, pois pioram a experiência dos utilizadores, podendo até perdê-los.

Para mitigar a possibilidade de acontecerem tais complicações, começaram a aparecer aplicações de teste de sistemas distribuídos, como o Jepsen [35], que verifica a correção de um sistema segundo o modelo definido. O propósito destas ferramentas é verificar se o sistema opera conforme o esperado, independentemente das operações executadas, e identificar os erros existentes para que os desenvolvedores os possam corrigir. Para encontrar estes erros, é necessário criar um conjunto de testes que explore várias combinações de parâmetros e operações do sistema.

No JepREST decidimos focar no teste de sistemas distribuídos baseados em REST, pois são os mais comuns na *web*, como relata um estudo feito em 2023 pela plataforma de uso e criação de APIs, Postman, que indica que 86% dos desenvolvedores *web* usam REST [1].

1.3 Problema

O estilo arquitetural REST baseia-se na organização das aplicações em recursos e define uma semântica que estabelece o significado das operações que modificam estes recursos. Isto torna o sistema mais escalável e fomenta a interoperabilidade entre sistema distribuídos, mas também limita as ações que as operações podem executar. Devido a isto, muitos sistemas REST oferecem operações que não respeitam a semântica, apresentando funcionalidades personalizadas que não são incluídas na especificação da API. Por exemplo,

a operação *move* [21] do sistema Dropbox [20], que remove um ficheiro de uma pasta e coloca-o noutra, aparece na especificação como um POST. Desta forma, não existe uma maneira de saber se a ação realizada por uma operação é indicada pelo método HTTP ou não. Esta é uma das desvantagens da primeira versão do JepREST, pois o modelo de verificação da correção do estado é guiado pela semântica da arquitetura, o que impossibilita que esta ferramenta seja utilizada para testar a maioria de aplicações REST existentes.

Para testar a funcionalidade de uma aplicação REST de forma rigorosa, pretende-se simular a execução real da aplicação, como se as operações fossem realizadas por clientes reais. Na primeira versão do JepREST, o utilizador é responsável por definir manualmente os cenários de teste, especificando a sequência de operações a serem testadas. Dado que o objetivo é que esses testes reflitam as operações realizadas numa aplicação real, pode ser complicado para um programador definir manualmente todas essas operações num ficheiro.

1.4 Abordagem

Nesta dissertação, estendemos a ferramenta JepREST para que possa testar a correção de aplicações REST com funcionalidades personalizadas.

Uma operação cujo comportamento não pode ser definido pela semântica padrão da arquitetura REST é aquela que executa uma ação diferente do método HTTP associado ou que envolve uma sequência de operações internas. Como o JepREST é uma ferramenta *black-box* que apenas precisa da especificação da API de uma aplicação para poder testá-la, foi necessário criar uma linguagem que permitisse descrever as operações que não seguem a semântica padrão. A linguagem definida é baseada em anotações Java e possibilita que os programadores especifiquem este tipo de operações dividindo-as em sequências de operações descritas com a semântica REST.

Quando uma operação é identificada como personalizada, o modelo definido neste trabalho determina como cada operação da sequência de operações internas deve ser analisada e de que forma cada uma altera o estado. Deste modo, o JepREST passa a verificar a existência de incoerências entre um passo de uma operação e o estado gerado pelos passos e operações executados anteriormente.

O histórico de pedidos e respostas analisados pelo JepREST, utilizando o *checker* Knossos e o modelo por nós definido, é gerado durante a fase de teste da ferramenta. Nesta fase, o JepREST cria vários clientes que interagem com a aplicação. Para simular clientes realistas, implementámos uma ferramenta que, a partir dos *logs* da aplicação, constrói um grafo de transições de operações. Este grafo determina a probabilidade de um cliente executar uma operação com base na última que realizou e no código de estado do resultado obtido. De modo a integrar o grafo gerado por esta ferramenta no JepREST, desenvolvemos um novo gerador de operações do Jepsen [41], que controla as operações que os clientes executam durante a fase de teste. Sempre que um cliente recebe uma

resposta, este gerador utiliza o grafo para determinar a próxima operação que lhe vai ser associada, dependendo do código presente na resposta.

1.5 Contribuições

Esta dissertação é uma contribuição para a expansão da ferramenta de teste de APIs REST JepREST [66, 61, 68]. No nosso trabalho estendemos o JepREST para poder testar a funcionalidade de aplicações REST que oferecem operações que não seguem a semântica padrão da arquitetura.

As contribuições deste trabalho são:

- Uma linguagem que permite especificar operações de APIs REST com funcionalidades personalizadas.
- Uma nova versão do modelo do *checker* Knossos, definido na versão anterior do JepREST, que permite detetar problemas de linearização em aplicações REST com semântica personalizada.
- Uma ferramenta que gera um grafo de transições de operações da API a partir dos *logs* da aplicação.
- Um gerador de operações do Jepsen que utiliza o grafo para determinar as operações que cada cliente do JepREST deve realizar durante a fase de teste.

1.6 Estrutura do Documento

Para além deste capítulo de introdução, este relatório tem os seguintes capítulos:

- **Capítulo 2 - Conceitos:** Apresenta os conceitos fundamentais para compreender a informação relatada nos capítulos a seguir.
- **Capítulo 3 - Trabalho Relacionado:** Primeiro, exibe as técnicas usadas para verificar a funcionalidade de um sistema. Depois, expõe várias ferramentas de teste, nas quais umas suportam testar APIs REST e outras necessitam ser estendidas para executar tal tarefa.
- **Capítulo 4 Desenho da Solução:** Apresenta em detalhe a arquitetura da ferramenta JepREST. Este capítulo clarifica as funcionalidades que já existiam na versão anterior e descreve as que foram adicionas neste trabalho. Também apresenta uma documentação para a linguagem que especifica as operações com comportamento personalizado, e explica a nova ferramenta de geração de testes.
- **Capítulo 5 Implementação da Solução:** Descreve a implementação do JepREST e da ferramenta que gera o grafo de transições de operações.

- **Capítulo 6 Avaliação Experimental:** Apresenta o processo de avaliação realizado para comprovar a capacidade de deteção de erros da nossa versão do JepREST. Para isso, executámos a ferramenta para analisar a correção de duas aplicações: uma com erros, criada especificamente para testar a ferramenta, e outra desenvolvida por outros programadores. Neste capítulo, mostramos como foram gerados os testes e analisamos os resultados obtidos.
- **Capítulo 7 Conlusão:** Resume as funcionalidades do JepREST 2.0 e apresenta algumas melhorias e adições a serem implementadas em trabalhos futuros, com o objetivo de aprimorar ainda mais a utilidade da ferramenta.

2

Conceitos

Neste capítulo são apresentados os dois conceitos fundamentais para a elaboração desta dissertação: *Representational State Transfer* e *OpenAPI Specification*.

2.1 REST

***RE**presentational **S**tate **T**ransfer* (REST) [26] é um estilo arquitetural que define padrões nas comunicações entre sistemas na *web*, com o intuito de minimizar a latência e o número de mensagens trocadas na rede e de maximizar a independência e escalabilidade da implementação de componentes [80, 25].

Devido a que o REST se foca na semântica da conexão entre sistemas, e não na semântica dos componentes [25], o código do cliente pode ser alterado sem mudar o resultado das operações do servidor e vice-versa [9], criando uma total separação entre cliente e servidor. Assim, é possível criar um servidor que satisfaça qualquer cliente que necessite os serviços sem adicionar complexidade, bastando definir uma API (*Application Programming Interface*) que especifique como cada operação disponibilizada deve ser invocada, que parâmetros são necessários, as respostas esperadas e que erros podem ocorrer.

O termo mais importante e sobre o qual gira toda a lógica do REST é o recurso. Um recurso é qualquer conteúdo guardado no servidor que possa ser acedido por um cliente. Ou como o Roy Fielding definiu: "Qualquer informação que possa ser nomeada pode ser um recurso (...) Por outras palavras, qualquer conceito que possa ser alvo de uma referência hipertextual de um autor deve enquadrar-se na definição de recurso." [25]. Todas as operações oferecidas pelo sistema têm como alvo um ou mais recursos, que podem ser criados, modificados ou removidos.

A interação entre cliente e servidor num sistema REST é básica e simples: um cliente faz um pedido, o servidor executa a operação e responde a este pedido. Para definir o resultado de cada operação, são usados os verbos HTTP (*Hypertext Transfer Protocol*) [32]:

- **GET** - Devolve o recurso ou recursos especificados no pedido.
- **POST** - Cria um recurso com os valores descritos no pedido.

- **PUT** - Substitui um recurso totalmente com os valores especificados no pedido.
- **PATCH** - Modifica uma parte de um recurso em específico.
- **DELETE** - Remove o recurso especificado.

Para além do URL, que identifica o recurso, ou recursos, e o verbo HTTP associado ao pedido, a informação adicional que seja requerida pode ser enviada no corpo ou no cabeçalho do pedido, ou até em forma de parâmetro de pesquisa no URL.

2.2 OpenAPI

A *OpenAPI Specification* (OAS) [53] é uma forma padrão de documentar APIs HTTP com a intenção de poderem ser entendidas e usadas por humanos e máquinas. O objetivo é definir como um sistema deve comunicar com a API, indicando que informações são necessárias para cada pedido e que informações serão apresentadas na resposta desse pedido. Estas informações estão expostas num documento OpenAPI, escrito em YAML ou JSON, que contem vários componentes que descrevem o funcionamento da API [76]:

- ***General Information*** - A secção *"info"* do documento inclui o título, a versão e uma descrição da API e, opcionalmente, informação de contacto do desenvolvedor.
- ***Paths* e *Operations*** - A secção *"paths"* inclui os caminhos para as operações oferecidas pela API, fornecendo detalhes de cada operação, como a descrição, os parâmetros necessários e as respostas possíveis.
- ***Components*** - A secção *"components"* define os objetos, ou itens, necessários nas operações descritas no documento.

Para minimizar erros humanos e facilitar a criação do documento OpenAPI, existem ferramentas que geram o documento automaticamente a partir do código-fonte do servidor. Para sistemas desenvolvidos em Java, a combinação do Swagger, para gerar o documento que descreve a API, com o Jakarta RESTful Web Services (JAX-RS), para simplificar o desenvolvimento do serviço RESTful, é a mais comum.

O JAX-RS [75] é uma especificação que fornece um mecanismo para desenvolvimento de serviços *web* que seguem os princípios REST [79]. De uma forma mais concreta, o JAX-RS é uma coleção de interfaces e anotações [2] que ajudam os desenvolvedores a construir o *backend* para seguir os padrões do estilo arquitetural REST, fazendo-o mais simples de programar e de escalar, comparado a outros tipos de sistemas distribuídos [79]. Para usar o JAX-RS existem várias implementações, cada uma com as suas vantagens e desvantagens. Entre as mais conhecidas estão o Jersey [22], o RESTEasy [58] e o Apache CXF [3].

O Swagger [82] é o maior ecossistema de ferramentas usado no desenvolvimento de APIs RESTful que usam a especificação OAS. Existem duas formas de definir uma API usando o Swagger:

- Criando uma API do zero, usando o Swagger Editor [71] para criar a especificação OAS e o Swagger Codegen [69] para gerar a base da implementação do servidor.
- Criando documento OAS a partir de uma API já existente usando o Swagger Core.

O Swagger Core [70] é uma ferramenta que gera o documento OAS a partir da implementação de uma API definida com o JAX-RS, contendo as devidas anotações desta última e da *framework* OpenAPI.

A listagem 2.1 exibe a interface de uma operação do sistema que simula uma loja de animais, *"petstore"* [72], disponibilizado pelo Swagger. As anotações contidas nesta listagem são as que expressam ao Swagger Core tudo o que necessita para gerar um documento OAS completo desta operação. As anotações @GET, @Path, @Produces e @PathParam são anotações da especificação JAX-RS e definem como a operação deve ser invocada e o tipo de resultados que pode produzir. Já o resto das anotações, @Operation, @ApiResponse, e @Parameter, pertencem à OpenAPI e servem para detalhar o documento especificação OAS, especificando cada parâmetro de *input* e detalhando cada resposta. A combinação destas anotações, e de outras não presentes nesta listagem, permite ter uma ideia concreta das possibilidades de utilização e de resultados de cada operação dum sistema RESTful, criando assim uma base bem estruturada para que os desenvolvedores de geradores de testes possam desenvolver as suas ferramentas partindo de uma base confiável.

Listagem 2.1: Código da operação 'getUserByName' do exemplo da petstore.

```
1  @GET
2  Path("/{username}")
3  @Produces({MediaType.APPLICATION_JSON})
4  @Operation(tags = {"user"}, summary = "Get user by user name", operationId = "
       ↪ getUserByName",
5  responses = {
6    @ApiResponse(responseCode = "200", description = "successful operation",
7    content = {@Content(mediaType = MediaType.APPLICATION_JSON,
8                    schema = @Schema(implementation = User.class))}),
9    @ApiResponse(responseCode = "400", description = "Invalid username supplied"),
10   @ApiResponse(responseCode = "404", description = "User not found")})
11 Response getUserByName(@Parameter(name = "username", in = ParameterIn.PATH,
12               required = true, schema = @Schema(implementation = String.class))
13               @PathParam("username") String username);
```

A listagem 2.2 apresenta a parte do documento OAS gerada a partir do código exposto na listagem 2.1 usando o Swagger Core. Neste caso, o documento está escrito em YAML, mas também é aceite em JSON. Este documento é fornecido aos geradores de testes para perceberem o escopo de cada parâmetro nos pedidos e nas respostas e é a base sobre a

qual muitos desenvolvedores constroem filtros mais completos que melhoram a geração de testes.

Listagem 2.2: Exemplo de um documento OAS.

```
paths:
  '/user/{username}':
    get:
      tags:
        - user
      summary: Get user by user name
      description: ''
      operationId: getUserByName
      parameters:
        - name: username
          in: path
          description: 'The name that needs to be fetched. Use user1 for testing. '
          required: true
          schema:
            type: string
      responses:
        '200':
          description: successful operation
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/User'
        '400':
          description: Invalid username supplied
        '404':
          description: User not found
```

Por tanto, é importante perceber a especificação OAS para poder usá-la como base para gerar testes automáticos, para perceber as limitações que pode implicar na eficácia e eficiência dos testes e para expandir a especificação de forma a suprimir testes que piorem a qualidade da avaliação do sistema e conceber novos com valores mais adequados.

3

Trabalho Relacionado

Neste capítulo são apresentadas as duas técnicas usadas para testar e verificar a funcionalidade de um sistema. Para além disto, também são exibidas algumas ferramentas de teste que dependem de um utilizador para o seu funcionamento. Por último, são descritas várias ferramentas que geram os testes automaticamente usando diferentes técnicas.

3.1 Técnicas de teste

3.1.1 White-Box Testing

White-box testing é uma técnica de teste de *software* onde o sujeito que testa o sistema (*tester*), seja um humano ou uma ferramenta de testes automatizada, tem acesso ao código completo do sistema. Ao ter o conhecimento completo da estrutura interna do sistema, esta técnica permite uma análise mais aprofundada das funcionalidades e do comportamento interno do servidor, como os caminhos possíveis de execução de uma operação e as condições que necessitam ser cumpridas para cada caminho. Existem várias ferramentas de teste de *software* usando a técnica *white-box testing*, entre as quais se podem encontrar o JUnit [36], JaCoCo [34] e Selenium [65].

Uma métrica muito usada no *white-box testing* é a cobertura do código, uma vez que se tem acesso ao código-fonte do sistema. Esta visa medir a extensão do código executado durante os testes. Existem três métodos para obter esta métrica [83]:

- ***Statement Coverage*** pretende assegurar que todas as instruções, ou declarações, do código são testadas pelo menos uma vez. Neste procedimento devem ser criados testes com todas as possibilidades de *input* para garantir que todas as linhas do código são executadas. De este modo podemos descobrir zonas de código que não são utilizadas ou até partes do código que estão em falta.

- ***Branch Coverage*** mapeia o código em ramificações de lógica condicional e garante que os testes cobrem cada ramificação. O *tester* deve identificar todas as ramificações, condicionais ou não condicionais, e escrever os testes de forma que todas sejam acedidas e testadas.

- ***Path Coverage*** preocupa-se com caminhos linearmente independentes no código. Neste processo são utilizados diagramas de fluxo de controlo de código e o *tester* deve criar testes que executem todos os caminhos de todos os grafos. O objetivo é encontrar caminhos que sejam redundantes, ineficientes, ou que estejam partidos.

Cabe destacar que esta métrica serve para detetar zonas do código que não foram executadas ou que estão em falta, o que ajuda na produção de testes mais completos, e para detetar alguns *bugs*, mas não garante que os testes tenham boa qualidade ou que o código não contenha falhos [23].

Outras vantagens desta técnica de teste de *software* são: a possibilidade de identificar *bottlenecks* no desempenho, monitorizando métricas como o uso da CPU, da memória e do disco e a *bandwidth* da rede, identificando possíveis zonas do código que geram problemas de desempenho; a facilidade de deteção de zonas de código que possam ser otimizadas, verificando os tempos de resposta de diferentes componentes ou funções do sistema; e a possibilidade de identificar áreas do código que podem não escalar bem sob cargas elevadas e tomar medidas para melhorar a escalabilidade do sistema.

Ainda que o acesso ao código do sistema possa ser uma vantagem na hora de testá-lo, o uso do *white-box testing* não é muito habitual para testar APIs REST devido à granularidade e ao foco em componentes individuais dos testes. Esta abordagem é heterogénea aos sistemas dinâmicos que utilizam a arquitetura REST, que se focam nas interações de diferentes componentes.

3.1.2 Black-Box Testing

Na técnica *black-box testing*, também conhecida por *specification-based testing*, a funcionalidade dum sistema é avaliada sem conhecer os detalhes internos sobre como são realizadas as operações. Para o *tester*, o sistema é tratado como uma caixa negra com *inputs* e *outputs*, logo, informações como a lógica da implementação das operações, o código-fonte e as estruturas de dados utilizadas, entre outras, não são conhecidas. O *tester* concentra-se no que o sistema faz e não em como o faz [29].

Existem várias formas da técnica *black-box testing*, entre as quais se encontram [39]:

- ***Equivalence Class Partitioning*** é uma técnica que organiza os *inputs* do sistema em grupos distintos, denominados classes ou partições, dos quais podem ser gerados casos de teste. Uma classe de equivalência representa um conjunto de estados para as condições de *input* que determinam se um valor é válido ou inválido. Se uma condição de input determina um *range* ou um valor específico, é criada uma classe de equivalência válida e duas inválidas. Se uma condição determina um membro de um conjunto ou um booleano, são criadas duas classes de equivalência, uma válida e uma inválida. Ao seguir estas diretrizes, podem ser criados casos de teste que cubram cada partição pelo menos uma vez, reduzindo a quantidade de testes necessários.

- ***Boundary Value Analysis*** é uma técnica baseada em criar testes no limite do domínio dos valores de *input*, mas só é eficiente para *inputs* de valores fixos, pois é necessário ter conhecimento do escopo dos valores que a variável pode tomar. Um valor limite pode ser um máximo, um mínimo, um valor típico, um valor no extremo, mas dentro do domínio, um valor no extremo, mas fora domínio, ou um valor de erro. O objetivo é criar testes que verifiquem o comportamento do sistema com todas as possibilidades de valores incorretos, e corretos, dos *inputs*.

- ***Fuzz testing***, ou simplesmente fuzzing, é uma técnica que injeta *inputs* inválidos, defeituosos, ou inesperados para descobrir vulnerabilidades no *software* do sistema. O objetivo é observar o comportamento do sistema ao receber estes *inputs* inválidos e identificar problemas de segurança ou desempenho [27]. Há várias formas de fazer *fuzzing*, entre elas estão: *mutation based fuzzing*, que consiste em mutar *inputs* válidos já testados para gerar testes novos que possam levar a erros ou *crashes*, e *generative fuzzing*, que consiste em criar testes com *inputs* gerados de maneira randomizada ou seguindo algum modelo específico.

- ***Cause Effect Graph*** é uma técnica onde é criado um grafo que estabelece a relação entre um efeito e as suas causas. No grafo, os nós são as condições de *input*, que representam as causas, e cada aresta é uma sequência de operações que leva a um efeito. O grafo é, então, convertido numa tabela de decisão que servirá para gerar os casos de teste. Com esta tabela, o *tester* tem conhecimento sobre quais combinações de condições de *inputs* são impossíveis de acontecer, eliminando a geração de testes dispensáveis. Ao contrário das técnicas anteriormente apresentadas, esta considera as combinações de *input* aquando da geração de testes.

- ***Orthogonal Array Testing*** (OAT) é aplicada a problemas com um domínio de *input* pequeno, mas grande até ao ponto que não valha a pena recorrer ao uso da técnica *exhaustive testing*, ou mais conhecida como *brute force testing*. Neste método são usados vetores ortogonais [37], onde cada coluna representa uma variável e cada linha um caso de teste. Desta maneira é possível criar casos de teste que forneçam máxima cobertura de teste, ainda que não garanta a cobertura do domínio de teste.

- ***All-Pair Testing***, tal como o próprio nome indica, pretende testar todos os pares de parâmetros de *input*. Para isto, os testes desenhados, usando este método de *black-box testing*, devem conter todas as combinações possíveis de pares de parâmetros de *input*. Suponha-se que existem três parâmetros A, B e C que podem ter como valores (a1, a2), (b1, b2) e (c1, c2), respetivamente. Cada teste deve conter uma combinação diferente de valores dos três parâmetros (e.g. (a1, b1, c1), (a1, b2, c2), (a2, b1, c1)) para, assim, poder testar todas as possibilidades de *input* e poder descobrir quais causam erros.

- ***State Transition Testing*** analisa o comportamento do sistema ao passar de um estado para outro, logo, é bastante útil para testar *state machines*, onde o *output* depende do estado anterior. Nesta técnica é gerado um diagrama de transição de estados, que contêm quatro componentes principais [31]:
  - O estado em que está o sistema em cada momento.
  - As transições possíveis para cada estado.
  - Os eventos que originam a transição de um estado para outro.
  - As ações resultantes de cada transição.

  Este diagrama é usado para gerar testes que comprovam as transições válidas, ou seja, que não deveriam resultar em erros, enquanto para as transições inválidas é gerada uma tabela em que as linhas correspondem aos estados, as colunas correspondem aos eventos e cada célula representa o estado do sistema após a ocorrência dum determinado evento [31]. Os objetivos deste método de *black-box testing* são: testar o sistema com um conjunto de valores de *input*, testar a dependência dos valores no passado, testar as mudanças nas transições dos estados e testar o desempenho do sistema [56].

Mais recentemente, com o avanço tecnológico nas áreas de inteligência artificial e *machine learning*, várias técnicas de automação inteligente têm sido aplicadas na geração de testes de APIs REST. Um dos grandes avanços nesta área foi a técnica *learning-based testing* [50] (LBT), que adiciona um algoritmo de aprendizagem ao *specification-based testing* para fazer com que a atividade de teste seja mais efetiva e escalável. Um algoritmo LBT começa por executar um conjunto de testes para inferir um modelo, usando os pares de *inputs* e *outputs* dos testes. Se nenhum modelo foi aprendido até ao momento, é criado um vazio e são executados testes, usando alguma técnica do método *specification-based learning*, ou outra à escolha do *tester*, para inferir um modelo base. Se já existir um modelo, o passo anterior não é necessário. A seguir, o algoritmo verifica os requisitos do sistema relativamente ao modelo em busca de um contra exemplo, ou seja, um novo *input* que gere um *output* que não satisfaça os requisitos. Depois da obtenção do *output*, deste novo caso de teste, é verificada a veracidade dos resultados negativos. Se o par (*input*, *output*) não satisfaz os requisitos, o *input* é um negativo verdadeiro e o algoritmo termina, se os satisfaz, é um negativo falso, que é ignorado, pois não encontrou nenhum problema, e o algoritmo começa de novo. Quantos mais testes sejam executados pelo algoritmo LBT melhor será o modelo e maior será a probabilidade de encontrar erros de implementação do sistema em teste.

Para que estas técnicas de *black-box testing* tenham o melhor desempenho possível, é necessário criar uma boa especificação do sistema. Quantos mais detalhes a especificação oferecer, mais problemas no sistema serão encontrados. Portanto, os *testers* têm que estudar bem o sistema e as suas operações para saber que técnica usar, para obter maior

possibilidade de encontrar erros, e como desenvolver umas especificações completas, para simplificar e acelerar a geração de testes úteis.

No JepREST decidimos usar a abordagem do *black-box testing* com a intenção de que qualquer pessoa em qualquer lugar possa testar qualquer API REST.

3.2 Ferramentas de teste

Nesta secção são apresentadas algumas ferramentas de teste de sistemas distribuídos. Umas dependem mais da interação com o utilizador do que outras, mas todas necessitam da colaboração do utilizador para construir o *input*. Estas ferramentas estão preparadas para testar APIs REST, ou podem ser modificadas para tal função.

3.2.1 cURL

O cURL [13] é um projeto *open-source* que oferece uma ferramenta de linha de comandos e uma biblioteca para transferir dados na internet através de múltiplos protocolos, como podem ser o HTTP, HTTPS ou FTP. Tanto a ferramenta, como a biblioteca, permitem ao utilizador fazer pedidos a um servidor mediante o URL do mesmo sem ter de se preocupar com este processo. Por sua vez, o cURL não tem qualquer noção dos dados que são enviados e recebidos, simplesmente trata de saber utilizar o protocolo escolhido pelo utilizador para a transferência de dados [15].

Por ser uma ferramenta bastante útil e portátil (i.e., funciona em diversas plataformas), é muito comum os desenvolvedores a utilizarem para fazer testes unitários a um *endpoint* de uma API REST, a fim de verificar se está a funcionar e se a operação em causa devolve o resultado esperado, podendo também aceitar *scripts* para automatizar os pedidos que o utilizador queira fazer. Para cada pedido à API é necessário definir o URL do recurso, o método HTTP associado (e.g., GET, POST), os *headers* indicados e o *body*, no caso de o método ser POST, PUT ou PATCH [14].

O uso do cURL ajuda os desenvolvedores na fase da construção de um sistema, pois podem ir testando individualmente o resultado de cada operação, mas também tem as suas limitações. Ao ser uma ferramenta manual, poucos casos de teste são verificados, então, apesar de poder demonstrar que a operação funciona, não assegura que está isenta de erros. Também não são testados casos de concorrência. Logo, não serve para testar um sistema por completo, pois seria um processo demasiado complexo e demorado e não daria nenhuma garantia sobre a confiabilidade do mesmo.

3.2.2 Postman

O Postman [55] é uma plataforma que simplifica o processo de construção, teste e uso de APIs, fornecendo um amplio conjunto de ferramentas que ajudam os utilizadores a elaborar APIs REST eficientes e bem desenhadas. Durante o processo de criação de uma API, o Postman disponibiliza informações relevantes sobre todas as operações, como

warnings, métricas ou relatórios, para ajudar a descobrir e resolver possíveis erros. Esta ferramenta também fornece assistência na fase de organização e colaboração com outros utilizadores [54].

O Postman é bastante usado para testar APIs REST manualmente mas também aceita *scripts* para automatizar a fase de teste. Estes *scripts* têm de ser escritos em JavaScript e podem ser executados em duas fases: antes do pedido, para, por exemplo, verificar os parâmetros da operação; e depois da resposta, para, por exemplo, ver se o resultado da operação é o esperado [64]. No Postman também é possível correr vários testes sequencialmente com a ferramenta *Collection Runner*. Ao usar esta ferramenta o resultado de cada operação é guardado num *log* e podem ser usados *scripts* para verificar cada teste, parando a execução na presença de um erro [74].

Ao contrário do cURL, o Postman tem uma interface gráfica (GUI). Esta GUI é bastante intuitiva e facilita o uso da ferramenta, afastando a necessidade de ler uma documentação para saber usá-la, como acontece no cURL. Esta é uma grande vantagem, pois os desenvolvedores podem focar-se só na construção da API sem perder tempo com coisas externas ao sistema que estão a desenvolver.

Apesar disto, o Postman tem o mesmo problema que o cURL, no sentido que ambas as ferramentas não garantem a confiabilidade do sistema, servindo apenas para fazer um teste unitário ou um conjunto limitado de testes definidos pelo *tester*.

3.2.3 REST-assured

O REST-assured [57] é uma biblioteca em Java usada para testar e validar APIs REST. A biblioteca oferece vários métodos e funcionalidades para executar pedidos a uma API e fazer verificações dos dados que vêm na resposta. Todos os campos da resposta (e.g., *headers*, *body*, *path param*) podem ser acedidos e podem ser efetuadas asserções sobre cada um destes. O REST-assured pode ser utilizado em combinação com *frameworks* de teste como o JUnit e o TestNG [17].

A sintaxe do REST-assured é bastante simples de compreender, pois segue a técnica *Behavior Driven Development* (BDD), usando partes da linguagem natural para simplificar a compreensão do código. Isto significa que qualquer programador perante uma operação de teste conseguiria traduzi-la facilmente para a linguagem natural, sem necessidade de ler nenhuma documentação nem fazer nenhum estudo. A sintaxe base do REST-assured é a seguinte [30]:

- ***Given()***. Define o contexto da operação. Aqui são passados os parâmetros necessários para executar o pedido, como os *headers*, o *body*, as *cookies*, etc.
- ***When()***. Determina a premissa do cenário. Aqui é definida a operação que vai ser enviada ao sistema REST, usando os parâmetros definidos no *Given()*.
- ***Then()***. Indica as verificações que serão feitas quando for recebida a resposta do pedido efetuado no *When()*.

O REST-assured, comparativamente com o cURL e o Postman, é uma ferramenta mais flexível por ser uma biblioteca e por estar preparada para ser usada juntamente com frameworks de teste, permitindo executar testes mais complexos. Porém, à semelhança do cURL e do Postman, o REST-assured é uma ferramenta direcionada à execução de testes unitários.

3.2.4 Artillery

O Artillery [6] é uma ferramenta que permite fazer testes de carga em aplicações *cloud*. A maior diferença entre esta ferramenta e as ferramentas descritas anteriormente é, precisamente, a possibilidade de fazer testes de carga. Um teste de carga é um tipo de teste de sistemas distribuídos onde se executam operações concorrentes, simulando o tráfego do mundo real, onde vários clientes interagem com o sistema ao mesmo tempo. No entanto, apesar de não ser a funcionalidade principal, o Artillery também oferece a possibilidade de fazer testes funcionais para verificar singularmente o resultado de cada operação [77].

Os testes no Artillery são definidos em *scripts* que descrevem como vão ser executados. Estes *scripts* indicam informações como o URL do sistema em teste, quanta carga será gerada e os *timeouts* das respostas, e, também estabelecem cenários onde são estipulados os passos que vão ser executados pelos clientes virtuais, podendo determinar a frequência com que cada cenário será executado. Geralmente, os *scripts* de teste são definidos num ficheiro YAML, mas também podem ser escritos em JavaScript [62].

Esta ferramenta é mais utilizada para avaliar o desempenho e escalabilidade do sistema em teste. Porém, um dos maiores inconvenientes desta ferramenta é que não suporta injeção de falhas, não podendo assim testar exatamente as qualidades do sistema numa situação de vulnerabilidade. Por estas razões, o Artillery é uma boa ferramenta para testar o desempenho na fase de teste e auxiliar o desenvolvimento dum sistema, mas não para garantir a confiabilidade do mesmo uma vez publicado.

3.2.5 Jepsen

O Jepsen [35] é uma biblioteca de software desenvolvida em Clojure para testar sistemas distribuídos com o objetivo de aumentar a segurança destes. Esta ferramenta é utilizada para comprovar se o estado de um sistema cumpre as propriedades definidas num modelo após várias execuções.

Um teste é um programa que usa a biblioteca Jepsen para executar várias operações no sistema e verificar se a historia, ou seja, a sucessão de respostas, é a esperada. A Figura 3.1 mostra o processo de teste do Jepsen. Neste caso é usado o Docker para criar um *cluster* com seis *containers* que simulam a interação entre vários clientes e uma base de dados distribuída. Dos seis *containers*, um é o nó de controlo e os cinco restantes são os nós da base de dados. O nó de controlo cria um grupo de *worker threads* que funcionam como clientes que interagem concorrentemente com a base de dados através do protocolo SSH.

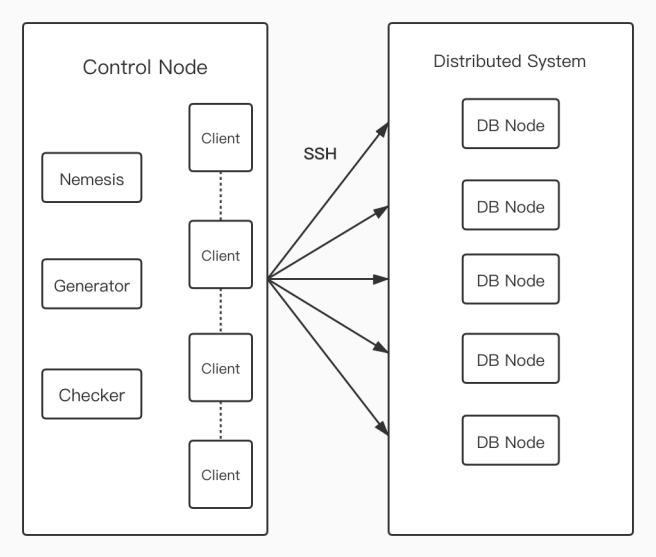


Figura 3.1: Processo de teste do Jepsen.

Dentro do nó de controlo, para além dos clientes, também existem três outros operadores: o *Generator*, que é o responsável por informar os clientes sobre as operações que têm de submeter à base de dados; o *Nemesis*, que é o encarregado de indicar o conjunto de falhas que cada cliente deve injetar no sistema; e o *Checker*, que analisa a precisão da história gerada com os resultados das operações submetidas pelos clientes para verificar se o modelo de consistência definido pelo sistema é cumprido [12]. O *checker* principal do Jepsen é o Knossos [43], que verifica se a história é linearizável, mas pode ser definido um novo, dependendo das necessidades do sistema, como o Elle [42], para analisar se existem problemas de coerência nas transações, ou o HISTEX [44], que pode ser usado para descobrir se as histórias geradas podiam ter sido produzidas com o nível de isolamento definido na base de dados.

Cada teste gera uma história [10] onde fica registada toda a informação relativa às operações executadas. Cada operação regista o *id* da *worker thread*, uma *tag* que indica se foi invocada ou concluída, que tipo de operação foi realizada (e.g., *read*, *write*), o *timestamp* da sua execução e, opcionalmente, alguns detalhes referentes à mesma.

O JepREST usa o Jepsen como base para verificar a história de um sistema RESTful. Para isto, foi necessário estender a biblioteca de forma a criar clientes que façam pedidos HTTP à API REST do sistema.

3.3 Ferramentas de geração automática de testes de APIs REST

Apesar de existirem ferramentas manuais de teste bastante úteis e efetivas, os resultados do seu uso são limitados pelo *input* de quem as esteja a usar. Isto deve-se a que uma pessoa, ou um grupo limitado de pessoas, não tem a capacidade de descobrir todas as vulnerabilidades que um sistema possa ter, ademais de ser um processo demoroso e

custoso. Para atacar este problema apareceram as ferramentas de geração automática de testes, que usam várias técnicas diferentes, desde *randomizers* até *Large Language Models* (LLMs). A seguir estão apresentadas algumas destas ferramentas.

3.3.1 EvoMaster

A EvoMaster [4] é uma ferramenta de geração automática de testes de integração para sistemas RESTful que utiliza a técnica *white-box testing* e só pode ser usada em APIs compiladas em JVM (e.g. Java ou Kotlin) [24]. Os seus principais objetivos são encontrar erros através dos estados HTTP das respostas dos pedidos e maximizar a cobertura do código. Por esta razão é que os autores decidiram seguir a abordagem *white-box*, assim podem ter conhecimento das zonas do código que não chegam a ser acedidas e usar esta informação para gerar novos testes.

Nesta ferramenta os valores dos parâmetros são inferidos mediante um algoritmo de pesquisa chamado Genetic Algorithm (GA). O GA é inspirado na teoria da evolução de Charles Darwin, onde os indivíduos mais aptos à sobrevivência se reproduzem entre si. Neste caso, os indivíduos mais aptos à sobrevivência são os valores com maior valor de fitness. Estes passam por um processo de reprodução, onde são misturadas partes de ambos e feitas pequenas mutações, para gerar novos valores com maior fitness, tendo a intenção de encontrar valores ótimos para os testes. Em particular, na EvoMaster, um indivíduo é um conjunto de casos de teste, inicializados com valores randomizados, com tamanho e comprimento das variáveis explícito. O fitness do conjunto de casos de teste é a soma do fitness de todos os casos de teste. Na fase de reprodução, os valores de dois testes são misturados e, no fim, modificados ligeiramente. Quando este novo teste é executado, se cobrir um alvo novo (e.g. uma zona do código não acedida por nenhum outro teste), é guardado para não se perder em possíveis futuras gerações. No fim da pesquisa, o conjunto de testes mantidos à parte é escrutinado, tirando os testes redundantes, e guardado no disco como um ficheiro de teste.

A EvoMaster usa os *status codes* das respostas como testing oracle. Os erros, ou *bugs*, são detetados quando o código do estado de uma resposta é da ordem dos 500. Quanto à parte da cobertura do código, a EvoMaster oferece uma biblioteca com funcionalidades para instrumentar o código do sistema. Esta biblioteca expõe uma API REST que oferece informações quanto à cobertura do código e comprimento do ramo da árvore de mutações genéticas do GA.

Segundo as experiências relatadas, os conjuntos de teste gerados conseguem obter entre 18% e 41% de cobertura do código, o que é bastante baixo, e as limitações das *Strings*, as bases de dados e os serviços externos ao sistema são os principais obstáculos. Não obstante, existem algumas extensões da técnica que melhoram a eficácia dos testes, contando com o estado da base de dados [5] e melhorando o uso de recursos e dependências [84, 40].

3.3.2 Geração automática de testes usando a especificação OAS

No trabalho feito por Ed-douibi, Canovas Izquierdo e Cabot [18] é apresentada uma solução para gerar casos de teste usando unicamente a especificação OAS do sistema em teste. A ideia passa por construir uma ferramenta que, através da especificação dum sistema RESTful, consiga gerar suficientes casos de teste para avaliar o funcionamento de cada operação da API. São usados dois tipos de teste de APIs: o *Specification-based API testing*, para verificar se as respostas do servidor condizem com as listadas na especificação, e o *Fault-based testing*, para provar que os possíveis erros já conhecidos não existem.

Para isto, a abordagem proposta segue quatro passos, tal como mostrado na Figura 3.2:

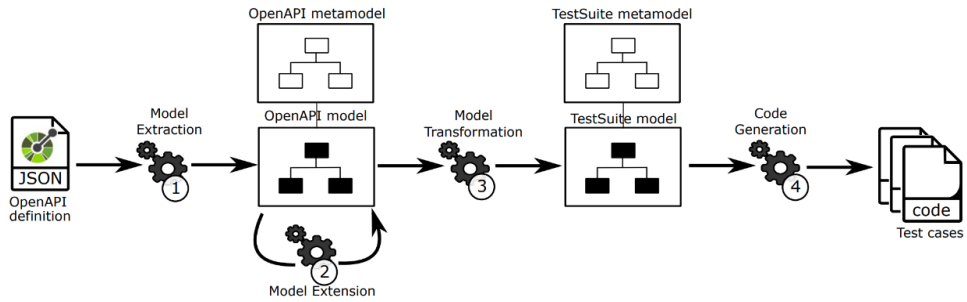


Figura 3.2: Gráfico dos passos da geração de testes usando a especificação OAS

Model Extraction. Neste primeiro passo, a ferramenta transforma o JSON, ou YAML, da especificação OAS do sistema num modelo OpenAPI, usando o meta modelo OpenAPI [19] apresentado pelos mesmos autores. Este é um processo bastante simples e direto onde cada campo do ficheiro JSON é transformado no elemento do modelo correspondente.

Model Extension. Este é o segundo passo da abordagem. O objetivo é dar ao modelo informação sobre os valores dos parâmetros para poderem ser inferidos aplicando três regras sequencialmente:

1. O valor de um parâmetro pode ser inferido através de exemplos, valores *default* ou enumerações.
2. O valor de um parâmetro pode ser um valor fictício se uma resposta a um pedido com esse valor foi bem sucedida.
3. O valor de um parâmetro pode ser inferido da resposta de uma operação se a resposta for bem sucedida e contiver o parâmetro em causa.

Model Transformation. No terceiro passo é onde os casos de teste são definidos. Para este propósito foram definidas duas regras de geração de teste.

A primeira regra serve para gerar casos de teste que retornem uma resposta de sucesso (i.e., HTTP *status-code* da família dos 2xx). Para cada operação vai ser gerado

um caso de teste que inclua todos os parâmetros necessários com os respetivos valores, inferidos através da informação detalhada no modelo definido no passo anterior.

A segunda regra é usada para elaborar casos de teste defeituosos, com valores de *input* incorretos, que devolvam uma resposta de erro (i.e., HTTP *status-code* da família dos 4xx). Para cada parâmetro de uma operação são gerados casos de teste que avaliem três cenários:

- Se o parâmetro for obrigatório, não o incluir.
- Se o parâmetro não for uma *String*, atribuir um valor errado (e.g., para um inteiro atribuir uma *String* como valor).
- Se o parâmetro tive uma limitação, atribuir um valor que quebre essa limitação (e.g., para um inteiro com uma limitação de máximo dar um valor superior ao máximo).

Code Generation. Neste último passo é feita a geração do código usando as definições de cada caso de teste construídas nos passos anteriores. Qualquer linguagem ou ferramenta de teste pode utilizar estas definições para gerar o código de teste. No caso dos autores, usaram a linguagem Java e geraram testes JUnit.

A abordagem descrita neste *paper* não tem em conta as possíveis dependências entre operações. Isto impossibilita desfazer os efeitos das operações após efetuadas, podendo afetar o resultado dos testes. A falta de informação necessária para operações mais complexas (e.g., operações que não seguem a semântica REST, operações que dependem de outras) na especificação OAS e a pobre definição desta, afetam a cobertura dos testes, especialmente dos testes defeituosos.

3.3.3 RestTestGen

O RestTestGen é tanto uma ferramenta [78] como uma *framework* [11] *black-box* de geração automática de casos de teste para APIs REST. Esta ferramenta inclui três módulos que são executados em sequência para gerar casos de teste nominais, ou seja, com respostas de sucesso (i.e., HTTP *status code* 2xx); e casos de teste de erro (i.e., HTTP *status code* 4xx). O único *input* que o RestTestGen requer é a especificação OAS do sistema em teste.

A informação descrita na documentação é fornecida ao primeiro módulo, onde é gerado um grafo de dependências entre operações. Uma operação depende de outra quando o *output* da primeira tem um campo em comum com o *input* da segunda. Isto pode causar problemas quando o nome dum campo é diferente em duas operações dependentes. Esta ferramenta tem isso em consideração e implementa um algoritmo que prevê este tipo de casos e aplica algumas alterações aos nomes dos campos para que sejam iguais entre operações dependentes.

A seguir, o grafo gerado e a especificação OAS são inseridos como *input* no segundo módulo, onde são gerados os casos de teste nominais. Primeiro, para definir a ordem das operações são usados o grafo de dependências e a semântica CRUD (*Create, Read, Update, Delete*). O algoritmo percorre o grafo do fim ao início, ordenando as operações segundo o seu efeito no sistema, por exemplo, uma operação *create* tem de acontecer antes de um *delete*. Uma vez ordenadas as operações, é necessário inferir os valores de *input* dos pedidos. Para isto é usado um mapa onde a chave é o nome dos campos e o valor é uma coleção de valores que o campo já tomou em pedidos anteriores. No caso deste método não ser apropriado (e.g., não existir nenhuma entrada no mapa para o campo em causa) ou falhar, os valores são induzidos através de valores *default*, de exemplo, de enumeradores ou são gerados de forma aleatória. O último passo é a validação da resposta do caso de teste. Tendo em conta o *status code* da resposta, se o valor for 4xx ou 5xx, o caso de teste é inválido e é ignorado; se o valor for 2xx significa que o pedido foi executado com sucesso e se a resposta estiver de acordo com a especificação OAS, o pedido é válido, caso contrário é considerado inválido e é ignorado.

Por último, os casos de teste nominais gerados no módulo anterior são introduzidos no terceiro, e último, módulo, onde são gerados os casos de teste de erro. Para isto, são efetuados três tipos de mutações aos *inputs* dos testes nominais. A primeira consiste em remover um campo de *input* obrigatório de um pedido alternadamente. A segunda corresponde a introduzir valores de *input* errados. E a terceira foca-se em violar as restrições dos parâmetros do pedido. Estas mutações são aplicadas no mínimo uma vez por cada caso de teste nominal com o objetivo de encontrar o maior número de vulnerabilidades possível, conseguindo saber o que as provoca. A fase de avaliação dos casos de teste de erro é parecida à do módulo de testes nominais, mas, neste caso, as respostas com *status code* 2xx são consideradas inválidas, as de 5xx válidas e as de 4xx válidas sempre e quando satisfaçam o esquema apresentado na especificação.

Os resultados empíricos relatados por Viglianisi, Dallago, e Ceccato [78] demonstram que o RestTestGen é uma ferramenta muito efetiva, sendo capaz de encontrar bastantes erros em APIs REST reais. Esta ferramenta destaca-se da abordagem anterior [18] justamente por ter em conta as dependências entre operações, embora ambas as abordagens negligenciem as dependências entre os parâmetros de cada operação.

3.3.4 Inter-parameter Dependency Language (IDL)

Muitas APIs RESTful oferecem operações onde o uso dos parâmetros não é trivial, podendo existir dependências entre estes. No entanto, a especificação OAS não tem como definir tais dependências. Para resolver este problema, Martin-Lopez, Segura e Ruiz-Cortés [48] propõem uma *domain-specific language* chamada Inter-parameter Dependency Language (IDL) que descreve as várias dependências entre parâmetros.

Dado um conjunto de parâmetros, os tipos de dependências que podem existir entre eles são:

1. **Requires**. A presença de um parâmetro requer a existência de outro.

2. **Or**. Pelo menos um dos parâmetros tem de estar presente no pedido.

3. **OnlyOne**. Um, e só um deles deve ser incluído no pedido.

4. **ZeroOrOne**. No máximo um dos parâmetros pode estar no pedido.

5. **AllOrNone**. Ou todos os parâmetros estão presentes, ou nenhum o pode estar.

6. **Arithmetic/Relational**. Esta dependência utiliza operações aritméticas e/ou relacionais para definir os valores válidos de um parâmetro em relação a outro/os.

7. **Complex**. Esta dependência aparece como uma combinação de duas ou mais dependências dos tipos descritos acima.

A Listagem 3.1 mostra como é definida cada dependência na IDL. Cada linha é referente à dependência da lista acima seguindo a mesma ordem.

Listagem 3.1: Exemplo da utilização da IDL.

```
1  IF videoDimension THEN type=='video';
2  Or(title, description);
3  OnlyOne(From, MessagingServiceSid);
4  ZeroOrOne(radius, rankby=='distance');
5  AllOrNone(subject\_id, subject\_type);
6  owner.percentage + owner2.percentage + owner3.percentage <= 100;
7  type=video -> OnlyOne(embed, data);
```

Para incluir na especificação as dependências entre os parâmetros de uma operação é usada uma extensão definida como "x-dependencies". Na Listagem 3.2 podemos ver um exemplo meramente figurativo onde está especificada uma operação GET. Na linha 8, a dependência $Or(p1, p2, p3)$ determina que pelo menos um dos três parâmetros tem de ser usado. Na linha 9, a dependência IF p_1 == true THEN p_2 diz que se o parâmetro p_1 for verdadeiro, então o p_2 deve estar incluído no pedido.

Listagem 3.2: Exemplo da utilização da IDL para definir as dependências entre os parâmetros de uma operação na especificação OAS.

```
1  paths:
2    /example/route:
3      get:
4        operationId: exampleOperation
5        parameters:
6          ...
7        x-dependencies:
8          - Or(p1, p2, p3);
9          - IF p1==true THEN p2;
```

O uso correto das dependências entre parâmetros na definição da especificação OAS é uma grande vantagem para os geradores de testes. Com estas informações, acerca da relação entre os diferentes parâmetros de cada operação, é muito mais provável que os casos de teste elaborados tenham valores mais apropriados para cada parâmetro. Para além disto, também serão descartados os testes com combinações de parâmetros erradas. Estes benefícios permitem que mais partes do sistema sejam avaliadas, levando assim a uma maior cobertura do sistema e a uma maior probabilidade de encontrar erros.

3.3.5 Automated IDL

Apesar da sua efetividade, a utilização da IDL requer que sejam escritas as dependências na especificação OAS, tarefa esta sujeita a erro humano. Para minimizar a probabilidade de erro, o desenvolvedor deve conhecer bem a linguagem IDL e ter noção de todas as dependências de cada operação do sistema em causa. Mesmo assim, ocorrem erros frequentemente ao definir APIs mais extensas. Para além disto, muitas especificações de sistemas reais não expõem as dependências entre os parâmetros nem tem qualquer informação em linguagem natural sobre estas.

Por estas razões, Mirabella et al. [51] apresentam uma abordagem baseada em *deep learning* para inferir se um pedido a uma API RESTful satisfaz todas as dependências da IDL. É proposta uma rede neuronal artificial (RNA), mais especificamente, um multilayer perceptron, que conclui automaticamente se um pedido a uma API é válido ou não sem chegar a executá-lo, resultando numa redução do número de interações com o servidor e num aumento na rapidez da geração de testes. O *dataset* que é dado como input à RNA na fase de treino é composto por vários pedidos à API devidamente identificados como válidos (i.e., HTTP *status-code* 2xx), ou inválidos (i.e., HTTP *status-code* 4xx). Todos estes pedidos devem estar corretamente formulados seguindo a especificação OAS para assegurar que qualquer pedido assinalado como inválido viola uma ou mais dependências entre parâmetros. Este *dataset* pode ser retirado da atividade real dos utilizadores do sistema ou usando uma ferramenta de geração de casos de teste como o RESTest [49]. Uma vez acabada a fase de treino da RNA com o *dataset*, basta introduzir um novo pedido à API como *input* para determinar a validade do mesmo.

Com esta ferramenta, a geração de testes válidos passa a ser mais eficiente e económica e resolve possíveis problemas causados por erros na definição das dependências entre parâmetros ou pela ausência desta.

3.3.6 Exploração de Comportamentos de APIs Através de Exemplos Gerados

A abordagem apresentada por Karlsson et al. [38] propõe a exploração do comportamento de APIs através da geração automática de exemplos de interações entre as suas operações, sem depender do código fonte ou de especificações formais. A base da abordagem reside nas "meta-propriedades", que descrevem comportamentos gerais esperados nas APIs, tais como alterações de estado e respostas consistentes a operações repetidas.

O processo começa por abstrair a API numa especificação genérica designada por *Abstract Method and Operations Specification* (AMOS). Esta especificação lista as operações disponíveis e os seus parâmetros, sem detalhes de implementação, tornando a ferramenta aplicável a diferentes tipos de APIs, como REST e GraphQL. A AMOS pode ser gerada automaticamente a partir de especificações existentes (e.g., OpenAPI) ou criada manualmente.

Com a AMOS e as meta-propriedades definidas, a ferramenta gera sequências de chamadas à API para testar o comportamento. A geração inclui três tipos de operações: operações com parâmetros criados de forma aleatória, operações que utilizam valores de parâmetros previamente definidos e operações derivadas das respostas de operações realizadas anteriormente. As operações são traduzidas de um formato abstrato para chamadas concretas à API.

Os exemplos gerados são avaliados com base em meta-propriedades. Por exemplo, uma meta-propriedade pode verificar se uma sequência de operações resulta em alterações de estado detetáveis ou se uma operação retorna consistentemente os mesmos resultados quando repetida. As sequências são refinadas iterativamente, removendo operações e parâmetros redundantes, até se obterem exemplos mínimos que ainda demonstram o comportamento relevante. Estes exemplos fornecem informações que ajudam o utilizador a compreender a API e a iterar na exploração, ajustando parâmetros ou meta-propriedades para investigar novos comportamentos.

As aplicações desta ferramenta incluem a facilitação da documentação de APIs, a verificação da conformidade com as especificações e a identificação de comportamentos inesperados em sistemas complexos.

Esta abordagem destaca-se por não exigir informações internas do sistema e por gerar exemplos significativos e concisos, complementando as práticas de teste e documentação de APIs. Embora seja eficaz para produzir exemplos de comportamentos em aplicações REST que utilizam a semântica da arquitetura, não é adequada para testar aplicações com semântica personalizada, devido à dificuldade, ou até impossibilidade, de generalizar as operações. Além disso, a abordagem só permite gerar exemplos sequenciais, não sendo possível introduzir concorrência.

3.3.7 TestPilot

O TestPilot [63] é uma ferramenta de geração de testes unitários em JavaScript usando um LLM. Na abordagem apresentada pelos autores, o objetivo principal é testar *packages* em JavaScript usando a *framework* de teste Mocha [52]. O conceito principal é dar a parte inicial do código ao LLM para que este gere o resto do código de teste. O *input* do LLM é a chamada das funções `describe` e `it` da *framework*, como apresentado na Listagem 3.3, e todo o código que venha a seguir a este é gerado pelo LLM. Para além disto, também podem ser incluídos comentários no *input* para ajudar o LLM a criar testes corretos.

Listagem 3.3: Exemplo de *input* do LLM usando a *framework* Mocha no TestPilot.

```
let countries_and_timezones = require('countries-and-timezones');
// countries-and-timezones.getCountry(id)
describe('test countries_and_timezones', function() {
  it('test countries-and-timezones.getCountry', function(done) {
```

A arquitetura do TestPilot está formada por cinco componentes. O primeiro é o *API Explorer*, que analisa o *package* em teste para determinar a lista de funções que podem ser usadas. O segundo é o *Documentation Miner*, que tem como objetivo extrair pedaços de código e comentários incluídos no *package* e associá-los a uma função da API. Os três próximos componentes funcionam em conjunto para gerar e validar testes para todas as funções identificadas na API usando a informação dada pelo *Documentation Miner*. As funções são processadas uma de cada vez e para cada uma só é gerado um teste, em vez de um conjunto de testes. O *Prompt Generator* gera o *input* inicial do LLM. Uma vez gerado o código de teste pelo LLM, o *Test Validator* verifica a sintaxe e, se estiver correta, é executado o teste usando o Mocha para determinar se a função passa o teste. Se o teste estiver correto a execução do TestPilot acaba aqui, mas se houver algum problema, quer de sintaxe, quer de execução, o teste é enviado para o *Prompt Refiner* para construir um *input* com mais informação nos comentários, como por exemplo, o teste que falhou e a mensagem de erro correspondente para que o LLM possa aprender dos erros cometidos.

Apesar do TestPilot estar feito para testar packages em JavaScript, pode ser usado como referência para criar um gerador de testes de APIs REST. Para este caso, não seriam necessários os dois primeiros componentes, *API Explorer* e *Documentation Miner*, já que a informação está toda disponível na especificação OAS da API do sistema em teste. Também seria fundamental usar outra *framework* de teste mais adequada para executar pedidos HTTP. Por último, em vez de pedir um teste por cada função ao LLM, seria vantajoso pedir que sejam gerados vários para cobrir mais funcionalidades e aumentar a possibilidade de encontrar erros.

3.3.8 RestGPT

O RestGPT [67] é uma ferramenta que utiliza LLMs para controlar aplicações RESTful através das suas APIs. A ideia principal é que um utilizador que queira executar uma operação não necessite saber como, e simplesmente tenha de descrever a operação que pretende executar. Por exemplo, se um utilizador do Spotify quer adicionar uma música a uma playlist, em vez de o fazer manualmente, basta dizer ao RestGPT "Adiciona a música X do autor Y à minha playlist." e essa música será adicionada. Os únicos *inputs* desta ferramenta são a instrução do utilizador e a especificação OAS das APIs necessárias.

O RestGPT é composto por três módulos principais: um *Planner*, um *API Selector* e um *Executor*. O *planner* decompõe cada instrução do utilizador em várias subtarefas. O *API selector* lê as descrições dos endpoints de todas as APIs e escolhe a API adequada para resolver cada subtarefa. O *executor* realiza pedidos à API e extrai informação útil da

resposta para formar o resultado da execução. Este último módulo está formado por um *Caller* e um *Parser* de respostas. O *caller* usa a informação de cada operação detalhada na especificação da API para gerar os parâmetros e o corpo do pedido corretos. O *parser* utiliza o esquema da resposta especificado para gerar o código de análise e extrair informação da resposta. O núcleo de cada componente é um modelo LLM que é iniciado com um pequeno texto de *input* em linguagem natural que descreve a sua funcionalidade. Neste texto são dadas informações como o objetivo do componente, a estrutura do *output* e outras várias instruções para ajudar o LLM a saber o que fazer.

O fluxo normal do RestGPT é o seguinte. Primeiro, o utilizador descreve a operação que deseja ser executada. De seguida, o *planner* gera um plano em linguagem natural, dividido em subtarefas. Para cada tarefa, o *API selector* decide qual é a API certa. Então, o plano com as APIs já definidas é enviado ao *executor* para executar a operação. Uma vez executadas todas as subtarefas do plano, o *planner* decide se o resultado do plano é o correto ou não. Caso o resultado esteja correto é devolvida a resposta ao utilizador. Caso contrário, a execução do plano errado é fornecida ao LLM do *planner* e é gerado um novo plano.

Esta ferramenta tem como objetivo simplificar o uso de APIs REST, mas também pode ser usada para testá-las. Para isto basta mudar o texto de *input* dos LLMs dando indicações para procurarem resultados que infrinjam as condições impostas na especificação.

3.3.9 Outras Ferramentas

A utilização de LLMs em ferramentas de teste de APIs REST é uma abordagem pouco estudada. Ainda assim, pode-se observar como são implementados estes modelos em outros tipos de ferramentas e aproveitar o trabalho já feito para tirar conclusões sobre a sua utilidade e eficácia, podendo assim construir uma solução em base a descobertas fundamentadas. Entre estas ferramentas podemos encontrar o AlphaCodium [60], que tem como principal objetivo gerar código para resolver um problema especificado em linguagem natural; o PentestGPT [16], que realiza testes de penetração automáticos usando um LLM; ou o PROXYQA [73], que é uma *framework* desenhada para avaliar textos compridos gerados por um LLM.

Para a realização deste trabalho, investigámos o uso de LLMs para gerar sequências de operações relevantes, mas decidimos usar uma abordagem distinta, apresentada mais à frente no relatório.

4

Desenho da Solução

O JepREST [1] [66] é uma ferramenta *black-box* que tem como propósito testar a funcionalidade de sistemas REST. Esta ferramenta estende a biblioteca Jepsen, descrita na secção 3.2.5, de modo a que seja possível fazer pedidos HTTP a qualquer servidor na *web* e verificar o resultado dos mesmos. O JepREST tem a possibilidade de realizar testes funcionais com e sem concorrência, para além de também poder injetar falhas no sistema em teste. A verificação da correção do sistema é feita analisando os resultados das operações para comprovar que tenham um comportamento linearizável, utilizando o *checker* Knossos [43]. Para poder fazer esta verificação, o estado do sistema em teste é guardado num mapa de recursos, representados em JSON, que é atualizado com cada operação de sucesso, mantendo localmente o estado correto do servidor, segundo o modelo definido.

A versão incial do JepREST [66] consegue testar a correção de aplicações que usam a semântica definida pela arquitetura REST. Além disto, também permite definir manualmente os pedidos que os clientes do JepREST devem realizar em cada momento.

Neste trabalho, expandimos o JepREST para poder testar sistemas REST mais complexos, onde a semântica da arquitetura não é adotada devido às limitações inerentes à sua aplicação. Para especificar o comportamento das operações destes sistemas, definimos uma linguagem, apresentada na secção 4.2, que permite descrever as ações internas das operações. Também criámos um novo gerador de operações do Jepsen [41], que permite produzir automaticamente sequências de operações semelhantes às executadas pelos clientes reais da aplicação, utilizando um grafo de transições de operações gerado a partir dos *logs* do sistema. Este grafo é criado pela ferramenta descrita na secção 4.3.

Este capítulo começa por uma apresentação geral do JepREST na secção 4.1. Uma parte significativa da estrutura base foi desenvolvida em [66], mas a sua apresentação é importante para o trabalho realizado. Nesta descrição indicaremos as partes que foram modificadas em relação à versão inicial.

[1] https://github.com/preguica/JepREST

4.1 JepREST

4.1.1 Visão Geral

O processo de teste de uma aplicação utilizando o JepREST é composto por várias etapas sequenciais que, juntas, garantem uma verificação abrangente da API REST sob teste. A seguir, descrevem-se os principais passos envolvidos:

1. **Definição da API**: O utilizador começa por fornecer a API REST que será testada. Esta API serve como a base sobre a qual todo o processo de teste será construído. A mesma deve estar especificada usando as anotações do OpenAPI, do JAX-RS e, quando aplicável, da linguagem introduzida neste trabalho, apresentada na secção 4.2.

2. **Geração de *Workload***: Em seguida, o utilizador indica como devem ser geradas as operações invocadas pelos clientes do JepREST. Existem duas maneiras de o fazer. A primeira consiste em definir um ficheiro YAML tipo Artillery que determina vários cenários de execução com sequências de operações predefinidas. A segunda consiste em utilizar a ferramenta descrita na secção 4.3 para gerar um grafo de transições de operações a partir dos *logs* da aplicação em teste.

3. **Sistema que Executa os Pedidos**: Com o acesso à API, o JepREST gera um conjunto de métodos responsáveis por efetuar os pedidos à aplicação, que serão utilizados pelos clientes durante a fase de teste.

4. **Execução dos Testes e Recolha de Resultados**: Durante esta fase, o JepREST utiliza o ficheiro YAML ou o grafo de transições de operações para determinar as operações que cada cliente executa em cada momento. Com o ficheiro tipo Artillery, o JepREST decide um cenário de cada vez e os clientes executam paralelamente as operações contidas nesse cenário. Com o grafo de transições, as operações são associadas aos clientes em tempo real, ou seja, quando um cliente recebe uma resposta, o novo gerador de operações utiliza o grafo para obter a próxima operação dependendo do resultado da anterior, simulando, assim, uma utilização mais realista da aplicação.

5. **Verificação dos *Logs***: Por fim, o JepREST aplica os modelos definidos em ambas versões da ferramenta e utiliza o *checker* para analisar os *logs* gerados durante a fase de teste. O *checker* verifica se os *logs* correspondem ao comportamento esperado da API, identificando possíveis erros ou incoerências.

Este processo garante que a API é testada de forma rigorosa, permitindo a deteção de erros tanto nas operações que seguem a semântica REST, quanto nas operações que possuem semântica personalizada.

A Figura 4.1 apresenta visualmente a visão geral do processo de teste do JepREST. Os retângulos azuis representam os componentes já existentes na primeira versão da

ferramenta e os retângulos vermelhos representam as adições que foram feitas nesta segunda versão.

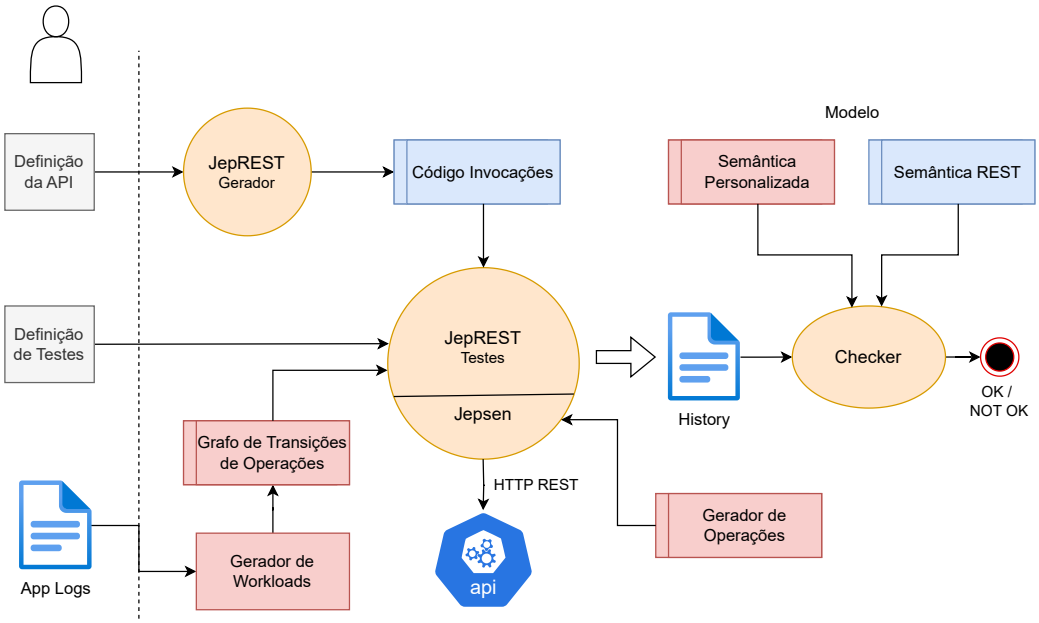


Figura 4.1: Processo de teste do JepREST.

4.1.2 Definição da API e Semântica da Aplicação

Durante a fase inicial do teste, o JepREST funciona como cliente da aplicação. Para que isto possa ocorrer, deve ser indicada toda a informação necessária para permitir a interação com a API da aplicação. Esta informação inclui os recursos usados na aplicação e as operações disponibilizadas. A informação sobre as operações também deve esclarecer como afetam cada recurso, pois na fase final do JepREST, é feita a verificação da correção da aplicação. Nas aplicações que utilizam a semântica REST, o método HTTP indica a ação realizada sobre os recursos. Nas aplicações que têm uma semântica personalizada, é necessário que na especificação das operações da API seja usada a linguagem definida na secção 4.2.

Estas informações fundamentais são detalhadas pelo utilizador em vários documentos, que, na prática, são interfaces escritas em Java. Existem três tipos de documentos:

- Documento Base - Este é um documento único que define a informação geral da API, como o título, a descrição, o *base path* dos recursos do serviço e os URLs dos servidores da aplicação, assim como os recursos presentes em cada servidor. Para isto são usadas as anotações @OpenAPIDefinition e @ApplicationPath, do OpenAPI e JAX-RS, respetivamente. Para além disto, este documento estende as interfaces relativas aos recursos oferecidos pela API, que são os que vão ser testados pelo JepREST.

A Listagem 4.1 apresenta um exemplo de um documento base de uma aplicação REST. A primeira linha define informações menos importantes, como o título da API, a versão, e a descrição. As informações mais relevantes são apresentadas a seguir. Primeiro, nas linhas 3-5, são definidos os URLs e os recursos de cada servidor, por exemplo, os recursos do tipo "*users*" e "*shorts*" estão todos armazenados no servidor com URL `http://150.30.208.1:8080/`. Nesta nova versão do JepREST, esta ultima informação também pode ser fornecida num ficheiro JSON, com o formato apresentado na Listagem 4.2. A seguir, na sétima linha, é especificado que a base do caminho de todos os pedidos feitos à aplicação é "*rest*". Por fim, na última linha, estão presentes todas as interfaces estendidas por este documento base.

Listagem 4.1: Exemplo de um documento base.

```
1 @OpenAPIDefinition(
2     info = @Info(title = "API Title", version = "1.0", description = "Description"),
3     servers = {
4         @Server(url = "http://150.30.208.1:8080/", description = "users, shorts"),
5         @Server(url = "http://155.2.116.23:8080/", description = "blobs") }
6 )
7 @ApplicationPath("rest")
8 public interface RestAppInterface extends RestUsers, RestShorts, RestBlobs { }
```

Listagem 4.2: Exemplo do ficheiro JSON que informa os URLs dos recursos.

```
1 [ {"resources": ["users", "shorts"], "servers": ["http://150.30.208.1:8080/"]},
2   {"resources": ["blobs"], "servers": ["http://155.2.116.23:8080/"]} ]
```

- Documento Interface - Cada um destes documentos é a interface de um dos recursos oferecidos pela aplicação, e contém informação relativa às operações que podem ser executadas sobre ele. Ademais, é com estas informações que o JepREST consegue saber o efeito que a execução de cada operação tem nos recursos da aplicação. Estas informações são detalhadas usando anotações do JAX-RS, do OpenAPI, do JepREST e da linguagem definida na secção anterior.

  Na Listagem 4.3, é possível ver um exemplo da definição de uma operação presente na interface do recurso "*users*". Como a operação em questão não contém a anotação @CustomMethod, o JepREST sabe que o efeito da mesma está especificado pelo método HTTP. Neste caso, a operação está definida como um POST, o que significa que cria um novo recurso do tipo "*users*", usando os dados enviados no objeto JSON do corpo do pedido.

- Documento Objeto - Estes documentos representam os objetos que vão ser usados na interação com o servidor, ou seja, os *Data Transfer Objects* (DTO). Os dados presentes nestes objetos são gerados automaticamente pelo JepREST, que interpreta as

Listagem 4.3: Exemplo de parte de um documento interface.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
@Operation(operationId = "createUser", description = "adds user", tags = {"users"},
    responses = {
        @ApiResponse(responseCode = "200", description = "Created",
            content = @Content(mediaType = MediaType.APPLICATION_JSON,
                    schema = @Schema(implementation = User.class)),
            links = {
                @Link(name = "GetUser", operationId = "getUser",
                  parameters = {
                    @LinkParameter(name = "id", expression = "$request.body#/userId"),
                    @LinkParameter(name = "pwd" , expression = "$request.body#/pwd")
                  })}),
        @ApiResponse(responseCode = "409", description = "User already exists") } )
String createUser(@RequestBody(required = true,
        content = @Content(mediaType = MediaType.APPLICATION_JSON, schema =
        @Schema(implementation = User.class))) User user);
```

anotações presentes em cada atributo, para saber como deve ser criado. As anotações que descrevem estas informações são do *javax.validation.constraints* ou do JepREST. Mais à frente neste relatório são definidas todas as anotações.

A Listagem 4.4 representa o documento objeto *User*, que é usado no corpo do pedido da operação da Listagem 4.3. Este objeto representa o recurso "*users*". Neste exemplo, são usadas as anotações @Pattern e @Size do *javax.validation.contraints*, para determinar o padrão e o tamanho das *Strings* geradas, e as anotações @Id e @Values, para definir o atributo que identifica o recurso e como são gerados os dados usando a biblioteca Faker [81].

Listagem 4.4: Exemplo de um documento objeto.

```
public class User {
    @Values("first (take 1 (faker.internet/emails))")
    public String email;

    @Pattern(regexp = "[A-Za-z0-9]+")
    @Size(min = 5, max = 25)
    @Id(ofResource = "users")
    public String userId;

    @Values("faker.name/first-name")
    @Pattern(regexp = "[A-Za-z]+")
    public String name;

    @Pattern(regexp = "[A-Za-z0-9]+")
    @Size(min = 6, max = 20)
    public String pwd; }
```

Estes documentos contêm toda a informação necessária para que o JepREST saiba como interagir com a aplicação em teste e como é alterado o estado após a execução de cada operação. Para processar todas estas informações, foi implementado um gerador de código que transforma os dados apresentados nas anotações em código Clojure, para que possam ser processados pelo JepREST, visto que o mesmo está implementado usando esta linguagem. Este gerador de código transforma as informações em funções que determinam como são realizados os pedidos, formatadas as respostas e definidos os dados enviados nos pedidos.

O gerador de código foi criado na primeira versão do JepREST e atualizado nesta nova versão para adicionar ou atualizar funcionalidades.

4.1.3 Definição dos Dados Usados nos Testes

Um teste do Jepsen é um programa executado no nó de controlo. Este nó tem o objetivo de controlar as operações executadas pelos clientes e recolher as respostas para, no fim, analisá-las. O resto dos nós representam o sistema em teste, que geralmente é uma base de dados. Para interagir com o sistema, o nó de controlo lança um número determinado de *worker-threads* que funcionam como clientes do sistema, conectando-se através do protocolo SSH. As operações que os clientes podem executar são escritas e leituras simples.

Ora, com este tipo de clientes e operações é impossível testar aplicações REST. Então, foi criado, por Simoes et al., um novo cliente do Jepsen com capacidade para submeter pedidos à aplicação em teste, via HTTP. Este cliente, chamado "*ClientREST*", é executado em todas as *worker-threads* e apenas tem a funcionalidade de fazer pedidos à aplicação. Os pedidos que são realizados, assim como a formatação das respostas, estão definidos no código gerado. Para além disto, o código gerado também tem as funções que determinam os dados que são enviados nos pedidos realizados.

O código gerado está dividido em duas partes. A primeira, é usada apenas pelo nó de controlo e contém as funções que geram os dados enviados nos pedidos. A segunda, é usada pelos clientes e envolve as funções que executam os pedidos, com os dados gerados na primeira parte, e interpretam as respostas. A seguir são explicadas em detalhe ambas as partes do código gerado.

4.1.3.1 Definição dos Parâmetros e Semântica dos Pedidos

Gerar os dados enviados nos pedidos não é uma tarefa simples. Os dados que os parâmetros de um pedido podem tomar dependem de vários fatores distintos, como o tipo de dados ou, até, o resultado de uma operação executada anteriormente, como é o caso dos identificadores de recursos. Portanto, o gerador de código é capaz de interpretar um conjunto de anotações Java que descrevem como devem ser definidos os valores dos parâmetros dos pedidos.

Parâmetros Independentes Os parâmetros independentes são aqueles que não dependem de nenhum outro parâmetro, nem de resultados de operações executadas anteriormente. O valor destes parâmetros é gerado automaticamente, seguindo as indicações das anotações.

Para definir o valor de um parâmetro que é uma *String*, podem ser usadas as anotações @Pattern e @Size. A @Pattern determina uma expressão regular que a *String* deve respeitar, enquanto que a @Size indica o número de caracteres mínimos e máximos que a *String* pode ter. Esta segunda anotação é complementar à primeira, logo, o seu uso não é obrigatório quando não existem restrições para o tamanho da *String*. Na Listagem 4.4 podemos encontrar o uso desta anotação, por exemplo, no atributo *userId* do documento objeto, onde é determinado que o valor pode conter letras e números e que deve ter entre 5 e 25 caracteres. Caso o parâmetro seja um inteiro, ou um *long*, podem ser definidos ambos os limites. O limite inferior é definido pela anotação @Min e o limite superior, pela anotação @Max. Estas anotações podem ser usadas em conjunto ou individualmente. Como é óbvio, estes limites não podem ir para além dos limites impostos pela linguagem de programação, neste caso, pelo Java. As quatro anotações descritas neste parágrafo são todas do *javax.validation.constraints* e é usada a biblioteca PETIT [59] para os gerar.

Muitas vezes, as anotações descritas no parágrafo anterior não bastam para definir como deve ser gerado um valor. Se o tipo de dado do parâmetro não for *String*, inteiro ou *long*, pode ser usada a anotação @Values. Nesta anotação pode ser definida uma função escrita em Clojure, que será a usada para gerar o valor do parâmetro. Por exemplo, com esta anotação, é possível definir que um parâmetro booleano é verdadeiro em metade dos pedidos de uma operação. Tal função, em Clojure, seria simplesmente "*if (< (rand) 0.5) true false*". É relevante destacar que as funções em Clojure são todas definidas dentro de parênteses, mas, nesta anotação, os parênteses da função externa não são incluídos, pois o gerador de código já os adiciona.

Como o JepREST tem o objetivo de testar a correção de aplicações REST num ambiente real, é importante que os dados enviados nos pedidos sejam realistas. Por isso, a anotação @Values é usada muitas vezes com funções da biblioteca Faker [81], como pode ser observado na Listagem 4.4. Mais especificamente, o atributo *email*, que usa a função "*(faker.internet/emails)*" para gerar vários emails realistas e escolher um deles.

Todas estas formas de gerar valores para os parâmetros independentes foram definidas na primeira versão do JepREST.

Dado que as anotações do *javax.validation.contraints* só permitem gerar tipos de dados simples, como *Strings*, inteiros e *longs*, e a anotação @Values necessita de certo conhecimento da linguagem Clojure, podem ser usadas as anotações @Collections e @Objects, definidas por Simoes et al., e a anotação @Blob, definida por nós.

A anotação @Collections determina de que maneira conjuntos, como listas e mapas, devem ser criados. Nesta, são definidas informações sobre os elementos das coleções, como o padrão e tamanho dos elementos, caso sejam *Strings*, ou os limites inferior e superior, caso sejam inteiros ou *longs*. No caso de serem um tipo de dados diferentes podem ser

definidos usando a anotação @Values ou @Objects.

A anotação @Objects é usada para indicar a classe à qual pertence um parâmetro que tem como tipo uma classe personalizada. Os atributos desta classe devem ter as devidas anotações para que o gerador saiba como os construir.

Quando o corpo de um pedido é um ficheiro (i.e., um *byte array*), como uma imagem ou um vídeo, as anotações anteriores não têm capacidade para indicar como deve ser formado. Para resolver esta limitação, criámos a anotação @Blob. Esta anotação permite definir se o *array* de *bytes* deve ser gerado aleatoriamente, dentro de uns limites de tamanho, ou se é um ficheiro armazenado no sistema, podendo definir o caminho para a pasta onde se encontra o ficheiro. Como o mais comum é que cada pedido envie um ficheiro distinto, o ficheiro enviado será um escolhido aleatoriamente dentro da pasta definida. Caso se queira enviar sempre o mesmo, a pasta deve apenas conter esse ficheiro.

Parâmetros Dependentes Nas aplicações REST, é normal que as operações dependam umas das outras. Numa rede social, a operação de comentar uma publicação depende da existência da mesma. Neste caso, o pedido que seria feito à aplicação teria de ter, como parâmetros, no mínimo, o identificador do utilizador e o identificador da publicação. Isto significa que tem de haver uma forma de determinar as dependências entre as operações e os parâmetros que as relacionam.

Nestas aplicações existem dois tipos de operações: operações dependentes e operações geradoras. Uma operação dependente têm pelo menos um parâmetro cujo valor foi gerado por outra operação. Uma operação geradora, que também pode ser dependente, é uma operação que gera dados que são usados por outras operações.

Para tratar destas dependências, é usada a anotação @Link do OpenAPI. Esta anotação permite determinar as dependências entre as operações e indicar quais são os parâmetros que passam de uma operação para outra. Ou seja, as operações geradoras retornam dados que vão ser usados como parâmetros nas operações dependentes. A operação que tem a anotação @Link presente, é a operação geradora, e o conteúdo da anotação determina qual é a operação dependente e quais são os parâmetros que as conectam. Geralmente, a dependência entre duas operações está relacionada com o identificador de um recurso, mas existem muitos casos onde as operações usam, como parâmetros, para além do identificador, outras informações relevantes do recurso, como a palavra-passe.

O gerador de código, perante o uso desta anotação, armazena numa lista os valores indicados, para que, quando um cliente do JepREST realize um pedido a uma operação dependente, sejam usados os dados criados pela operação geradora nos parâmetros. Isto pode causar alguns problemas quando o nome de um parâmetro numa operação dependente é diferente do nome do atributo do resultado de uma operação geradora. Nestes casos, deve ser determinado que o parâmetro Y da operação dependente, tomará o valor do atributo X do resultado da operação geradora. Para isto, nesta versão, foi definida um extensão do OpenAPI, chamada "*LinkParameterId*", que permite indicar o nome do

parâmetro dependente e o nome do atributo do resultado da operação geradora. Esta extensão deve estar presente na anotação @Link correspondente.

O uso da anotação @Link tem algumas limitações, que são contornadas através da anotação @ExtendLink, definida por Simoes et al. Esta anotação deve ser usada em conjunto com uma de duas outras anotações, dependendo da limitação que se pretende contornar. Estas anotações são: @RequestBodyParameter, criada na primeira versão do JepREST, e @ExtendLinkParameter, definida nesta nova versão.

A limitação principal da @Link é que só permite definir os valores dos parâmetros de caminho/pesquisa dos pedidos. Portanto, caso o atributo do corpo de um pedido dependa de um valor gerado por outra operação, deve ser usada a anotação @RequestBodyParameter. Nesta anotação, são definidas informações como o identificador da operação dependente, os atributos do corpo do pedido que usam valores gerados pela operação geradora e os atributos gerados que criam a dependência.

A outra limitação é mais específica e está relacionada com o facto de, por vezes, uma operação dependente ter dois ou mais parâmetros que dependem de execuções diferentes da mesma operação geradora. Por exemplo, nas redes sociais, a operação de seguir um utilizador precisa de dois identificadores distintos, que são gerados por diferentes execuções da mesma operação de criação de conta. Ademais, a operação pode exigir que o utilizador tenha que enviar a palavra-passe para fazer a autenticação. Neste caso, o identificador e a palavra-passe têm que ter sido gerados pela mesma execução da operação de criar conta. Para solucionar isto, foi criada a anotação @ExtendLinkParameter, que permite definir quantas execuções distintas da operação geradora são necessárias para poder executar a operação dependente e os nomes dos parâmetros e atributos correspondentes. Na maioria dos casos, o uso desta anotação deve ser acompanhado da extensão "*LinkParameterId*" do OpenAPI, pois, quando dois parâmetros referenciam o mesmo atributo criado pela operação geradora, pelo menos um deles tem que ter um nome diferente, uma vez que é impossível ter dois parâmetros iguais numa operação.

Conforme foi explicado mais acima, o código gerado tem listas que contêm os valores dos quais algumas operações dependem. Estes valores são introduzidos nas listas correspondente pelas operações geradoras e são acedidos pelas operações dependentes. Isto significa que, depois de serem adicionados à lista, nunca são atualizados. Como estes valores representam os valores dos recursos guardados no estado da aplicação, quando os recursos são atualizados, os valores correspondentes na lista também devem ser atualizados. Para tratar desta situação, criámos a anotação @LinkUpdate, que deve ser usada em todas as operações que atualizem algum atributo de um recurso que esteja presente na lista, ou seja, que vá ser usado numa operação dependente. Nesta anotação devem ser definidas as seguintes informações: o identificador da operação geradora, para saber qual a lista que contêm os valores do recurso atualizado; o conjunto de parâmetros que identificam o recurso dentro da lista, normalmente é só o identificador, mas é um conjunto para o caso do recurso ser identificado por vários atributos; e um conjunto com os atributos da lista que são atualizados e qual o valor que devem tomar após a execução

da operação.

Nesta nova versão do JepREST, o gerador de código, para além de processar as anotações que determinam como devem ser gerados os dados dos pedidos, também interpreta as anotações da linguagem apresentada na secção 4.2. Para as operações da API que não contêm a anotação @CustomMethod, não é adicionada qualquer informação sobre o comportamento interno da operação, uma vez que o método HTTP associado já o determina (i.e., um POST cria um recurso, um GET devolve um ou mais recursos, um PUT atualiza um recurso). Para as operações que utilizam a anotação, é indicado que as mesmas apresentam um comportamento personalizado e que o método HTTP associado apenas indica como devem ser realizados os pedidos a essas operações. O código gerado para estas operações contém toda a informação relativa às operações internas e ao resultado esperado delas. Posto de outra forma, quando uma operação da API não cumpre a semântica padrão da arquitetura REST, o código gerado apresenta a sequência de operações que interagem com os recursos, dentro dessa mesma operação da API, e o resultado que cada uma deve apresentar, dependendo do código de estado presente na resposta do pedido. Esta informação é relevante para o *checker* utilizado no JepREST, uma vez que, ao verificar a linearização da história gerada, simula as transições do estado do sistema, e, para isso, precisa saber como cada operação modifica os recursos desse estado.

Estas informações são todas armazenadas no formato de uma operação Jepsen, para poderem ser interpretadas pelos clientes na hora de realizar os pedidos à aplicação. Cada operação do Jepsen, no JepREST, é apenas um mapa com a estrutura apresentada na Listagem 4.5. O significado da estrutura é o seguinte:

:type Indica se a operação é uma invocação de um pedido (*:invoke*), resposta a um pedido (*:ok*), ou que a resposta não pode ser processada corretamenta devido a algum erro (*:fail*).

:f Determina o método HTTP que o cliente deve usar no pedido.

:value Contém os dados que foram gerados para a realização do pedido e toda a informação relacionada com a semântica da operação.

Listagem 4.5: Estrutura de uma operação do Jepsen no JepREST.

```
1 {:type :invoke/:ok/:fail
2  :f :get/:post/:put/:patch/:delete
3  :value <informação relevante para o cliente realizar o pedido>}
```

4.1.3.2 Definição dos Pedidos e Respostas

Após a definição dos parâmetros e da semântica das operações da API, o gerador de código cria as funções que executam os pedidos e que interpretam as respostas de uma maneira que o *checker* Knossos possa verificar.

Estas funções começam por enviar o pedido à aplicação com as informações definidas na especificação da API, como o caminho, o método HTTP, os cabeçalhos e o corpo. Os dados enviados no pedido são os que foram calculados na fase anterior e que estão presentes na operação do Jepsen, que é recebida como parâmetro da função. Após receber a resposta ao pedido é verificado o código de estado e a resposta é adicionada ao elemento *:value* da operação do Jepsen. Caso o código não seja nenhum dos presentes na especificação, significa que um erro não esperado pode ter acontecido (e.g., a especificação indica que uma operação devolve os códigos 200 e 404, mas na resposta está o código 409), então, o elemento *:type* da operação do Jepsen é trocado para *:fail*, que significa que o *checker* vai ignorar a operação em questão. Caso o código seja um dos esperados, o elemento *:type* toma o valor *:ok*.

4.1.4 Definição e Execução do Conjunto de Testes

Existem duas soluções distintas para executar testes no JepREST. Estas dependem da forma como é definida a sequência de operações que serão executadas pelos clientes. Na primeira solução, a sequência de operações é predefinida manualmente num ficheiro, enquanto que na segunda, a sequência de operações é construida à medida que os clientes vão executando operações.

4.1.4.1 Definição Manual dos Testes

Na solução apresentada por Simoes et al., as operações que a ferramenta executa são especificadas manualmente num ficheiro YAML inspirado nos *scripts* do Artillery [62], onde são definidos um ou mais cenários, cada um com um peso relativo, que contêm um conjunto de operações. Na Listagem 4.6, pode ser observado um exemplo de um ficheiro YAML que define as operações que o JepREST vai testar. No primeiro cenário, são criados dois recursos, um utilizador e um *short* (vídeo de curta duração). No segundo cenário, somente são executadas duas leituras, uma a um utilizador e outra a um *short*. A probabilidade do JepREST escolher o primeiro cenário (70%) é superior à probabilidade de escolher o segundo (30%). Quando utilizamos o JepREST para testar uma aplicação, este ficheiro é lido e interpretado pelo nó de controlo, escolhendo o cenário que irá ser executado em cada momento. Uma vez escolhido o cenário, o nó de controlo envia as operações aos clientes para que as executem paralelamente e espera pela resposta de todos antes de decidir um novo cenário. Ora, esta abordagem tem duas limitações importantes:

1. Os testes não representam execuções realistas da aplicação. Normalmente, num sistema REST, as operações que os utilizadores executam são dependentes do resultado das anteriores. Se determinamos que operações serão executadas antes sequer de correr a aplicação, retiramos o sentido de causalidade que estas têm. Olhando para o exemplo listado, nenhum dos dois cenários parece que seja uma execução real de um sistema.

2. Um cliente só executa uma operação quando os outros estiverem disponíveis. Aqui é criada uma espécie de dependência entre os clientes, onde se desperdiça tempo de teste. Quando um cliente recebe uma resposta ao pedido que efetuou, em vez de atribuir-lhe uma nova operação, adicionando mais concorrência ao sistema para aumentar a possibilidade de descobrir problemas, a ferramenta deixa-o inativo. No exemplo apresentado, por cada execução de um cenário, apenas dois clientes executarão as operações, mesmo tenham sido definidos mais clientes.

Listagem 4.6: Exemplo de um ficheiro de teste YAML.

```
scenarios:
  - name: 'Create user and short'
    weight: 70
    flow:
      - createUserData
      - createShortData
  - name: 'Get user and short'
    weight: 30
    flow:
      - getUserData
      - getShortData
```

4.1.4.2 Definição Automática dos Testes

Considerando que o JepREST tem o objetivo de testar aplicações REST reais, a solução anterior não é a mais adequada. Por isso, criámos uma ferramenta que gera um grafo de transições das operações especificadas na API da aplicação. Este grafo permite ao nó de controlo do JepREST pedir as operações que cada cliente deve executar em cada momento, dependendo da última resposta que cada um recebeu. Esta ferramenta esta detalhada na secção 4.3.

Tendo uma ferramenta que permite a cada cliente gerar a sua sequência de operações em tempo real, para poder usá-la no JepREST, foi necessário criar um novo gerador do Jepsen. Os geradores do Jepsen são os responsáveis por determinar em qualquer momento as operações que os clientes devem executar na aplicação em teste. Este novo gerador é o intermediário entre o nó de controlo do JepREST e o grafo de transições de operações.

O gerador usa os dois únicos métodos oferecidas pelo grafo. Ambos devolvem o identificador de uma operação, mas são usados em momentos diferentes. O primeiro método, indica a primeira operação que um cliente deve executar, enquanto que o segundo, usa o resultado da última operação executada pelo cliente para gerar a próxima. Para iniciar o novo gerador, é criada uma lista de primeiras operações, onde cada elemento da lista é referente a um cliente. Assim que é criado o gerador, o nó de controlo envia as operações iniciais para os respetivos clientes, que realizam os devidos pedidos à aplicação. Enquanto nenhum cliente recebe uma resposta, o gerador não atua. Assim que um cliente recebe uma resposta, o gerador obtém o código de estado da mesma e solicita ao grafo

a próxima operação para esse cliente, que será colocada na posição da lista respetiva a esse cliente. Quando o grafo indica que o cliente não tem mais operações para executar, o gerador reinicia o seu comportamento para esse cliente, pedindo uma nova primeira operação ao grafo. Este processo é executado continuamente durante o tempo estabelecido pelo utilizador.

O novo gerador, juntamente com o grafo de transições de operações, elimina as limitações da abordagem anterior, onde é usado um ficheiro estilo Artillery para definir as operações que os clientes devem executar. Primeiro, a ferramenta que gera o grafo tem como referência a atividade dos utilizadores reais da aplicação, através dos *logs*, para saber que operação deve ser executada em cada momento, dependendo da resposta da anterior. Portanto, os testes representam execuções realistas da aplicação. E segundo, para escolher a operação que o cliente deve executar em cada momento, basta saber qual foi o resultado da operação anterior executada por esse mesmo cliente. Desta forma, não existe nenhuma dependência entre os clientes, o que significa que um cliente pode fazer um pedido à aplicação logo após receber uma resposta. Apesar disso, o JepREST permite que o utilizador escolha qual abordagem utilizar, uma vez que, na ausência de acesso aos *logs* da aplicação, não é possível usar a nova abordagem apresentada.

4.1.5 Avaliação dos Resultados

Todos os pedidos realizados pelos clientes, assim como as respetivas respostas, são armazenados num ficheiro chamado *history*. Assim que o nó de controlo do JepREST decide a operação da API que um cliente deve efetuar, é gerada a operação do Jepsen que contém todos os dados necessários para executar o pedido (i.e., a operação que tem como *:type*, *:invoke*) e é adicionada à *history*, juntamente com informações adicionais como o cliente que a executa, o *timestamp* e o índice da operação na história. Quando é recebida a resposta a esse pedido (i.e., a operação que tem como *:type*, *:ok* ou *:fail*), as mesmas informações são adicionadas e a história é atualizada para conter essa resposta. Na prática, a história é mantida em memória durante a fase de teste, e só é escrita no ficheiro *history* quando o tempo definido pelo utilizador termina.

A Listagem 4.7 mostra o exemplo de uma resposta presente no ficheiro *history*. A única diferença entre o conteúdo das respostas e dos pedidos armazenados na história é que a resposta contém o output gerado pela aplicação. Cada operação do Jepsen contida no ficheiro *history* apresenta as seguintes informações:

- Tipo da operação Jepsen (*:type*) - Indica se a informação representa uma invocação (*:invoke*) ou uma resposta da aplicação (*:ok* ou *:fail*).
- Tipo de operação REST (*:f*) - Indica o método REST usado no pedido executado pelo cliente.
- *Timestamp* (*:time*) - Representa em que momento foi invocado um pedido ou recebida uma resposta.

- Identificador do processo (*:process*) - Identifica o cliente que realizou o pedido ou recebeu a resposta da aplicação.

- Índice (*:index*) - Estabelece a posição do pedido/resposta na história. Este elemento é crucial para determinar se a aplicação respeita as garantias de consistência, pois permite verificar se a ordem das operações é coerente.

- Informação adicional (*:value*) - Representa informação relevante sobre os pedidos realizados ou respostas recebidas.

  Quando é um pedido, este elemento contém os dados necessários para que o cliente possa executá-lo, como os parâmetros, cabeçalhos e corpo. Esta informação é definida no elemento *:input*. Em adição, no elemento *:custom-op*, é adicionada a sequência de operações REST que determina o comportamento interno da operação. Por exemplo, a resposta apresentada na listagem é referente a uma operação que, ao apagar um utilizador da aplicação, também remove todos os artigos criados por ele.

  Quando é uma resposta, para além dos dados relativos ao pedido, estabelecidos no campo *:input*, e dos dados relativos ao comportamento da operação, estabelecidos no elemento *:custom-op*, também são apresentados os dados devolvidos pela aplicação, isto é, o código de estado HTTP, os cabeçalhos e o corpo da resposta. Estes dados estão presentes no elemento *:output*.

Após a criação do ficheiro *history*, começa a fase de avaliação dos resultados obtidos na fase de teste. Para avaliar a correção da aplicação em teste, o JepREST usa um *checker* do Jepsen chamado Knossos. O Knossos verifica se a história gerada na fase de teste é linearizável, ou seja, se a história apresenta um comportamento equivalente ao de um sistema com uma só cópia do estado. Para isto, o *checker* percorre o ficheiro *history* e executa localmente as operações para poder verificar se as respostas são coerentes com o modelo definido. Este modelo indica as alterações que são efetuadas no estado por cada operação do sistema em teste. Deste modo, o *cheker* consegue manter o estado localmente, podendo, assim, saber qual deveria ser o resultado das operações em cada momento da história.

Para o caso do teste de aplicações REST, foi criado um novo modelo do Knossos, que determina como as operações de uma aplicação REST modificam o estado. Neste tipo de aplicações, podem existir dois tipo de operações, as que seguem a semântica da arquitetura e as que têm um comportamento personalizado.

Para as operações que seguem a semântica do REST, o modelo define que as atualizações do estado são determinadas pelo método HTTP do pedido e pelo código de estado da resposta. Então, o Knossos sabe que um GET devolve um recurso, um POST cria um recurso, um PUT/PATCH atualiza um recurso e um DELETE remove um recurso. Para além disto, também sabe que quando uma resposta contém o código 404, a operação falhou porque um recurso não existe, quando o código é 409, a operação não teve sucesso devido a que já existe o recurso em questão, e que qualquer código 2XX implica que a

operação teve sucesso e que devem ser feitas as devidas alterações ao estado. Ademais, quando o código é de sucesso, o *checker* verifica se o resultado da operação é coerente com o estado. Por exemplo, se a operação de sucesso for um GET, o *checker* verifica se o recurso retornado na resposta da história é igual ao recurso presente no estado. Esta parte do modelo foi implementada na versão anterior da ferramenta. Pelo contrário, a parte explicada a seguir foi definida nesta nova versão.

Listagem 4.7: Exemplo de uma resposta no ficheiro *history*.

```
{:type :ok,
 :f :delete,
 :value {
    :custom-op {
        :op-steps
            [{:method :delete,
              :resource "users",
              :filter {:simple-filter [{:attribute :userId,
                                        :value "$request.path#/userId"}]}}
             {:method :delete,
              :resource "articles",
              :filter {:simple-filter [{:attribute :author,
                                        :value "$request.path#/userId"}]}}],
        :op-status-code-seq
            [{:op-status 404, :steps-status-seq [404 -1]}
            {:op-status 200, :steps-status-seq [200 1]}}],
    :input {:typeOp "deleteUser",
               :path {:userId "Enoch"}},
    :output {:status 200,
                :headers {"Date" "Sat, 07 Sep 2024 21:01:56 GMT",
                          "Content-type" "application/json",
                          "Content-length" "88"},
                :body {:userId "Enoch",
                       :pwd "Nn3a4w",
                       :email "myrtis@gmail.com",
                       :displayName "Wilmer Veum"}}},
 :time 2520276832,
 :process 2,
 :index 5}
```

Para as operações que têm uma funcionalidade personalizada, o modelo é um pouco mais complexo. O comportamento destas operações não é determinado pelo método HTTP e os códigos de estado das respostas também não têm um significado fixo. Neste caso, o modelo foi adaptado para o uso da linguagem apresentada na secção 4.2. Esta linguagem permite especificar as alterações que as operações realizam no estado, dividindo as mesmas em sequências de suboperações simples que adotam a semântica REST. Isto quer dizer que o método HTTP associado a cada suboperação dessa sequência indica como é afetado o estado, pois cada suboperação usa a semântica REST. A linguagem define o comportamento de cada suboperação da sequência, dependendo do código de estado da

resposta. Portanto, para verificar o resultado de uma operação, o *checker* utiliza o código da resposta na história para determinar quais são os possíveis resultados das suboperações. A seguir, o *checker* itera sobre estas suboperações e, uma a uma, executa-as, verificando se o resultado é coerente com o estado. Tomando como exemplo uma operação de uma API dividida em dois PUTs, cuja especificação diz que, quando a resposta contém o código 404, a sequência de operações PUT deve retornar 200 e 404, se o Knossos as executa e ambas retornam 200, e na resposta da história o código é 404, significa que o recurso do segundo PUT devia existir no estado da aplicação no momento da resposta na história. Neste caso, a verificação terminaria e a inconsistência seria exposta ao utilizador.

Definir uma suboperação através de um verbo HTTP não significa que essa suboperação seja necessariamente um pedido HTTP executado dentro da operação da API. Ou seja, se uma suboperação for definida como um GET, isso não implica que seja um GET a um servidor, mas pode, em vez disso, representar uma operação de leitura numa base de dados. Os métodos HTTP das suboperações podem ser compreendidos através da lógica da semântica CRUD, onde o GET corresponde a uma operação de leitura (*Read*), o POST representa a criação de novos dados (*Create*), o PUT ou PATCH indica uma atualização (*Update*) e, como seria de esperar, o DELETE corresponde à remoção de dados (*Delete*).

Após a verificação de incoerências nas suboperações, se a linguagem especificar quais são os dados enviados na resposta pela aplicação, o *checker* verifica se o conteúdo da resposta é coerente com os recursos armazenados no estado local. O insucesso desta verificação indica que os valores dos recursos enviados na resposta não podem ter sido gerados pela sequência de operações executadas anteriormente. O *checker*, então, notifica o utilizador de que o estado da aplicação está incorreto, de acordo com o modelo definido.

Como o modelo necessita ter o estado atualizado em cada momento do teste, e o estado apenas é alterado pelas operações presentes na história, o estado inicial da aplicação tem de estar vazio. Caso não esteja, o *checker* pode detetar incoerências com o estado local mesmo quando a resposta é coerente com o estado da aplicação. Devido a isto, a aplicação não pode receber pedidos de escrita antes de usar o JepREST. Logo, para testar uma aplicação REST, a mesma deve ser colocada a executar momentos antes do processo de teste do JepREST.

Este modelo permite ao JepREST testar a correção de aplicações REST que oferecem operações com funcionalidades personalizadas nas suas APIs.

4.2 Suporte de Semântica e Funcionalidades Personalizadas em Métodos REST

4.2.1 Motivação

Muitos sistemas REST expõem operações que não seguem o conjunto de regras semânticas do estilo arquitetural, visto que este limita o tipo de operações que um sistema pode disponibilizar. Existem várias razões para que os desenvolvedores de sistemas REST

decidam fugir às regras para poder criar as operações que desejam. Por exemplo, os desenvolvedores do sistema Dropbox decidiram usar o método HTTP POST em vez de GET, pois este último não permite que o pedido tenha um corpo, levando a que todos os parâmetros necessários tenham de estar no caminho do pedido, podendo gerar problemas com o tamanho do URL, pois a maioria dos clientes e servidores HTTP têm um limite de entre 2kB a 8kB [45]. Isto leva a uma certa confusão, pois as operações estão definidas como POST, mas não criam nenhum recurso. Para além disto, muitas operações dos sistemas atuais, em vez de executar apenas uma operação num recurso, ou em vários do mesmo tipo, executam várias operações num ou mais recursos (i.e., cada operação da API pode realizar múltiplas ações, permitindo aceder, modificar e/ou apagar um ou mais recursos).

Nenhuma destas informações é possível ser apresentada na especificação OAS, visto que não tem capacidade para descrever a execução interna das operações da API, usando simplesmente o método HTTP para sinalizar o resultado da operação (i.e., um POST cria um recurso, um GET retorna um recurso). Um exemplo simples pode ser um sistema de publicação de vídeos, onde o recurso dos vídeos guarda o número de visualizações. Quando um utilizador assiste a um vídeo, para além da operação ter de devolver o vídeo em questão, também tem de atualizar o número de visualizações do mesmo. Isto significa que a operação não devolve simplesmente um recurso, mas também o atualiza. Podemos, então, dividir o comportamento desta operação em duas, primeiro um GET e depois um PUT, como apresentado na Listagem 4.8.

Listagem 4.8: Divisão visual dos passos executados na operação que devolve um vídeo.

```
1 video = GET videos/{videoId}
2 video.views++
3 PUT videos/{videoId} body={video}
```

Para lidar com este problema, definimos uma linguagem capaz de descrever o funcionamento interno das operações da API, dividindo-as em operações REST simples.

4.2.2 Desenho da Linguagem

A linguagem é expressa num conjunto de anotações Java que permite dividir uma operação da API numa sequência de operações que respeitam a semântica da arquitetura REST e especificar qual deve ser o resultado de cada uma dessas operações, dependendo do resultado da operação principal.

A seguir estão apresentadas as construções da linguagem.

4.2.2.1 CustomMethod

A @CustomMethod é a anotação principal da linguagem. Esta especifica que a operação em questão não respeita a semântica da arquitetura REST e que apresenta uma funcionalidade personalizada. O resto das anotações da linguagem são usadas dentro desta. A Listagem 4.9 apresenta o código da anotação. As duas primeiras linhas definem, respetivamente, que a

anotação só pode ser usada em métodos e que está disponível durante todo o tempo de execução do programa, pois guarda informação relevante sobre a operação da aplicação. Quanto ao conteúdo da anotação, existem duas listas (dois *arrays*, em termos de código). A *operation* é a lista que contém a sequência de operações REST. Os elementos dessa lista são chamados de passos, uma vez que a ordem deles é importante. A outra lista, *statusCodeSeq*, define o resultado que cada passo deve retornar dependendo do *status code* da operação principal. Para além destas duas listas, a anotação contém um elemento opcional chamado *responseCondition*, que pode ser utilizado para definir quais são os dados presentes na resposta da operação.

Listagem 4.9: Código da anotação @CustomMethod.

```
1 @Target({METHOD})
2 @Retention(RUNTIME)
3 public @interface CustomMethod {
4     CustomMethodStep[] operation();
5     StatusCodeSequence[] statusCodeSeq();
6     ResponseCondition responseCondition() default @ResponseCondition(function = "",
          ↪ functionParams = {});}
```

4.2.2.2 CustomMethodStep

A anotação @CustomMethodStep representa os passos do CustomMethod. É aqui que são definidas as operações REST que especificam o comportamento interno das operações da API. Na listagem 4.10 pode-se ver o código da anotação.

O *resource* representa o caminho do pedido e serve para determinar o tipo do recurso (e.g., "/users"), ou o recurso (e.g., "/users/u1", "/users?age=30&sex=M"), ao qual vai ser aplicada a ação indicada pelo método HTTP definido no *operationType*. Estes dois elementos são obrigatórios, pois todas as operações REST têm de ter um caminho e um método HTTP associados. O resto dos elementos desta anotação são opcionais e usados em diferentes ocasiões.

O *filter*, tal como o nome indica, serve para filtrar os recursos ou o recurso alvo de cada passo. Quando o recurso já está especificado no elemento *resource*, este elemento é ignorado. Quando o *resource* só especifica o tipo do recurso, então, o *filter* é usado para definir o recurso desejado.

O *write* tem como finalidade determinar o valor que os atributos dos recursos vão tomar. Este elemento só é usado quando é criado ou atualizado algum recurso, ou seja, quando o *operationType* é um POST, PUT ou PATCH. Os dois últimos elementos não estão associados à arquitetura REST, mas sim à relação entre os passos do CustomMethod.

O *storeInVar* permite definir o nome de uma variável que guarda os recursos resultantes da operação executada no próprio passo. Esta variável pode ser acedida pelos passos seguintes, caso seja necessário obter alguma informação desses recursos. A Listagem 4.11 expõe um exemplo do uso deste elemento. Considere-se um sistema de publicação de

artigos de pesquisa, onde cada artigo guarda o identificador do seu autor. Para saber qualquer dado acerca do autor de um artigo, conhecendo apenas o identificador do artigo, é necessário ter uma operação cujo comportamento seja igual a executar um GET desse artigo (linha 1) e um GET do utilizador com o identificador do autor do artigo (linha 2). Neste caso, no primeiro passo, o *storeInVar* tem como valor "*article*" e guarda o artigo com identificador "art1". No passo seguinte, é usado o autor do artigo guardado na variável "*article*" para obter o utilizador desejado.

O elemento *condition* estabelece uma condição que deve ser atendida para a execução de um passo. Se essa condição estiver presente num passo e não for satisfeita, o passo deve ser ignorado.

Listagem 4.10: Código da anotação @CustomMethodStep.

```
1 public @interface CustomMethodStep {
2     String resource();
3     HTTPVerb operationType();
4     Filter filter() default @Filter;
5     Write write() default @Write;
6     String storeInVar() default "";
7     Condition condition() default @Condition;
8 }
```

Listagem 4.11: Passos executados na operação que devolve o utilizador que escreveu um determinado artigo.

```
1 article = GET articles/art1
2 GET users/article.author
```

4.2.2.3 Filter

A anotação @Filter, apresentada na Listagem 4.12, tem como finalidade definir os recursos que são afetados pelo passo em questão. Para filtros simples baseados em igualdades, usa-se o *simpleFilter*, que é um conjunto de anotações @SimpleFilter, presente na Listagem 4.13. Esta última anotação permite definir um atributo do recurso e o valor que tem de tomar, usando os elementos *attribute* e *value*, respetivamente, enquanto que o elemento *type* determina o tipo de dado do elemento *value*. Uma forma menos verbosa de fazer este tipo de filtragem é usando o elemento *resource*, da anotação @CustomMethodStep, e escrever as igualdades em forma de *query parameters*. Convém destacar que se algum dos valores de pesquisa não for uma *String*, é obrigatório usar o *simpleFilter*. As Listagens 4.14 e 4.15 mostram a diferença entre as duas formas de filtragem simples. A primeira abordagem ocupa menos código e é mais legível, mas os valores têm de ser *Strings*, enquanto que na segunda podem ser definidos os tipos de dados dos valores, sendo *String* o tipo padrão.

Listagem 4.12: Código da anotação @Filter.

```
public @interface Filter {
    SimpleFilter[] simpleFilter() default {};
    String filterFunction() default "";
    String[] functionParams() default {};
}
```

Listagem 4.13: Código da anotação @SimpleFilter.

```
public @interface SimpleFilter {
    String attribute();
    String value();
    Class<?> type() default String.class;
}
```

Listagem 4.14: Filtragem com *query parameters*.

```
@CustomMethodStep(operationType = HTTPVerb.GET, resource = "users?role=employee&sex=M")
```

Listagem 4.15: Filtragem com @SimpleFilter.

```
@CustomMethodStep(operationType = HTTPVerb.GET, resource = "users",
          filter = @Filter(
              simpleFilter = {
                    @SimpleFilter(attribute = "age", value = "30", type = Integer.class),
                    @SimpleFilter(attribute = "sex", value = "M")}))
```

Muitas vezes a filtragem dos recursos não pode ser feita unicamente usando igualdades. Por exemplo, uma operação que devolve os utilizadores maiores de idade (i.e., que a idade seja maior que 18), não pode ser definida usando nenhum dos dois métodos anteriores. Para isto, existem os elementos *filterFunction* e *functionParams*, que permitem definir uma operação de filtragem e os seus parâmetros. No caso da utilização desta linguagem no JepREST, a operação deve estar escrita em Clojure. A Listagem 4.16 exibe como é definida a operação em questão e os seus parâmetros. A função em Clojure "(fn [x] (> x 18))" é verdadeira quando x é maior que 18 e o parâmetro "$*resource*#/*age*" define que o valor que x toma é o que está guardado no atributo *age* do recurso. Esta função é aplicada a todos os recursos do tipo "*users*", e o resultado da filtragem é o conjunto de recursos que cumpriram a condição definida na função.

Listagem 4.16: Filtragem usando uma função.

```
@CustomMethodStep(operationType = HTTPVerb.GET, resource = "users",
          filter = @Filter(
              filterFunction = "(fn [x] (> x 18))",
              functionParams = {"$resource#/age"}
          ))
```

4.2.2.4 Write

A anotação @Write é usada para indicar os valores dos atributos de recursos que são criados ou atualizados. Quando o método HTTP é definido como POST, essa anotação determina os valores atribuídos aos atributos do recurso criado. Quando o método é definido como PUT ou PATCH, a anotação especifica os valores que os atributos atualizados dos recursos tomarão, após a aplicação da filtragem explicada anteriormente. Os valores atribuídos aos atributos podem ser constantes ou calculados por meio de uma função. Por isso, a anotação, apresentada na Listagem 4.17, possui dois elementos: *simpleWrite*, para atribuir valores constantes, e *complexWrite*, para atribuir valores calculados usando a função desejada. O elemento *simpleWrite* é uma lista de anotações @SimpleWrite (Listagem 4.18), onde são definidas as seguintes informações: o nome do atributo, o valor que ele deve assumir e o tipo de dado, através dos elementos *attribute*, *value* e *type*. Além disso, quando a operação envolve a criação de um recurso, o elemento *isID* indica se o atributo é o identificador do recurso. O elemento *complexWrite* é uma lista de anotações @ComplexWrite (Listagem 4.19), que permite definir uma função e os seus parâmetros, através dos elementos *function* e *functionParams*. O resultado dessa função será o valor atribuído ao atributo especificado no elemento *attribute*.

Listagem 4.17: Código da anotação @Write.

```
1 public @interface Write {
2     SimpleWrite[] simpleWrite() default {};
3     ComplexWrite[] complexWrite() default {};
4 }
```

Listagem 4.18: Código da anotação @SimpleWrite.

```
1 public @interface SimpleWrite {
2     String attribute();
3     String value();
4     Class<?> type() default String.class;
5     boolean isID() default false;
6 }
```

Listagem 4.19: Código da anotação @ComplexWrite.

```
1 public @interface ComplexWrite {
2     String attribute();
3     String function();
4     String[] functionParams() default {};
5 }
```

A Listagem 4.20 mostra como é definida a criação de um recurso usando a linguagem. Neste caso, a operação cria um recurso do tipo *articles*, onde os valores que os atributos, *id*, *author* e *text*, vão assumir, estão no corpo do pedido que o utilizador executa. Para além disto, o atributo *views*, que representa o número de utilizadores que acederam ao

artigo, começa a zero, logo, é necessário definir que este valor não é uma *String*, mas sim um inteiro. Ao usar a anotação @SimpleWrite, é sabido que os valores do recurso serão exatamente os especificados no elemento *value*.

Listagem 4.20: Exemplo do uso do @SimpleWrite.

```
@CustomMethodStep(operationType = HTTPVerb.POST, resource = "articles",
       write = @Write(
              simpleWrite = {
                     @SimpleWrite(attribute = "id", value = "$request.body#/id"),
                     @SimpleWrite(attribute = "author", value = "$request.body#/userId"),
                     @SimpleWrite(attribute = "text", value = "$request.body#/text"),
                     @SimpleWrite(attribute = "views", value = "0", type = Integer.class)
              }
       ))
```

Por outro lado, a Listagem 4.21 apresenta um exemplo de definição de atualização de um recurso. Uma atualização deve especificar qual recurso será modificado, utilizando a anotação @Filter ou o elemento *resource* do @CustomMethodStep. No exemplo apresentado, o *resource* define que o artigo a ser atualizado possui um identificador igual ao do parâmetro *id* do caminho do pedido. Caso o artigo exista, o *complexWrite* indica que o atributo *views*, do recurso em questão, receberá um novo valor, resultante da execução da função definida em *function* com os parâmetros estabelecidos em *functionParams*. A função do exemplo retorna o inteiro seguinte ao parâmetro, que, neste caso, corresponde ao número de visualizações do artigo no momento da execução da operação. Em essência, o passo apresentado no exemplo define o comportamento da atualização do número de visualizações de um artigo.

Listagem 4.21: Exemplo do uso do @ComplexWrite.

```
@CustomMethodStep(operationType = HTTPVerb.PUT, resource = "articles/$request.path#/id",
       write = @Write(
              complexWrite = {
                     @ComplexWrite(attribute = "views", function = "(fn [v] (+ 1 v))",
                                   functionParams = "$resource#/views")
              }
          ))
```

4.2.2.5 Condition

Muitas vezes, as aplicações REST possuem operações cujos resultados internos dependem de uma ou mais condições. As condições podem fazer com que uma operação interna não seja executada, ou que uma seja executada em vez de outra. Para representar estas situações, a linguagem oferece a anotação @Condition, apresentada na Listagem 4.22, cuja finalidade é estabelecer uma condição que determina se um passo é executado. A condição determinada na anotação pode ser definida de duas formas distintas: usando um booleano presente no pedido do utilizador, com o elemento *bool*; ou usando uma

função que devolve um booleano, com os elementos *function* e *functionParams*. Pode-se, também, especificar se a condição deve ser verdadeira ou falsa, através do elemento *result*.

Listagem 4.22: Código da anotação @Condition.

```
public @interface Condition {
    String bool() default "";
    String function() default "";
    String[] functionParams() default {};
    boolean result() default true;
}
```

Usando o exemplo da atualização do número de visualizações de um artigo, apresentado na Listagem 4.21, pode ser adicionada uma condição para que não sejam contabilizadas as vezes em que o autor do artigo acede ao mesmo, considerando assim apenas as visualizações de terceiros. Na Listagem 4.23 é apresentada a adição desta condição, usando o elemento *condition*. A função definida devolve um valor booleano que indica se duas *Strings* são iguais ou não. Os parâmetros usados são os identificadores do utilizador que lê o artigo, presente no corpo do pedido, e do autor, presente no recurso do artigo. Portanto, a função indica se o utilizador que quer ler o artigo é o autor do mesmo. Como o componente *result* tem o valor *false*, este passo do CustomMethod só será executado quando o resultado da função for negativo, o que significa que o número de visualizações só altera quando não é o autor a ler o artigo.

Listagem 4.23: Exemplo do uso da @Condition.

```
@CustomMethodStep(operationType = HTTPVerb.PUT, resource = "articles/$request.path#/id",
       condition = @Condition(function = "(fn [user author] (= user author))",
                         functionParams = {"$request.body#/userId", "$resource#/author"},
                         result = false),
       write = @Write(
              complexWrite = {
                     @ComplexWrite(attribute = "views", function = "(fn [v] (+ 1 v))",
                                   functionParams = "$resource#/views")}))
```

4.2.2.6 Localização dos dados usados no CustomMethod

Na maioria dos casos, como os apresentados nas listagens anteriores, os passos do CustomMethod necessitam de aceder a valores definidos no pedido, na resposta, nos recursos ou em passos anteriores. Para indicar onde estão localizados estes valores, foi definida uma linguagem regular, que consiste numa sequência limitada de concatenações de *Strings*.

Para indicar onde estão localizados estes valores, foi ampliada a linguagem regular, definida por Simoes et al., que consiste numa sequência limitada de concatenações de *Strings*.

A Figura 4.2 estabelece as combinações possíveis de acesso a valores fora do contexto de cada passo. A *String* deve começar com o símbolo do *dollar* "$", para assinalar o início. A seguir, é especificado onde se encontra o valor, que pode estar no pedido ("*request*"),

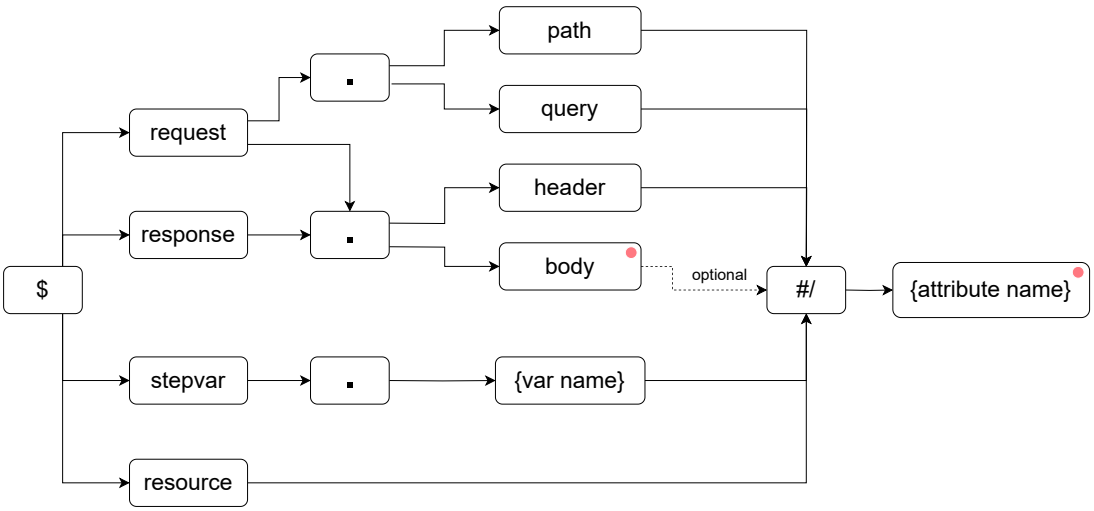


Figura 4.2: Visualizador da expressão regular que define a localização dos dados usados no CustomMethod.

na resposta ("*response*"), numa variável definida num passo anterior ("*stepvar*"), ou no recurso acedido no passo atual ("*resource*"). Nos três primeiros casos, a *String* é seguida de um ponto ".", enquanto que, no último, é adicionada a sequência especial "#/".

Depois, no caso do valor se encontrar no pedido, pode estar no caminho, como parâmetro identificador ("*path*") ou parâmetro de pesquisa ("*query*"); no corpo ("*body*"); ou, num cabeçalho ("*header*"). Estes dois últimos também se aplicam quando o valor se encontra na resposta do pedido. Por fim, adiciona-se a sequência especial "#/" e o nome do atributo ao qual se deseja aceder. A isto, é acrescentado um caso particular, onde o corpo da resposta não é um recurso, mas sim um tipo de dado primitivo (e.g., String, int, char) ou uma coleção (e.g., List, Set). Neste caso, a *String* final é apenas "*$response.body*", pois o corpo só tem um valor.

A maioria destes casos já tinham sido definidos na versão inicial da linguagem. Nesta nova versão, acrescentámos os dois casos seguintes.

Caso o valor esteja numa variável definida num passo anterior, basta adicionar o nome da variável, seguido da sequência especial "#/", e, por último, o nome do atributo.

O último caso possível ocorre quando o valor é um atributo do recurso acedido no passo atual. Nesta situação, basta adicionar o nome do atributo desejado.

Na Listagem 4.24 estão apresentados alguns exemplos de como aceder a valores externos no CustomMethod. Cada linha da listagem contém, respetivamente, uma *String* referente a:

1. O parâmetro "*userId*" do caminho do pedido.

2. O cabeçalho "*Content-Type*" da resposta.

3. O atributo "*name*" da variável "*user*", definida num passo anterior.

4. O atributo "*id*" do recurso acedido no passo.

Listagem 4.24: Exemplo de Strings de acesso a variáveis externas.

```
1 "$request.path#/userId";
2 "$response.header#/Content-Type";
3 "$stepvar.user#/name";
4 "$resource#/id";
```

4.2.2.7 StatusCodeSequence

As anotações descritas anteriormente são capazes de especificar os passos internos das operações de uma aplicação REST. Assim, quando uma operação é executada, é possível saber que alterações foram feitas no estado da aplicação, sem necessidade de ter acesso a este. A verdade, é que esta afirmação só é certa se a operação for bem-sucedida e todos os passos também forem concluídos com sucesso. Porém, esta é uma visão limitada das aplicações REST, visto que uma operação pode retornar um código de erro, e, mesmo assim, alterar o estado. Além disso, os passos de uma operação podem não ser todos bem-sucedidos, e, no entanto, a operação pode retornar um código de sucesso. Por isso, é necessário definir qual deve ser o resultado de cada passo de uma operação, dependendo do resultado final desta.

A anotação @StatusCodeSequence permite definir, para um dado resultado final da operação, o resultado esperado de cada passo. Como os passos representam operações REST simples, os resultados são simplesmente códigos de estado HTTP. Como pode ser observado na Listagem 4.25, a anotação só necessita de dois elementos: o *opStatus*, que representa o código de estado HTTP do resultado da operação da aplicação; e, o *stepsStatusSeq*, que representa a sequência de códigos de estado HTTP dos passos, que leva ao resultado determinado da operação.

Listagem 4.25: Código da anotação @StatusCodeSequence.

```
1 public @interface StatusCodeSequence {
2     int opStatus();
3     int[] stepsStatusSeq();
4 }
```

Como os passos podem ser interdependentes, ou depender de uma condição, aos códigos de estado HTTP, são adicionados três códigos mais para tratar destas situações: -1, 0 e 1. A seguir são apresentados os tipos de código que cada passo suporta, assim como, o significado de cada um:

- 2XX - O passo teve sucesso e as alterações feitas ao estado mantêm-se. No caso em que a operação é bem-sucedida, e na definição dos resultados, utilizando a anotação, não existe nenhum código específico para indicar sucesso, por definição, todos os passos terão o código 200.

- 404 - O recurso ao que se pretende aceder no passo não existe.
- 409 - O recurso ao que se pretende aceder no passo já existe.
- -1 - O passo é ignorado. Normalmente, este código é usado quando o resultado final da operação é de erro (i.e., 4XX). Isto deve-se a que existe sempre um passo final que despoleta o erro na operação, e quando esse passo não é o último da sequência, os restantes devem ser ignorados.
- 1 - A operação definida no passo é executada, mas o seu resultado é ignorado. Por vezes, a execução de um passo não pode ser ignorada, mas o resultado desta não afeta o resultado final da operação, nem o resultado dos passos seguintes. Ou seja, o comportamento da operação não se altera, independentemente do resultado do passo em questão.
- 0 - A condição não foi cumprida e o passo foi ignorado. Este código é usado quando é definida uma condição para que o passo seja executado. Se o passo tiver uma condição, mas o código for 2XX, 404, ou 409, significa que a condição foi cumprida e que um destes deve ser o resultado.

Na Listagem 4.26, encontra-se um exemplo de como é usada esta anotação. A operação pode retornar 409, 403 ou 204, e está dividida em 3 passos, visto que a sequência de códigos dos passos tem três elementos. Quando a operação retorna 409, significa que o primeiro passo foi executado com sucesso, mas o segundo falhou porque um recurso já existia. Como o passo que despoletou o erro na aplicação não é o último da sequência, os restantes são ignorados, neste caso, só existe um passo restante. Por outro lado, quando a operação retorna 403, significa que os dois primeiros passos foram bem-sucedidos, mas que a condição do último não foi cumprida. Por último, quando a operação devolve 204, isso implica que o primeiro e último passos foram concluídos com sucesso, e o segundo passo falhou devido à ausência de um certo recurso.

Listagem 4.26: Exemplo do uso do @StatusCodeSequence.

```
1  @CustomMethod(
2          operation = {...},
3          statusCodeSeq = {
4                  @StatusCodeSequence(opStatus = 409, stepsStatusSeq = {200, 409, -1}),
5                  @StatusCodeSequence(opStatus = 403, stepsStatusSeq = {200, 200, 0}),
6                  @StatusCodeSequence(opStatus = 204, stepsStatusSeq = {200, 404, 200})
7          }
8  )
```

Com esta anotação, é possível redefinir o significado dos códigos de estado das respostas das aplicações REST, permitindo relatar os resultados dos passos internos das operações. Isto significa que um 404, em vez de simplesmente sugerir que um recurso não existe, pode, por exemplo, indicar que um recurso é criado quando outro não existe. Assim, cada aplicação pode definir o significado do código do resultado das suas operações.

4.2.2.8 ResponseCondition

O conteúdo do resultado das operações com semântica personalizada é definido pelos desenvolvedores das diferentes aplicações. Como estas não seguem as regras impostas pela arquitetura REST, os dados apresentados nas respostas dos pedidos não podem ser inferidos. Por exemplo, duas aplicações distintas podem ter duas operações cujo comportamento é semelhante, como a criação de um utilizador, porém, as respostas podem ser completamente distintas. Numa aplicação, a resposta pode conter o recurso do utilizador criado, enquanto que, na outra, a resposta pode incluir apenas o identificador.

Para poder identificar os valores que os dados enviados na resposta de uma operação devem tomar, foi criada a anotação @ResponseCondition. Esta anotação permite que seja implementada uma função em Clojure, no elemento *function*, que define uma condição que deve ser cumprida após a execução da operação da API. Os parâmetros utilizados pela função são definidos no elemento *functionParams* e devem representar tanto os dados da resposta, como os valores que devem tomar.

Listagem 4.27: Código da anotação @ResponseCondition.

```
1 public @interface ResponseCondition { String function(); String[] functionParams(); }
```

A Listagem 4.28 mostra um exemplo da utilização desta anotação numa operação que devolve todos os utilizadores maiores de 18 anos. A condição definida no elemento *function* é verdadeira quando dois conjuntos são iguais. O elemento *functionParams* determina que os conjuntos são, por um lado, os utilizadores maiores de idade guardados na variável *adults*, e, por outro, os valores presentes na resposta. Deste modo, é determinado que a resposta desta operação contém apenas a lista de utilizadores maiores de idade.

Listagem 4.28: Exemplo do uso do @ResponseCondition.

```
1 @CustomMethod(operation = {@CustomMethodStep(
2        operationType = HTTPVerb.GET, resource = "users", storeInVar = "adults",
3        filter = @Filter(filterFunction = "(fn [x] (> x 18))",
4                         functionParams = {"$resource#/age"}))},
5    statusCodeSeq = {...},
6    responseCondition = @ResponseCondition(
7        function = "(defn f [l1 l2] (= (set l1) (set l2)))",
8        functionParams = {"$stepvar.adults", "$response.body"} ))
```

O uso desta anotação permite ao JepREST verificar se os valores presentes nas respostas das operações correspondem aos respetivos valores no estado gerado pelas operações previamente executadas.

4.3 Geração de Workloads

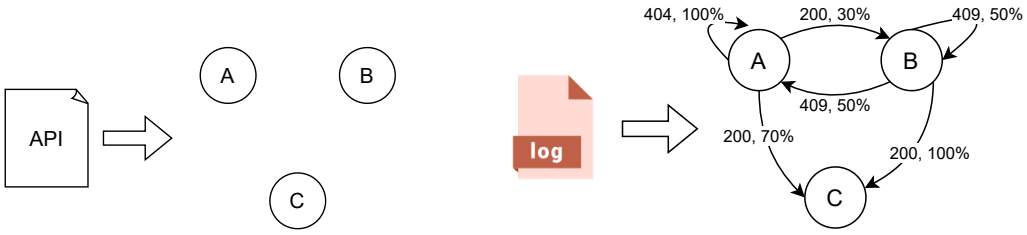
Para superar as limitações da definição manual dos testes, descritas na secção 4.1.4.1, criámos uma ferramenta que gera uma cadeia de Markov de transições das operações

especificadas na API [2]. Uma cadeia de Markov é um grafo no qual a probabilidade de transição para um estado depende somente do estado atual [47]. Num sistema REST, a execução de uma operação depende da operação executada anteriormente e do resultado dela. Portanto, os vértices do grafo são compostos pelos identificadores das operações e as arestas definem que operações podem ser executadas a seguir, dependendo do código da resposta, e com que probabilidade. Para obter todas estas informações e poder formar o grafo, necessitamos de dados que indiquem como é usada a aplicação. Nesta solução, usamos os *logs* da aplicação para obter tais informações.

Primeiro, esta ferramenta lê a API da aplicação e gera um grafo vazio (i.e. sem arestas), com tantos vértices quanto o número de operações disponibilizadas na API, sendo que cada vértice é único e corresponde exclusivamente ao identificador de uma operação distinta, como ilustrado na Figura 4.3b, onde A, B e C representam as operações presentes na API. Depois, a ferramenta lê o ficheiro de *logs*, que o utilizador indica, para, no fim, preencher o grafo, como exemplificado na Figura 4.3b. O ficheiro de *logs* é lido linha a linha e deve seguir o fomato apresentado na Listagem 4.29. Cada linha que representa um pedido ou uma resposta contém o IP do cliente e várias informações sobre o pedido ou a resposta, como o método HTTP, o caminho, a *query*, o *status code*, etc., e o resto das linhas são ignoradas. Os pedidos de cada cliente apresentados nos *logs* devem exibir um comportamento sequencial da aplicação, logo, se a última aparição de um cliente for um pedido, antes da resposta a esse pedido não pode haver nenhum outro pedido feito por esse cliente, ou seja, os clientes só devem efetuar novos pedidos depois de receber uma resposta. Isto deve-se a que a ferramenta, após encontrar um pedido do cliente, interpreta a próxima aparição do mesmo como uma resposta. Um caso diferente é quando aparecem pedidos seguidos de diferentes clientes, aqui não há problema nenhum, pois os *logs* de cada cliente são tratados à parte, ainda que estando no mesmo ficheiro.

Uma vez interpretado o ficheiro de *logs*, a ferramenta preenche o grafo com as informações retiradas do mesmo. Para cada operação da API, verifica-se se existem operações que a sucederam. Caso não exista nenhuma, o grafo não sofre qualquer alteração. Caso exista uma ou mais, são adicionadas as devidas arestas, que terão como fonte o vértice em questão e como fim o vértice respetivo à operação que a sucedeu. A estas novas arestas vão ser adicionados pesos condicionais. Um peso condicional é a probabilidade de uma aresta ser escolhida para percorrer o grafo quando uma condição é cumprida. Neste caso, a condição é o código da resposta da operação respetiva ao vértice. Para definir a probabilidade das diferentes arestas com a mesma condição, dividimos o número de vezes que cada operação foi executada, depois da resposta com um determinado código, pelo número total de pedidos efetuados após a resposta com esse código. Outra informação relevante, que também é calculada, é a probabilidade de cada vértice ser o primeiro na sequência. Isto serve para que quem utilize esta ferramenta possa começar com as operações que os utilizadores reais da aplicação começam. O cálculo é, basicamente, a divisão do número

[2] https://github.com/rrive/workload-generator



(a) Primeira fase. (b) Segunda fase.

Figura 4.3: Fases da construção do grafo de transições das operações.

de vezes em que cada operação foi a primeira a ser executada por diferentes clientes pelo número de sessões existentes.

As operações que o grafo oferece são duas. A primeira, retorna o identificador de uma operação quando ainda não foi executada nenhuma pelo respetivo cliente, e é usada para saber qual o primeiro pedido a fazer à aplicação. A segunda, retorna o identificador de uma operação sabendo a resposta do pedido executado anteriormente pelo cliente. Caso esta última não retorne nenhuma informação, a sessão chegou ao fim. Com apenas estas duas operações, é possível gerar casos de uso realistas em que as operações executadas dependem dos resultados das anteriores.

Listagem 4.29: Exemplo do log de um pedido e da sua resposta.

```
1 [1705:192.168.0.2] Tue Aug 13 13:41:56 UTC 2024 - GET users/john params={{id=[john]}}
      ↪ query={pwd=123}
2 [1705:192.168.0.2] Tue Aug 13 13:41:56 UTC 2024 - GET users/john 200 params={{id=[john]}}
      ↪ query={pwd=123}
```

Na Figura 4.3b, podemos ver um exemplo simples de um grafo gerado com esta ferramenta. O grafo tem três vértices, o que significa que a aplicação REST expõe três operações, A, B e C. O passo inicial, é pedir ao grafo a primeira operação que vamos executar. Imaginando que a operação devolvida foi A, pedimos à aplicação para executar A. Agora, com o resultado do pedido que acabámos de executar, pedimos ao grafo que nos indique que operação executar a seguir. Se o código da resposta de A for 200, então o grafo devolve a operação B, com 30% de probabilidade, ou C, com 70%. Se o código for 404, então, o grafo dirá que devemos executar A, outra vez. No caso da operação seguinte ser B, e o resultado desta, 409, a probabilidade da próxima operação ser A ou B é igual, ou seja, 50%. Caso a operação seguinte seja a C, como não é um vértice fonte, é a ultima operação a ser executada e a sessão acaba. Este é o comportamento que os clientes do JepREST seguem para simular a interação entre um cliente real e a aplicação em teste.

Esta ferramenta ainda está numa fase inicial, o que quer dizer que ela própria tem as suas limitações. A mais óbvia, é que não pode ser usada quando não temos acesso aos *logs* da aplicação em teste. Neste caso, devemos recorrer à abordagem apresentada por Simoes et al. para gerar as operações. As outras limitações têm a ver com a simplicidade desta solução. Numa aplicação REST, para determinar que operação executar, não só

importa o código da resposta da operação anterior, como também os dados que vêm no corpo da resposta e outras informações relevantes, como o número de pedidos executados anteriormente. Estas são otimizações para serem realizadas em trabalhos futuros.

5

Implementação da Solução

5.1 JepREST

Nesta secção é apresentada a implementação da segunda versão da ferramenta de teste JepREST.

5.1.1 Geração do Código de Teste

Para testar uma API REST é necessário que o JepREST tenha conhecimento sobre as operações que a API oferece. Mais especificamente, o JepREST tem de saber como devem ser realizados os pedidos às operações e como devem ser gerados os dados enviados nos mesmos. Para isto, a ferramenta usa um gerador de código que lê os documentos da especificação dos serviços da API e gera um conjunto de métodos em Clojure que serão usados pelo JepREST para gerar os dados e realizar as invocações dos pedidos à aplicação.

Cada operação presente no documento interface vai ter dois métodos associados: um que realiza o pedido a essa operação da aplicação e o outro que gera os dados necessários para executar esse pedido.

5.1.1.1 Definição dos Pedidos e Respostas

O gerador de código, para definir como é realizado um pedido, verifica as anotações da biblioteca *javax.ws.rs*. Este gerador utiliza as anotações @GET, @POST, @PUT, @PATCH e @DELETE para determinar o método HTTP que deve ser usado em cada pedido. Para definir o caminho que deve constar no pedido, é utilizada a anotação @Path, e, para indicar se o pedido deve conter um corpo, é utilizada a anotação @Consumes. Para executar os pedidos HTTP, o código gerado utiliza a biblioteca *clj-http* [8].

A Listagem 5.1 mostra a primeira parte do método gerado que define uma operação chamada "*upload*", que, como o nome indica, realiza um *upload* de um ficheiro. O método recebe como parâmetros o URL base da aplicação (*conn*) e uma operação do Jepsen (*op*). Esta operação do Jepsen contém todos os dados necessários para realizar o pedido, e é criada pelo outro método gerado associado à mesma operação. Na terceira linha, verifica-se que a operação é um POST, pois é utilizada a função "*post*" da biblioteca *clj-http*. Esta

função recebe como argumentos o URL completo do pedido e o resto de dados enviados no mesmo, como o corpo e/ou os cabeçalhos. O URL completo é formado pela concatenação do parâmetro *conn* do método *upload*, que representa a união entre o protocolo, o domínio e, opcionalmente, a porta do servidor que vai receber o pedido (e.g., *http://155.2.116.23:8080*), com o caminho */rest/blobs/* e com o parâmetro do caminho, cujo valor está definido na operação do Jepsen, mais especificamente, no campo *:blobId* do elemento *:path*, dentro do *:input*, que, por sua vez, está contido dentro do campo *:value*. Nas duas linhas seguintes está o mapa que representa os dados enviados no pedido. Na linha 4, no campo *:content-type*, é estabelecido que o tipo de dados enviados no pedido é *"application/octet-stream"*, e, na linha 5, no campo *:body*, é definido o corpo do pedido, que também está especificado na operação do Jepsen *op*, dentro do campo *:input* do elemento *:value*. A resposta devolvida pela aplicação é associada à variável local *response*, que será processada na segunda parte do método.

Listagem 5.1: Exemplo da primeira parte de um método que realiza pedidos à aplicação.

```
1 (defn upload [conn op]
2     (let [response
3             (http/post (str conn "/rest/blobs/" (:blobId (:path (:input (:value op)))))
4                        {:content-type "application/octet-stream"
5                         :body (:body (:input (:value op)))})
6           ] ...)
```

A segunda parte deste método trata da resposta da aplicação e atualiza a operação do Jepsen, enviada como parâmetro, para que contenha a informação da resposta fornecida pela aplicação. Para isto, o gerador de código verifica os códigos de estado que podem ser retornados pela operação, especificados nas anotações @ApiResponse do OpenAPI, e determina o *media type* utilizado no corpo da resposta, caso a anotação @Produces esteja presente. A Listagem 5.2 apresenta a implementação desta segunda parte do método. Como a resposta pode variar dependendo de ser um sucesso ou um erro, o método gerado define três resultados diferentes.

No primeiro cenário, o código da resposta indica sucesso. Neste contexto, o campo *:type* da operação do Jepsen, que no momento da execução deste método tem o valor *:invoke*, é alterado para *:ok*. Além disso, a resposta obtida na primeira parte do método também é inserida, e, quando o corpo da resposta é um JSON, é utilizada a função *parse-string* da biblioteca Cheshire [7]. Esta função, ao receber o segundo parâmetro igual a *true*, transforma as chaves do objeto JSON em *keywords* do Clojure, o que é necessário porque o nosso modelo do Knossos assume que as chaves dos mapas são sempre *keywords*.

O segundo cenário refere-se a situações em que a resposta contém um código de erro esperado. Neste caso, o corpo da resposta não é relevante para o JepREST, uma vez que a ferramenta utiliza os códigos de estado HTTP para verificar o resultado da operação, ou das operações internas. Assim, com exceção da codificação do JSON, são feitas as mesmas alterações à operação do Jepsen que no primeiro cenário.

No terceiro e último cenário, a resposta não inclui um código de estado previamente definido na API. Quando isto acontece, o JepREST não consegue determinar se o resultado da operação está correto ou se apresenta incoerências. Por conseguinte, o valor do campo *:type* é alterado para *:fail*, e um novo campo *:msg* é adicionado, contendo a informação presente na resposta.

Listagem 5.2: Exemplo da segunda parte de um método que realiza pedidos à aplicação.

```
(defn upload [conn op]
  (let [response ...]
    (if (int-in-range? 200 300 (:status response))
        (assoc op :type :ok, :value (assoc (:value op) :output {:status (:status response),
               :headers (:headers response),
               :body (try (json/parse-string (:body response) true)
                     (catch JsonParseException e (:body response)))}))
    (if (or (= (:status response) 403) (= (:status response) 409))
        (assoc op :type :ok, :value (assoc (:value op) :output {:status (:status response),
                  :headers (:headers response), :body (:body response)}))
        (assoc op :type :fail, :msg (assoc (:value op) :output {:status (:status response),
                  :headers (:headers response), :body (:body response)}))
))))
```

Antes da definição dos métodos explicados acima, o gerador de código define quais são os atributos que identificam os recursos. Esta informação é armazenada num mapa onde a chave é o nome do recurso e o valor é a *keyword* do Clojure relativa ao atributo identificador. Este mapa é apenas usado durante a fase de avaliação dos resultados, e, mais especificamente, na verificação do resultado de operações que cumprem a semântica REST. Isto deve-se a que quando um pedido tem um identificador de um recurso no caminho, o *checker* não sabe qual é o atributo que deve ter esse valor, pois o nome do atributo que identifica o recurso não é especificado no pedido. Na primeira linha da Listagem 5.3, pode-se ver como o identificador do recurso *users* é *userId* e o do recurso *blobs* é *blobId*. Para determinar qual é o atributo identificador, é usada a anotação @Id no atributo correspondente do documento objeto, como apresentado na sétima linha da Listagem 4.4.

A seguir a isto, o gerador de código cria as listas que guardam os parâmetros dependentes entre as operações. Estas são as listas que são acedidas durante a fase de teste para realizar os pedidos de operações dependentes. Como as listas são usadas pelos clientes, que executam pedidos concorrentes, a lista é uma CopyOnWriteArrayList, para garantir *thread-safety*. O nome das listas é a concatenação da *String "listInfoOf"* com o nome da operação geradora, que é a que contém a anotação @Link. Na Listagem 5.3 são apresentadas duas destas listas. A primeira é utilizada pelas operações dependentes da operação *createUser*, e a segunda, pelas operações dependentes da operação *upload*. Ambas listas podem ser acedidas por uma única operação, quando esta depende das duas operações geradoras.

Listagem 5.3: Código que cria as listas de parâmetros dependentes.

```
(def id-of-resource (doto (HashMap.) (.put "users" :userId) (.put "blobs" :blobId)))
(def listInfoOfcreateUser (CopyOnWriteArrayList.))
(def listInfoOfupload (CopyOnWriteArrayList.))
```

Os parâmetros dependentes, como os identificadores dos recursos e palavras-passe, ficam armazenados nas listas de parâmetros dependentes. A seguir, é explicado como são adicionados, atualizados e apagados os parâmetros dependentes das listas. Para exemplificar cada caso, é usada a lista gerada pela operação *createUser*, que, como o nome indica, cria um utilizador.

Quando um recurso é criado na aplicação, ou seja, quando é executada uma operação geradora, o JepREST adiciona à lista correspondente os dados relativos aos atributos relevantes para as operações dependentes. Como os parâmetros dependentes podem ser vários, e diferentes clientes podem modificá-los ao mesmo tempo, é utilizado um ConcurrentHashMap para armazená-los. O exemplo dado na Listagem 5.4 mostra como são adicionados os parâmetros dependentes (*userId* e *pwd*) na lista correspondente à operação geradora (*createUser*). O valor que os parâmetros tomam são os que são enviados no corpo do pedido.

Listagem 5.4: Código de adição de um elemento à lista de parâmetros dependentes.

```
(.add listInfoOfcreateUser (doto (ConcurrentHashMap.)
          (.put :userId (:userId (:body (:input (:value op)))))
          (.put :pwd (:pwd (:body (:input (:value op)))))))
```

Quando é executada uma operação que remove um recurso que, por sua vez, contém parâmetros dependentes, esta remoção deve ser espelhada na lista correspondente. Deste modo, os clientes não realizarão pedidos irrelevantes para o JepREST, pois esse pedidos não seriam realistas. A Listagem 5.5 mostra como é efetuada a remoção dos elementos do recurso apagado na aplicação. A lista dos parâmetros dependentes *listInfoOfcreateUser* é filtrada usando os parâmetros *userId* e *pwd* presentes no *path* e na *query* do caminho, respetivamente. O filtro é utilizado devido a que o mapa pode conter mais elementos e a remoção da lista deve conter o mapa completo. Após a filtragem, o respetivo mapa é removido da lista.

Listagem 5.5: Código de remoção de um elemento da lista de parâmetros dependentes.

```
(.removeAll listInfoOfcreateUser
   (->> listInfoOfcreateUser
        (r/filter #(.containsAll (.entrySet %) (.entrySet (doto (ConcurrentHashMap.)
                          (.put :userId (:userId (:path (:input (:value op)))))
                          (.put :pwd (:pwd (:query (:input (:value op)))))
)))) (into [])))
```

Quando um recurso é atualizado, se os atributos modificados são parâmetros dependentes, a respetiva lista também deve ser atualizada para manter os valores iguais aos do estado da aplicação. Para isto, é definido o atributo que identifica o respetivo elemento da

lista, i.e., a entrada do mapa cujo valor é único na lista, que geralmente é o identificador do recurso. Depois, a lista é percorrida e, quando esse elemento é encontrado, são atualizadas as respetivas entradas. No caso da Listagem 5.6, é atualizada a palavra-passe do utilizador com o identificador especificado no *path*. O valor da nova palavra-passe é o apresentado no corpo do pedido

Listagem 5.6: Código de atualização de um elemento da lista de parâmetros dependentes.

```
1 (doseq [map listInfoOfcreateUser
2       :when (or (= (.get map :userId) (:userId (:path (:input (:value op))))))]
3  (.put map :pwd (get (json/parse-string (:body response) true) :pwd)))
```

O código que gera os métodos que efetuam os pedidos à aplicação foi implementado na primeira versão do JepREST, mas sofreu algumas alterações nesta segunda versão. Realizámos duas pequenas modificações na segunda parte do método, visíveis nas linhas 3 e 8 da Listagem 5.2, para que os códigos de resposta esperados fossem determinados pela linguagem descrita na secção 4.2, quando aplicável, e para que o valor do corpo da resposta fosse formatado em JSON apenas quando possível. A única adição foi a geração do código que atualiza os elementos das listas de parâmetros dependentes, exemplificado na Listagem 5.6.

5.1.1.2 Definição dos Parâmetros e Semântica dos Pedidos

O gerador de código, para cada operação do documento interface, interpreta as anotações presentes e gera um método que cria uma operação do Jepsen, onde é contida toda a informação necessária para a geração dos parâmetros enviados nos pedidos. Para além disto, a operação criada no método também guarda a sequência de suboperações que definem o comportamento interno da operação da API, de modo a que o *checker* possa saber as alterações feitas no estado e, assim, verificar a correção da aplicação. O nome deste método é formado pela concatenação do nome da operação da API com a *String* "*Data*" (e.g., "createUserData").

Parâmetros Dependentes

Uma operação pode depender de uma ou mais operações, de várias execuções da mesma, ou de várias execuções de diferentes operações. Devido a isto, estes métodos definem um índice para cada lista da qual dependem. Isto quer dizer que, se uma operação X depende de duas execuções diferentes da operação Y e de uma da operação Z, o método que cria a operação do Jepsen relativo à operação X vai ter dois índices para a lista *listInfoOfY* e um para a *listInfoOfZ*. Os nomes dos índices são gerados usando a mesma lógica dos nomes das listas de parâmetros dependentes. Estes são formados pela concatenação da *String* "*indexOf*" com o nome da operação correspondente (e.g., *indexOfZ*). Se a operação depende de duas ou mais execuções de uma operação geradora, ao nome do índice, é

adicionado um identificador definido para cada execução, que é especificado na anotação @ExtendLinkParameter (e.g., *indexOfY1* e *indexOfY2*).

Para explicar o uso dos índices, vamos usar a operação de seguir um utilizador numa rede social simples. Esta operação depende da operação que cria os utilizadores. Em particular, a operação de seguir recebe três parâmetros: o identificador do utilizador que executa a operação; a palavra-passe do mesmo, para motivos de autenticação; e o identificador do utilizador que vai ser seguido. Portanto, como a ideia da operação é que um utilizador siga outro que não ele, a operação depende de duas execuções diferentes da operação de criar utilizador. Então, o método vai gerar dois índices aleatoriamente, um para identificar o elemento da lista que contém o identificador e a palavra-passe referentes ao utilizador que efetuou o pedido, e outro para identificar o elemento da lista que contém o identificador do utilizador seguido. Estes índices são definidos no início do método, como pode ser visto na Listagem 5.7, onde está definido o método *followData* referente à operação de seguir um utilizador chamada *follow*.

Listagem 5.7: Código que define onde são criados os índices das listas

```
1 (defn followData [_ _]
2 (let [clonelistInfoOfcreateUser (.clone listInfoOfcreateUser)
3       indexOfcreateUser1 ((... (.nextInt random (.size clonelistInfoOfcreateUser)))
4       indexOfcreateUser2 ((... (.nextInt random (.size clonelistInfoOfcreateUser)))] ...)
```

Estes índices são utilizados mais à frente no método, quando são determinados os valores que os parâmetros do pedido vão tomar. Esta informação é definida dentro do campo *:value* da operação do Jepsen, mais especificamente, no campo *:input*. Os parâmetros dependentes podem ser usados nos parâmetos do caminho, na *query* e no corpo do pedido. O mais comum é que no corpo só sejam definidos parâmetros independentes, mas não há nenhuma limitação quanto a isso. Na Listagem 5.8 está apresentada a segunda parte do método *followData*. Neste caso, o caminho tem dois parâmetros chamados *userId1* e *userId2*, que representam os identificadores do utilizador que realiza o pedido e do utilizador que é seguido por ele, respetivamente. Por isso, para usar dois identificadores de utilizadores distintos é usado um índice para cada um. Agora, como o parâmetro de pesquisa é a palavra-passe associada ao primeiro utilizador, é usado o mesmo índice, deste modo são usados os dados armazenados na lista correspondentes ao mesmo recurso.

Listagem 5.8: Exemplo de uma operação do Jepsen

```
1 ... {:type :invoke, :f :post, :value {:input {:body generateData("upload", nil, nil)
2   :path {:userId1 (...(.get (.get clonelistInfoOfcreateUser indexOfcreateUser1) :userId))
3          :userId2 (...(.get (.get clonelistInfoOfcreateUser indexOfcreateUser2) :userId))}
4   :query {:pwd (... (.get (.get clonelistInfoOfcreateUser indexOfcreateUser1) :pwd))}}}}))
```

A parte do gerador de código que criava estes métodos na versão anterior do JepREST permitia tratar apenas casos em que um único parâmetro da lista era utilizado. Isto apresentava várias limitações, pois não era possível enviar, num mesmo pedido, dois parâmetros com valores correspondentes ao mesmo recurso. Por exemplo, não era possível

definir que os valores dos parâmetros *pwd* e *userId1* na Listagem 5.8 correspondessem ao mesmo recurso. Além disso, não era permitido que dois parâmetros referentes ao mesmo atributo de um recurso (e.g., *userId1* e *userId2*) assumissem valores distintos, gerados por diferentes execuções da operação geradora. Nesta versão, adicionámos o uso dos índices descritos acima, o que elimina as limitações apresentadas na versão anterior.

Parâmetros Independentes

Para gerar os valores dos parâmetros independentes, são usadas as bibliotecas PETIT e Faker. A biblioteca PETIT oferece funções que permitem gerar *Strings*, como a *generateString(min, max)*, que gera uma *string* aleatória com o tamanho dentro dos limites *min* e *max*, a *generateFromRegex(regex)*, que gera uma *String* que satisfaz a expressão regular dada como parâmetro, e a *generateFromRegex(regex, min, max)*, que é a junção das duas funções anteriores. Esta biblioteca também tem duas funções que permitem gerar inteiros ou *longs* dentro de uns limites, chamadas *generateInRange(min, max)* e *generateLong(min, max)*. A biblioteca oferece mais funções, mas estas são as únicas suportadas pelo JepREST. Por outro lado, para gerar valores realistas dos parâmetros, o JepREST suporta todas as operações da adaptação da biblioteca Faker para a linguagem Clojure [28]. Estas funções geram strings com conteúdos semelhantes aos que utilizadores reais costumam usar.

Caso as funções destas bibliotecas não sejam suficientes, seja porque não geram os tipos de dados necessários, ou por falta de complexidade, usando a anotação @Values nos documentos interface e/ou objeto, pode ser definida uma função em Clojure que será utilizada para determinar o valor do parâmetro correspondente.

Estas funções também podem ser usadas para gerar valores de parâmetros de caminho ou de pesquisa, mas geralmente são utilizadas na geração dos dados enviados no corpo do pedido. A Listagem 5.9 mostra a função que gera os dados do corpo de um pedido, chamada "*generateData*". Esta, recebe como parâmetros o nome da operação da API, um mapa com listas de parâmetros dependentes e um mapa com os índices necessários para aceder a essas listas. O exemplo apresentado na listagem corresponde ao corpo do pedido relativo à operação *updateUser*. Este exemplo apresenta as quatro diferentes formas de gerar os valores dos parâmetros. O parâmetro *userId* representa o identificador do utilizador que vai ser modificado, por isso, é usada a lista *listInfoOfCreateUser*. A nova palavra-passe do utilizador, representada pelo parâmetro *pwd*, é gerada através de uma operação da biblioteca PETIT, que cria uma *String* de entre seis e vinte caracteres, composta por letras e números. O parâmetro *name* usa a função *first-name* do Faker para gerar um nome próprio realista. Por fim, o parâmetro *email* usa uma função definida pelo utilizador do JepREST, que devolve o primeiro elemento da lista de *emails* gerada pelo Faker.

O código que gera a operação *generateData* foi implementado na versão anterior do JepREST. Nesta versão, atualizámos esse mesmo código para que a função receba, para além do identificador da operação da API, dois mapas que vão conter as listas de parâmetros dependentes e os índices usados para aceder aos elementos dessa lista. Para os casos em

que no corpo é enviado um *array* de *bytes*, ou seja, quando a especificação do parâmetro contém a anotação @Blob, adicionámos uma função que gera o *array* especificado.

Listagem 5.9: Código que gera os valores do corpo dos pedidos

```
1 (defn generateData [method-name info-list-map index-map] (case method-name
2    "updateUser" { :userId (... (.get (.get (:listInfoOfcreateUser info-list-map)
3                                      (:indexOfcreateUser index-map)) :userId)),
4                   :pwd (PrimitiveTypes/generateFromRegex "[A-Za-z0-9]+" 6 20),
5                   :name (faker.name/first-name),
6                   :email (first (take 1 (faker.internet/emails)))} ...)
```

Semântica Personalizada

À operação do Jepsen, criada pelo método gerado, também é adicionada a informação relevante sobre as operações com funcionalidades personalizadas. O JepREST interpreta a linguagem descrita na secção 4.2 e transforma o conteúdo especificado para cada operação num mapa do Clojure. Este mapa é incluído no campo *:input*, dentro do *:value*, e é apelidado de *:custom-op*. O mapa inclui três campos: o primeiro representa a sequência de passos da operação (*:op-steps*), o segundo representa as sequências de resultados dos passos que levam a que a operação retorne os diferentes códigos de estado determinados (*:op-status-code-seq*), e o terceiro indica uma condição que deve ser cumprida pelos dados recebidos na resposta (*:response-condition*).

O campo *:op-steps* é uma lista onde cada elemento é um mapa que representa uma operação REST simples. Cada mapa contém as informações necessárias para que o *checker* saiba como deve ser alterado o estado em cada momento. Na Listagem 5.10, pode ser observado um exemplo desta lista (linhas 1-11). A lista *:op-steps* apresentada contém três elementos, o que significa que a execução desta operação da API é igual à execução em sequência das três operações da lista. Nenhuma destas operações tem a obrigatoriedade de representar uma operação real da aplicação, simplesmente servem para explicar as alterações que são efetuadas no estado. Todas estas operações têm presentes os campos *:method*, para determinar o verbo HTTP que estabelece o resultado de cada passo, e o *:resource*, que determina qual é o tipo do recurso afetado. Os campos *:filter* e *:write* definem quais são os recursos afetados e como são modificados, respetivamente. Pelo menos um destes dois últimos campos deve estar presente nos passos, podendo estar os dois em conjunto, como no terceiro passo do exemplo apresentado na listagem. O resto dos campos são usados dependendo do comportamento de cada passo, como o *:filter-by-id?*, que é usado quando o recurso afetado pelo passo é identificado utilizando o seu identificador; ou o *:store-in-var*, que define uma variável que, após executar as devidas alterações, guarda o recurso de modo a que possa ser usado nos passos seguintes.

O campo *:op-status-code-seq* também é uma lista; porém, neste caso, cada elemento é um mapa que, para cada código de estado HTTP que a operação pode retornar (*:op-status*), define qual é a sequência de resultados que os passos devem apresentar (*:steps-status-seq*).

O campo *:response-condition* é um mapa que contém apenas duas chaves. A chave *:response-condition-function* contém a função em Clojure que determina se os dados recebidos na resposta de um pedido são os esperados ou não. Já a chave *:response-condition-params* é a lista que indica os valores dos parâmetros usados na condição.

O exemplo apresentado na Listagem 5.10 é relativo à operação *updateUser* da API, onde apenas o email do utilizador é atualizado. Esta operação está dividida em três passos. O primeiro passo é um simples GET, que devolve, caso exista, o utilizador com o identificador igual ao enviado no parâmetro *userId* do caminho. O segundo passo representa um GET onde o recurso devolvido é o utilizador do primeiro passo, caso a palavra-passe enviada como parâmetro de pesquisa seja igual à que está armazenada no recurso. Estes dois passos servem para que o *checker* possa verificar se a autenticação devia falhar ou não. Isto acontece porque o modelo definido apenas conhece os códigos de estado 2XX, 404 e 409, devido a que o significado destes é fixo. No caso da operação *updateUser*, quando a palavra-passe enviada no pedido está errada, a aplicação retorna o código 403. Por isso, são definidos dois passos, um primeiro para verificar se o recurso existe, e um segundo para confirmar se o mesmo recurso tem a palavra-passe indicada. Quando o resultado da operação contém o código 404 significa que o primeiro passo falhou. Caso o resultado contenha o código 403, é confirmado que o primeiro passo teve sucesso, i.e. o recurso existe, mas que o segundo passo falhou, devido a que a palavra-passe enviada não está correta. No cenário em que ambos os passos anteriores têm sucesso, o recurso que representa o utilizador em questão fica guardado na variável *:usr*, e, a seguir, é avaliado o terceiro passo. Este último passo é identificado como um PUT que atualiza o email do utilizador. Segundo a função em Clojure definida no campo *:write-function*, o email do recurso só é modificado se o email enviado no corpo do pedido é um *gmail*.

Listagem 5.10: Exemplo da definição da semântica dos pedidos dentro de uma operação do Jepsen

```
... :value {:custom-op { :op-steps [
{:method :get, :resource "users", :filter-by-id? true,
   :filter {:simple-filter [{ :attribute nil :value "$request.path#/userId" }]}}
{:method :get, :resource "users", :store-in-var :usr,
   :filter { :simple-filter [{ :attribute :userId :value "$request.path#/userId" }
                              { :attribute :pwd :value "$request.query#/pwd" }]}},
{:method :put, :resource "users", :filter-by-id? true,
   :filter {:simple-filter [{ :attribute nil :value "$request.path#/userId" }]},
   :write {:complex-write [ {:attribute :email,
              :write-function "(fn [old new] (if (includes? new "@gmail.com") new old))",
              :write-function-params ["$stepvar.usr#/email" "$request.body#/email"]}]}}]
:op-status-code-seq [{ :op-status 404 :steps-status-seq [ 404 -1 -1 ] },
                     { :op-status 403 :steps-status-seq [ 200 404 -1 ] }]}, :input {...
```

Como a função definida no *:write-function* apenas determina se o atributo email do recurso é atualizado quando uma certa condição é verdadeira, o passo pode ser substituído pelo apresentado na Listagem 5.11. Neste passo alternativo, é utilizado o campo *:condition* para definir a condição que determina quando o passo deve ser executado ou ignorado; e

o campo *:write* é simplificado, pois, agora, quando a condição é cumprida, o email enviado no corpo pode ser inserido no recurso sem a necessidade de usar o *:complex-write*.

Listagem 5.11: Exemplo alternativo a um passo que representa um PUT com uma condição

```
{:method :put, :resource "users", :filter-by-id? true,
   :condition {:condition-function "(fn [new-email] (includes? new-email "@gmail.com"))",
               :condition-function-params ["$request.body#/email"], :result true}
   :filter {:simple-filter [{ :attribute nil :value "$request.path#/userId" }]},
   :write {:simple-write [{ :attribute :userId :value "$request.body#/email" }]}}
```

Todas estas informações são utilizadas pelo *checker* durante a fase de avaliação para recriar o estado da aplicação com a semântica definida pela API e, assim, poder verificar a correção do estado da aplicação após cada execução de uma operação.

Esta última parte do gerador de código foi implementada nesta versão do JepREST, uma vez que a versão anterior não suporta operações com semântica personalizada.

5.1.2 Definição e Execução dos Testes

Após a execução do gerador de código, o JepREST cria um teste do Jepsen que indica ao nó de controlo qual o tipo de clientes que será utilizado, como serão gerados os pedidos realizados pelos clientes e qual é o *checker* que avaliará a correção da aplicação com o modelo definido.

Como os clientes padrão do Jepsen apenas suportam operações de escrita e leitura a bases de dados, o JepREST usa um tipo de clientes personalizados. Estes clientes são instâncias da classe *ClientREST*, definida por Simoes et al. Esta classe estende a classe *Client* do Jepsen, logo, implementa todos os métodos dessa classe. O método mais relevante é o *invoke!*, cuja finalidade é realizar pedidos à aplicação em teste e avisar o nó de controlo quando a resposta é recebida. Quando um cliente recebe a operação do Jepsen criada pelo gerador de operações, que define todos os dados necessários para realizar o devido pedido, constrói o URL correspondente e envia o pedido à aplicação, utilizando o respetivo método do código gerado. No momento em que recebe a resposta, envia-a para o nó de controlo, que sinaliza ao gerador de operações que o cliente em questão está livre para executar uma nova operação.

As operações que os clientes vão executar na aplicação são definidas pelo gerador de operações. O JepREST suporta dois tipos de geradores: um que lê um ficheiro tipo Artillery e escolhe as operações predefinidas nos cenários; e um que gera as operações em tempo real.

5.1.2.1 Definição Manual dos Testes

Um ficheiro YAML tipo Artillery, como o apresentado na Listagem 4.6, pode conter vários cenários. Cada cenário tem um peso e pode incluir várias operações. Para escolher qual o cenário a executar, o JepREST gera um número aleatório entre 1 e a soma dos pesos. Usando o ficheiro apresentado na listagem, a soma dos pesos é 100, logo, o número gerado

vai estar entre 1 e 100. Caso o número seja menor que 30, será escolhido o segundo cenário, chamado "*Get user and short*", e as operações do Jepsen serão criadas pelos métodos do código gerado *getUserData* e *getShortData*. Caso o número esteja entre 30 e 100, será escolhido o primeiro cenário.

Para este tipo de testes, o JepREST usa dois geradores definidos pelo Jepsen: *phases* e *once*. O *phases* recebe um conjunto de geradores e, quando todos os clientes estão livres, usa-os para gerar as operações e atribuí-las aos clientes. O gerador *once* simplesmente gera uma operação uma única vez. O JepREST cria um conjunto de geradores *once* que usam os métodos do cenário escolhido para gerar as operações do Jepsen. O tamanho do conjunto é o número de clientes ou, se for menor, o número de operações num cenário. Este conjunto de geradores é passado como parâmetro ao gerador *phases*, junto com o gerador *sleep*, que indica o tempo que o *phases* espera até verificar se já pode gerar novas operações. No JepREST foi definido que o tempo de espera é 1 segundo. Portanto, só serão geradas operações a cada segundo, e se houver menos operações no cenário escolhido do que clientes, haverão sempre clientes inativos.

A estrutura do ficheiro YAML e o gerador o utiliza para definir as operações efetuadas pelos clientes num dado momento foram definidos na primeira versão do JepREST.

5.1.2.2 Definição Automática dos Testes

Se o utilizador do JepREST tem acesso aos *logs* da aplicação em teste, recomenda-se a utilização da ferramenta que gera o grafo de transições das operações da API. Este grafo permite que os clientes do JepREST executem sequências de operações similares às dos clientes reais da aplicação.

Para poder usar o grafo de transições das operações da API, criámos o gerador apresentado na Listagem 5.12, denominado *RestOpGenerator*. Este gerador recebe como parâmetro um vetor, denominado *gens*, cujos elementos são métodos do código gerado que criam as operações do Jepsen. Cada posição do vetor corresponde ao cliente com o mesmo identificador, visto que os clientes são identificados por inteiros de ordem crescente e começando no 0. Por exemplo, se o parâmetro *gens* é [*createUserData*, *uploadData*], significa que o cliente 0 executará a operação *createUser* com os parâmetros definidos na operação do Jepsen gerada pelo método *createUserData*, e que o cliente 1 efetuará a operação *upload* com os parâmetros presentes na operação do Jepsen criada pelo método *uploadData*. Inicialmente, o valor de cada posição do vetor *gens* é determinado pela operação *getFirstVertex* do grafo de transições.

Os geradores do Jepsen têm dois métodos: *op* e *update*. O método *op* é o responsável por enviar aos clientes as operações que devem executar. Internamente, o Jepsen executa este método de maneira contínua para verificar se podem ser atribuídas novas operações aos clientes. A responsabilidade por decidir quando uma operação pode ser enviada a um cliente recai sobre o gerador. No *RestOpGenerator*, quando um cliente recebe uma resposta da aplicação, é-lhe imediatamente associado o próximo pedido a ser executado.

Como pode ser observado na listagem, quando não existe nenhum cliente disponível (linha 5), ou seja, quando todos estão ocupados a efetuar pedidos à aplicação, o método *op* retorna um vetor cujo primeiro elemento é a *keyword :pending*, que indica ao Jepsen que ainda não existem clientes prontos para executar novos pedidos. O segundo elemento do vetor é o próprio gerador, sem modificações.

Por outro lado, quando há clientes livres, o gerador seleciona o primeiro da lista e executa a operação *op* no método correspondente ao utilizador no vetor *gens* (linhas 7-9). Vale mencionar que qualquer função que retorne uma operação do Jepsen pode ser considerada como um gerador simples, onde a execução do método *op* é equivalente à chamada direta da função. Ou seja, o resultado da execução do método *createUserData* é igual ao da execução do método *(op createUserData ...)*. Portanto, o resultado é um vetor que contém, na primeira posição, a operação do Jepsen a ser enviada ao cliente, com os parâmetros já definidos, e, na segunda posição, o gerador sem alterações.

Já o método *update* dos geradores do Jepsen tem a responsabilidade de atualizar o estado dos mesmos. Este método é utilizado logo após receber um evento, que tem o mesmo formato que uma operação do Jepsen (i.e., :type ..., :f ..., :value ...). Este método é invocado logo após a execução do método *op*, assim como no momento em que um cliente recebe uma resposta da aplicação em teste. No primeiro caso, o evento representa a operação do Jepsen gerada pelo método *op*. No segundo caso, o evento reflete essa mesma operação com a adição da resposta da aplicação.

No *RestOpGenerator*, o estado é o vetor *gens* e o método *update* define, em cada momento, as operações que devem ser associadas aos clientes. Quando a invocação do *update* é realizada a seguir à execução do método *op*, o método apenas retorna o gerador sem alterações (linhas 12 e 13). No cenário em que o *update* é invocado quando um cliente recebe uma resposta, o gerador pede ao grafo de transições a próxima operação que o cliente deve executar. Para isto, o gerador recolhe duas informações relevantes da última operação executada: o identificador da operação da API (linha 15) e o código de estado HTTP presente na resposta (linha 16). Estes dois dados são enviados ao grafo, que devolve o identificador da operação da API que o cliente deve executar a seguir. A esse identificador é adicionada a *String* "*Data*" (dentro da função *next-op-gen*, na linha 15), para representar o método do código gerado que cria a operação do Jepsen relativa à operação indicada da API (e.g., o método *createUserData* cria a operação do Jepsen relativa à operação *createUser* da API). A seguir, o vetor *gens* é atualizado na posição do cliente que recebeu a resposta, associando o respetivo método do código gerado (linha 17). Por fim, o método retorna o gerador que reflete essa modificação.

No Jepsen, as operações dos geradores retornam outros geradores para permitir que o processo de geração de operações seja dinâmico e adaptável ao longo do tempo. Assim, é permitido o uso de diferentes tipos de geradores durante a execução dos testes. No nosso caso, é sempre utilizado o mesmo gerador, pois o que interessa é a alteração dos métodos do vetor *gens*.

Listagem 5.12: Código do gerador de operações.

```
(defrecord RestOpGenerator [gens]
   gen/Generator
   (op [this test ctx]
      (let [free-threads (gen/free-threads ctx)]
         (if (empty? free-threads)
            [:pending this]
            (let [thread (first free-threads)]
               (when-let [[op _] (gen/op (get gens thread) test ctx)]
                  [op this])))))

   (update [this test ctx event]
      (if (= (:type event) :invoke)
         this
         (let [thread (:process event)
               gen' (next-op-gen (:typeOp (:input (:value event)))
                                 (:status (:output (:value event))))
               gens' (assoc gens thread gen')]
            (RestOpGenerator. gens')))))
```

Durante a execução do gerador de operações, para além de serem geradas as operações que os clientes devem executar, também é construída a história que será usada pelo *checker* para analisar a correção da aplicação. Esta é atualizada em dois momentos distintos: quando o gerador associa uma nova operação a um cliente; e quando um cliente avisa o gerador de que recebeu uma resposta.

5.1.3 Avaliação dos Resultados

Uma vez terminada a fase de teste, onde são realizados os pedidos à aplicação, o Jepsen armazena a história gerada num ficheiro chamado *history*.

O JepREST usa o *checker* Knossos para analisar se a história gerada é linearizável. Para isto, definimos um novo modelo que indica como as operações alteram o estado das aplicações REST, denominado JepRestModel.

O JepRestModel, cuja estrutura está apresentada na Listagem 5.13, tem um parâmetro que vai representar o estado da aplicação durante a fase de avaliação dos resultados (*allJsons*). A estrutura de dados escolhida para representar o estado da aplicação localmente é um mapa de mapas. As chaves do primeiro mapa representam os tipos de recursos, e os seus valores são mapas que contêm os recursos criados pela aplicação. As chaves desses segundos mapas são os identificadores dos recursos armazenados, enquanto os valores correspondem aos próprios recursos, representados como objetos JSON. A escolha desta estrutura de dados foi feita com a ideia de minimizar a complexidade das operações de leitura no estado local.

Listagem 5.13: Código do modelo do JepREST.

```
1 (defrecord JepRestModel [allJsons]
2    Model
3    (step [this op]
4       (if (some? (:custom-op (:value op)))
5          // Custom Semantics
6          // REST Semantics
7    )))
```

Este novo modelo, para poder ser usado pelo Knossos, implementa a interface *Model*. Esta interface apenas contém o método "*(step [this op])*", cujo resultado é o próprio modelo com o estado atualizado aplicando as alterações determinadas pela operação *op*. Como o JepREST permite testar aplicações que contêm operações com semântica personalizada, o novo modelo define duas formas de verificar se o resultado de uma operação está correto. A condição presente na quarta linha determina se a operação que o Knossos vai verificar tem um comportamento personalizado.

No caso em que o Knossos identifica uma incoerência entre o resultado de uma operação e o estado gerado pelas operações anteriores, o resultado da operação *step* é a invocação da operação "*(model/inconsistent msg)*". Esta operação indica que foi identificado um problema e, ao ser invocada, o Knossos termina a fase de avaliação dos resultados, apresentando a mensagem *msg*, que descreve a causa do problema de coerência.

A seguir, é apresentada com detalhe a implementação do modelo para ambos tipos de operações REST.

5.1.3.1 Operações com Semântica Personalizada

As operações com semântica personalizada têm o seu comportamento interno definido dentro do campo *:value* de cada operação presente na história. Este comportamento envolve uma sequência de uma ou mais suboperações que são executadas concorrentemente, logo, o Knosso analisa uma a uma pela ordem apresentada. No JepREST, chamamos "passo" a cada suboperação da sequência, e cada um tem um identificador associado que representa a sua posição na sequência.

O Knossos começa por recolher o código de estado HTTP presente na operação registada na história, juntamente com as sequências de códigos de estado dos passos que conduzem a cada código que a operação pode devolver, apresentadas no campo *:op-status-code-seq*. A seguir, são selecionadas as sequências de códigos dos passos que levam ao código retornado pela operação. Se o código da operação for de sucesso e não tiver sido especificada nenhuma sequência de códigos dos passos para esse código, o Knossos assume que todos os passos devem ser bem-sucedidos. Na Listagem 5.14, estão apresentadas duas sequências que determinam o resultado interno dos passos quando a operação devolve 404. Caso o resultado da operação presente na história contenha este código, o Knossos vai selecionar ambas sequências e verificar se o resultado dos passos satisfaz alguma delas. No caso em que o código da operação é de sucesso, como 200 ou

201, por defeito, o Knossos cria uma sequência onde os códigos dos passos também são de sucesso ([200 200]).

Listagem 5.14: Exemplo do campo *:op-status-code-seq* de uma operação com semântica personalizada.

```
1 {:type :ok, :f :get, :value {:custom-op { :op-steps ..., :op-status-code-seq
2           [{:op-status 404, :steps-status-seq [404 -1]}
3            {:op-status 404, :steps-status-seq [200 404]}}}}}
```

O resultado da análise de cada passo é uma instância das classes ConsistentModel ou InconsistentModel, apresentadas na Listagem 5.15. A classe ConsistentModel representa uma análise correta de um passo e tem o propósito de acumular as alterações no estado realizadas pelos passos. Ou seja, esta classe guarda, no atributo *state*, o estado da aplicação no momento em que o Knossos executa a operação e vai aplicando as alterações devidas após cada passo analisado. Para além disto, também guarda, no atributo *status*, o código do resultado do último passo analisado, de modo a que o Knossos saiba quais podem ser os resultados coerentes do passo seguinte. O atributo *step-result* armazena os recursos afetados pelo passo (e.g., se um passo é um POST, o recurso criado é guardado no *step-result*). Isto é feito a pensar nos casos em que um passo depende do resultado de um anterior, onde é usado o campo *:store-in-var* para determinar o nome da variável que vai conter esse resultado.

Quando o resultado da analise de um passo é uma instância da classe InconsistentModel, o Knossos percebe que existe um problema de incoerência provocado por esse passo. A mensagem de erro que indica qual foi o passo que falhou e o motivo do erro é definida no único atributo da classe, *error-msg*.

Listagem 5.15: Classes que representam o resultado dos passos.

```
1 (defrecord ConsistentModel [status state step-result])
2 (defrecord InconsistentModel [error-msg])
```

Primeira Fase da Análise

O Knossos cria uma variável chamada *previous-step-info* que guarda as informações referentes ao último passo analisado. Esta variável é do tipo ConsistentModel e começa com o *status* e *step-result* a *null*, e com o *state* igual ao estado armazenado no *allJsons*.

A seguir, o Knossos itera sobre os passos e verifica se respeitam algum código definido para eles nas sequências. Como explicado na secção 4.2.2.7, existem três códigos especiais para os passos, que têm prioridade na análise: -1, 0 e 1.

Primeiro, o *checker* comprova se alguma das sequências selecionadas contém o código -1 na posição referente ao passo da iteração. Caso uma ou mais sequências satisfaçam a condição, o *checker* atualiza a variável *previous-step-info* com um ConsistentModel, onde o *status* é igual a -1 e o *state* não sofreu nenhuma alteração, e avança para o passo seguinte,

sinalizando que o passo anterior foi ignorado. Por exemplo, se, numa operação, para o código 403 são definidas as sequências de códigos dos passos: [200 409 -1] e [200 -1 404], o Knossos, ao analisar o segundo passo, descarta a primeira sequência, o que determina que esta operação é coerente apenas se resultado do terceiro passo demonstrar que um recurso não existe. Caso contrário, a análise do passo continua.

Em seguida, o Knossos avalia se o passo em causa possui alguma condição definida. Se não tiver, a análise prossegue normalmente. Se existir alguma condição que determina quando o passo deve ser ignorado, o *checker* comprova essa mesma condição. No cenário em que a condição é respeitada, a análise do passo não sofre qualquer alteração. Pelo contrário, quando a condição não é cumprida, são examinados os códigos possíveis que o passo pode gerar, à procura do código 0. O código 0 determina que a condição do passo falhou. Logo, se nenhuma sequência de resultados dos passos contém este código, na posição referente ao passo em questão, a analise do passo retorna um InconsistentModel e o Knossos sinaliza que encontrou um problema, pois a condição do passo não era suposto falhar. Caso exista alguma sequência que contenha o código, ou seja, que determine que a condição definida para o passo não deve ser satisfeita, o *checker* ignora o passo e atualiza o código da variável *previous-step-info* para 0.

Por fim, quando alguma sequência contém o código 1 na posição do passo em análise, é indicado que o passo deve ser verificado normalmente, mas que o código resultante da análise é ignorado. Isto é utilizado nos casos onde a execução de um passo não afeta o código retornado pela operação da API. Por exemplo, numa rede social simples, quando um utilizador é removido, os seus *posts* também o são. No entanto, o código da resposta é o mesmo, independentemente dos recursos serem removidos com sucesso (200) ou do utilizador não ter recursos para remover (404).

Segunda Fase da Análise

Após estas três verificações iniciais, é realizada a análise normal do passo.

O *checker* começa por processar as informações relativas ao comportamento do passo em análise. Estas estão descritas nos campos *:filter* e *:write* da operação, e definem quais são os recursos afetados pelo passo e as alterações que vão sofrer, respetivamente. O campo *:write* nunca está presente quando o passo é definido como um GET, pois apenas representa uma operação de leitura, ou como um DELETE, visto que os recursos removidos são estabelecidos pelo campo *:filter*. Para determinar o valor dos parâmetros usados em ambos campos, o JepRestModel suporta o uso da linguagem regular definida na secção 4.2.2.6, que permite indicar onde estão localizados os valores dentro de um pedido ou de uma resposta. Esta linguagem habilita que o *checker* possa executar o passo com os mesmos valores que o cliente. Por exemplo, o passo relativo à operação GET "*users/john-1*" está definido na Listagem 5.16. O valor do campo *:value* do *:simple-filter* determina que o recurso filtrado deve ter o mesmo identificador que o enviado pelo cliente no parâmetro *userId* do caminho do pedido, neste caso "*john-1*". O campo *:attribute* do *:simple-filter* é definido

como *nil* devido à presença do *:filter-by-id? true*, que indica que o primeiro elemento da lista de filtros representa o identificador do recurso, logo, não é necessário saber o nome do atributo. Isto é possível pois o estado mantido pelo modelo, no *allJsons*, guarda os recursos dentro de um mapa cujas chaves são os identificadores.

Listagem 5.16: Definição do passo que representa a operação GET "*users/{userId}*"

```
1 {:method :get, :resource "users", :filter-by-id? true,
2  :filter {:simple-filter [{:attribute nil :value "$request.path#/userId"}]}}
```

Os valores que o *checker* usa quando analisa um passo são determinados pela operação *get-value-of-attribute*, que recebe como parâmetros o corpo e os cabeçalhos do pedido realizado e da resposta recebida, assim como os parâmetros de caminho e pesquisa enviados pelo cliente. Para além disto, também recebe um mapa que contém as variáveis criadas por passos anteriores, que guardam os recursos afetados por eles. Esta operação do modelo tem como resultado o valor presente na localização indicada na especificação, e pode ser, por exemplo, uma *String*, um inteiro, um *array* de *bytes* ou, até, um objeto JSON que representa um recurso, no caso da localização ser uma variável criada por algum passo.

Existe um caso especial para quando o valor requerido está presente num atributo do recurso afetado pelo passo. Este caso é usado quando um passo tem a necessidade de aceder a um atributo do próprio recurso afetado por ele. Um bom exemplo é a operação que aumenta o número de visualizações de um artigo, onde o passo simplesmente usa o valor que já está presente no recurso e soma-lhe 1, como está apresentado na Listagem 4.21. Esta abordagem foi criada para diminuir a complexidade do comportamento deste tipo de passos, pois, sem ela, o processo teria de ser dividido em dois: um GET, para obter o recurso, e um outro passo, para realizar a operação desejada com o uso de um atributo desse recurso. Neste caso, a operação *get-value-of-attribute* do modelo devolve a *keyword* relativa ao atributo especificado na linguagem regular. Logo, para o caso exemplificado, onde a localização do valor é definida como "*$resource#/views*", a operação devolve *:views*.

O JepRestModel tem duas operações que utilizam a função *get-value-of-attribute* de modo a obter os valores necessários para executar operações de pesquisa e de escrita no estado. A operação *process-filter* cria uma instância da classe Filter, onde ficam contidos todos os valores relevantes para identificar os recursos afetados pelo passo. Quando o campo *:simple-filter* está especificado na descrição do passo, os atributos definidos recebem os valores exatamente iguais aos contidos nas localizações especificadas. Caso seja definida uma *String* que representa uma função de pesquisa mais complexa no campo *:filter-function*, a mesma é convertida numa função em Clojure, cujos parâmetros conterão os valores especificados na lista *:filter-function-params*.

De forma semelhante, a operação *process-write* segue um princípio similar, mas ao invés de criar uma instância da classe Filter, cria uma instância da classe Write. Esta classe contém informação sobre as alterações que um passo efetua nos recursos filtrados. O campo *:simple-write*, representa operações de escrita que simplesmente substituem o valor

antigo pelo novo. Por outro lado, o campo *:complex-write*, como o nome sugere, representa operações de escrita mais elaboradas, onde funções específicas determinam como os novos valores dos atributos são gerados. Neste campo, cada atributo atualizado tem a sua própria função, ao contrário do Filter, onde apenas é necessária uma função, uma vez que o resultado deve ser um booleano. Cada atributo a ser modificado, definido no campo *:attribute*, possui uma função em Clojure e parâmetros próprios, definidos nos campos *:write-function* e *:write-function-params*, que estabelecem qual será o valor final do atributo após a execução do passo. Os valores dos elementos presentes na lista de parâmetros são obtidos através da operação *get-value-of-attribute*.

Operações de Leitura

Uma vez determinados os valores de pesquisa e de escrita, o *checker* começa por filtrar os recursos utilizando a operação *filter-resources*, que recebe como parâmetros a instância da classe Filter, criada anteriormente, e o mapa que representa os recursos do tipo determinado no campo *:resource* do passo. O comportamento desta operação está dividido em duas partes.

Primeiro, a operação filtra os recursos usando os dados do *simple-filter*, ou seja, percorre todos os recursos e devolve apenas aqueles que contêm os valores dos atributos iguais aos especificados no filtro. Por exemplo, se o *simple-filter* determina que o atributo "*age*" dos recursos do tipo "*users*" deve ter o valor 18, a operação devolve apenas os "*users*" com idade igual a 18. O *simple-filter* também pode especificar unicamente o identificador do recurso que o passo pretende aceder, sendo que, neste caso, a operação devolve apenas esse recurso.

Em seguida, a operação aplica um filtro adicional sobre os recursos já filtrados na primeira parte, utilizando a função definida no *complex-filter*. Nesta fase, a operação percorre os recursos filtrados e aplica a função com os parâmetros indicados. Ampliando o exemplo anterior, se o *complex-filter* definir a seguinte função "*(fn [str] (string/starts-with? str "A"))*", em que o valor do parâmetro *str* está definido como "*$resource#/name*", os recursos resultantes serão aqueles cujo nome começa com a letra "A". Assim, o resultado da filtragem seriam todos os "*users*" com 18 anos cujos nomes começam por "A". O conjunto de recursos para os quais a função retorna *true* constitui o resultado deste filtro e, por conseguinte, da operação *filter-resources*.

A filtragem é a primeira operação a ser realizada devido a que o seu resultado é importante na análise de qualquer tipo de passo.

Operações de Escrita

Quando a análise dos passos definidos como POST, PUT, PATCH ou DELETE é considerada coerente, as alterações que o estado sofre são determinadas pelas operações

de escrita. Ao contrário das operações de leitura, as operações de escrita têm três propósitos diferentes: atualizar, criar ou remover recursos.

A operação responsável por atualizar um ou mais recursos é chamada *update-resources*. Esta operação utiliza os dados armazenados na instância da classe Write, passada como parâmetro, para atualizar os mapas do estado que representam os recursos filtrados. Para isso, a operação também recebe como parâmetros o tipo do recurso e o estado completo no momento da execução do passo.

O comportamento desta operação é aplicado a cada recurso filtrado e divide-se pela forma em como podem ser modificados. Na primeira fase, o *checker* itera sobre o *simple-write* e, para cada elemento, adiciona uma entrada no mapa com a chave igual ao dado definido no campo *:attribute* e o valor igual ao definido no campo *:value*, sobrepondo os valores antigos. Na segunda fase, o mesmo processo é realizado para o *complex-write*, que aplica a função indicada no campo *:write-function*, utilizando os parâmetros fornecidos em *:write-function-params*, para gerar o valor da entrada no mapa com a chave correspondente ao dado no campo *:attribute*. O resultado desta operação é o estado após realizadas as alterações atualizações.

A operação *update-resources* é utilizada dentro da operação *create-resource*, cujo papel é criar um recurso. Ao contrário da operação anterior, esta não recebe uma lista de recursos filtrados, pois quando um recurso é criado não existe nenhum outro com o mesmo identificador. Porém, a operação *update-resources* recebe uma lista de parâmetros filtrados como um dos argumentos, o que não existe quando o passo se refere a um POST. Portanto, primeiro o *checker* cria uma nova entrada no mapa dos recursos com o tipo indicado, onde a chave é o identificador fornecido nos parâmetros e o valor, um mapa vazio. A seguir, o mapa que representa o recurso criado é adicionado a uma lista vazia, e esta mesma é enviada à operação que atualiza recursos, juntamente com a instância da classe Write, do nome do tipo do recurso e do estado que agora contém uma entrada para o recurso prestes a ser criado. Por fim, a operação retorna um mapa que contém o estado atualizado e o objeto JSON que representa o recurso criado, caso seja necessário armazená-lo numa variável associada ao passo.

No caso dos passos definidos como DELETE, o *checker* utiliza a operação *detele-resources*, que tem uma implementação mais simples. Esta operação recebe como argumentos o nome do tipo dos recursos, uma lista que contém os recursos filtrados e o estado. A operação simplesmente elimina as entradas do mapa que representa o tipo de recursos determinado que contêm como chave algum dos identificadores dos recursos filtrados. Após a remoção de todos os recursos indicados, a operação retorna o estado modificado.

Verificação de Incoerências

Após a execução da operação que filtra os recursos afetados pelo passo em análise, o *checker* verifica a correção do mesmo. Esta verificação consiste em analisar se os recursos filtrados são coerentes com os recursos criados e atualizados pelas operações da API

executadas anteriormente ou pelos passos da própria operação. Como cada passo pode ter mais do que um código referente a uma mesma resposta da aplicação, o *checker* verifica a correção do passo para cada um desses códigos. Por exemplo, o fragmento da operação Jepsen presente na Listagem 5.14 indica que quando o resultado da operação contém o código 404, o primeiro passo pode resultar em dois códigos possíveis: 404 ou 200. Logo, o *checker* tem de verificar qual dos dois casos é coerente.

Este comportamento é implementado pela operação *verify-step*, apresentada na Listagem 5.17. No início, a operação decide quais são os códigos possíveis para o passo identificado pelo parâmetro *num-step*. De seguida, para cada código possível, é invocada a operação *verify-method*, cujo resultado é uma instância da classe ConsistentModel, quando o passo com o respetivo código é coerente, ou InconsistentModel, quando não o é. Esta operação decide qual é o tipo de verificação que deve ser realizada, dependendo do método listado no passo. Caso exista algum código que torne o passo coerente com o estado da aplicação, a operação retorna a instância correspondente do ConsistentModel. Se, nesse caso, o resultado do passo não é relevante, o código armazenado na classe é trocado para 1. Caso contrário, a operação retorna a respetiva instância da classe InconsistentModel. O modelo coerente é sempre o primeiro da lista de modelos coerentes (linha 9), devido a que um passo nunca pode ter mais do que um resultado coerente. Por sua vez, nas linhas 14 e 15, a mensagem de erro que o *checker* envia ao utilizador é a união de todas as mensagens dos resultados incoerentes. Nesta versão do JepREST, nunca podem haver dois códigos incoerentes, pois apenas são definidas as verificações para os códigos 2XX, 404 e 409. Porém, em versões posteriores, onde novos códigos sejam definidos, um passo poderá ter vários códigos que denotem incoerências.

Listagem 5.17: Código de verificação de um passo.

```
1  (defn verify-step [step-method op-status step-possible-status-list resource-name
2                     post-id resources write allJsons num-step ignore-step-status?]
3     (let [status-list (if (= ignore-step-status? true) ...)
4           v-list (map #(verify-method step-method op-status % ...) status-list)
5           consistent-model-info (set (filter #(instance? ConsistentModel %) v-list))
6           inconsistent-model-info (set (filter #(instance? InconsistentModel %) v-list))
7           step-consistent? (not (empty? consistent-model-info))]
8        (if step-consistent?
9           (let [consistent-model (first consistent-model-info)]
10             (if (= ignore-step-status? true)
11                (->ConsistentModel 1 (:state consistent-model)
12                                     (:step-result consistent-model))
13                consistent-model))
14          (->InconsistentModel (string/join "\n"
15                                     (map #(:error-msg %) inconsistent-model-info))))))
```

A verificação de incoerências dos passo definidos como GET, PUT, PATCH ou DELETE é igual, uma vez que suportam os mesmos códigos HTTP (i.e., 2XX e 404). O código utilizado para determinar se um passo destes é coerente ou não, está apresentado na

Listagem 5.18. Neste, são definidas três condições que, ao serem cumpridas, demonstram que o código indicado não é suposto ocorrer, tendo em conta os recursos presentes no estado nesse momento.

Primeiro, quando a análise é referente ao código 404 e os recursos filtrados existem, a operação retorna uma instância da classe InconsistentModel com a mensagem de erro apropriada. O código 404 implica que um ou mais recursos não existem; portanto, se a lista de recursos filtrados contiver pelo menos um elemento, o código do passo não pode ser esse. Para a análise deste código, verifica-se apenas se existem recursos que contêm os dados enviados no pedido do cliente, sem avaliar se os recursos filtrados são os corretos, pois é impossível ter acesso aos recursos acedidos internamente por uma operação da API. Caso os recursos especificados existam, a operação retorna um ConsistentModel sem qualquer alteração no estado.

Em segundo lugar, quando o código é de sucesso, mas não existem recursos que satisfaçam os filtros, a operação retorna novamente um InconsistentModel. Neste caso, o problema está em que o passo deveria ter sido bem-sucedido; no entanto, não existe nenhum recurso que possa ser afetado pelo mesmo. Isto acontece porque este tipo de métodos HTTP só pode ter sucesso se o recurso existir.

Por último, quando o código não é 2XX nem 404 (e.g., 409), a especificação está mal definida, e um InconsistentModel é retornado, descrevendo este mesmo cenário na sua mensagem. Neste caso, o problema encontrado demonstra um erro na especificação da API e não uma incoerência com o estado.

Listagem 5.18: Código base das operações de verificação de incoerências dos passos :get :put :patch e :delete.

```
(defn verify-get/put/del [op-status step-status resource-name resources allJsons num-step]
   (if (= step-status 404)
      (if (not (empty? resources))
         (->InconsistentModel <error message>)
         (->ConsistentModel step-status allJsons nil))
      (if (and (>= step-status 200) (< step-status 300))
         (if (empty? resources)
            (->InconsistentModel <error message>)
            <Code that determines how the state is changed>)
         (->InconsistentModel <error message>))))
```

No caso dos passos identificados com o verbo HTTP POST, a análise difere da anterior, uma vez que este passo não aceita o código de erro 404, mas sim o 409. Este código indica que o recurso que se pretende criar já existe. A operação que trata deste caso está definida na Listagem 5.19. Como é possível observar, a estrutura é idêntica à da verificação dos passos definidos com os outros verbos HTTP, mas as condições são diferentes.

Quando o código em análise é 409, é verificado se a lista de recursos filtrados está vazia. Se não estiver, significa que o código é coerente com os recursos presentes no estado, e a operação retorna um ConsistentModel, sem que o estado sofra qualquer alteração. Se, por outro lado, não existir nenhum recurso no estado com os atributos especificados no filtro

(i.e., a lista está vazia), a operação devolve um InconsistentModel, uma vez que a ação de criar um recurso deveria ter sido bem-sucedida.

Quando o código é de sucesso, ou seja, quando o recurso enviado no pedido é suposto ser criado, verifica-se se a lista de recursos filtrados contém algum elemento. No caso da lista não estar vazia, isto é, já existir um recurso com o identificador do recurso que se pretende criar, o resultado da operação é um InconsistentModel, onde a mensagem descreve que o passo falhou, uma vez que um POST só pode criar um recurso se não existir nenhum outro com o mesmo identificador.

Por fim, quando o código não é esperado para operações definidas como POST (e.g., 404), a operação retorna um InconsistentModel, com uma mensagem a indicar que existe um problema na especificação.

Listagem 5.19: Código base das operações de verificação de incoerências dos passos :post.

```
(defn verify-post [op-status step-status resource-name post-id resources write allJsons
                   num-step]
   (if (= step-status 409)
      (if (empty? resources)
         (->InconsistentModel <error message>)
         (->ConsistentModel step-status allJsons nil))
      (if (and (>= step-status 200) (< step-status 300))
         (if (not (empty? resources))
            (->InconsistentModel <error message>)
            <Code that determines how the state is changed>)
         (->InconsistentModel <error message>))))
```

Atualização do Estado

As alterações que os passos efetuam no estado apenas são efetuadas quando não é detetada nenhuma incoerência entre o estado que o *checker* mantém e o código definido na análise do passo. A seguir, são explicadas as atualizações realizadas no estado pelos diferentes tipos de passos:

- **:get** - Uma operação GET nunca altera o estado da aplicação, pois o seu propósito é apenas devolver ao cliente os recursos solicitados. Portanto, no JepREST, quando um passo definido como GET é coerente, o estado não sofre qualquer alteração, e o resultado contém os recursos armazenados com os atributos definidos no pedido. Em termos de código, a operação retorna o ConsistentModel definido na Listagem 5.20, onde o resultado do passo é a lista de recursos filtrados no inicio da análise. Por isso, é aplicada a função *vals* do Clojure no mapa de recursos filtrados, *resources*, visto que os recursos estão nos valores do mapa.

Listagem 5.20: Código que determina as alterações no estado quando um passo :get é coerente.

```
1 (->ConsistentModel step-status allJsons (vals resources))
```

- **:post** - O objetivo do método POST é criar um recurso. Por isso, quando um passo definido como POST é coerente, o recurso é adicionado ao estado. No JepREST, este comportamento é efetuado pelo método *create-resource*, descrito na secção 5.1.3.1. Como está apresentado na Listagem 5.21, após a execução do método, é criado um ConsistentModel que contém o estado atualizado e o recurso criado, para o caso de ser necessário guardá-lo numa variável.

Listagem 5.21: Código que determina as alterações no estado quando um passo :post é coerente.

```
1 (let [create-resource-res (create-resource write resource-name post-id allJsons)
2       created-resource (:resource-created create-resource-res)
3       new-state (:state create-resource-res)]
4    (->ConsistentModel step-status new-state created-resource))
```

- **:put/:patch** - O método PUT é utilizado para substituir completamente um recurso. O PATCH, por outro lado, é usado para modificar parcialmente um recurso. Como a finalidade de ambos é atualizar recursos, o JepREST não faz distinção entre os passos definidos como PUT ou PATCH. A Listagem 5.22 mostra como é criado o ConsitentModel quando este tipo de passos é coerente. Primeiro, realiza-se a atualização do estado utilizando o método *update-resources*, descrito na secção 5.1.3.1. A seguir, ao utilizar o método *get-resources-with-keys*, obtém-se uma lista com todos os recursos modificados. Esta lista representa o resultado do passo e poderá ser utilizada em passos subsequentes. Por fim, é criado o ConsistentModel com todos estes dados.

Listagem 5.22: Código que determina as alterações no estado quando um passo :put ou :patch é coerente.

```
1 (let [updated-state (update-resources resource-name resources write allJsons)
2       updated-resources (get-resources-with-keys update-resources
3                                                  (keys resources) resource-name)]
4    (->ConsistentModel step-status updated-state updated-resources))
```

- **:delete** - O método DELETE representa a remoção de um recurso. Neste tipo de passos, utiliza-se o método *delete-resources*, definido na secção 5.1.3.1, para remover do estado os recursos identificados no mapa que contém os recursos filtrados, *resources*. O resultado do passo, que é incluído no ConsistentModel, é a lista que inclui os recursos que foram removidos, correspondendo exatamente aos que foram filtrados. O código que cria este ConsistentModel está definido na Listagem 5.23.

Listagem 5.23: Código que determina as alterações no estado quando um passo :delete é coerente.

```
1 (->ConsistentModel step-status (delete-resources resource-name resources allJsons)
2                    (vals resources))
```

Uma vez realizadas as verificações de incoerências dos passos e as devidas atualizações no estado, o *checker* analisa a condição definida para a resposta da operação, caso exista. Este comportamento ocorre quando os passos são coerentes. A função definida no campo *:response-condition-function* é executada utilizando os parâmetros indicados no campo *:response-condition-params*. Se a condição for satisfeita, os dados presentes na resposta estão corretos. Caso contrário, o Knossos indica que os valores devolvidos pela aplicação são incoerentes com o estado local e retorna um *InconsistentModel*, que explica essa incoerência.

5.1.3.2 Operações que Usam a Semântica Padrão do REST

A verificação de incoerências em operações que seguem a semântica padrão do REST, definida na primeira versão da ferramenta, é semelhante a que é realizada singularmente nos passos correspondentes aos diferentes métodos HTTP. A única diferença está nas operações GET, onde, para além de efetuar as mesmas verificações que no passo definido como GET, verifica-se se os dados presentes na resposta são coerentes com o estado mantido pelo Knossos.

Isto é possível porque a arquitetura REST determina que a resposta a qualquer operação GET deve ser uma lista com os recursos solicitados. Deste modo, o JepREST pode verificar se os recursos devolvidos estão corretos, tendo em conta o estado. Se algum recurso da resposta não estiver no estado ou tiver valores diferentes, o Knossos indica um problema de linearização, uma vez que o GET deve devolver exatamente os recursos presentes no estado gerado pelas operações executadas anteriormente.

5.2 Geração de Workloads

Nesta secção é detalhada a implementação da ferramenta que gera o grafo de transições das operações da API em teste.

5.2.1 Implementação do Grafo de Transições das Operações da API

Para a implementação do grafo de operações da API, foram definidas três classes distintas: *Edge*, *Weight* e *Graph*.

A classe *Edge* representa as arestas do grafo. Esta classe guarda o identificador da operação do vértice inicial, o identificador da operação do vértice destino, e o peso associado à aresta. Este peso é uma instância da classe *Weight*, e representa o peso de uma aresta, que corresponde à probabilidade de essa aresta ser escolhida como o próximo caminho a ser percorrido no grafo. Esta classe contem um valor do tipo *byte*, que indica essa

probabilidade. Este valor é um byte por ser o tipo de dado primitivo, capaz de representar probabilidades, que menos espaço ocupa em memória. Isto é relevante, pois uma aplicação complexa pode ter um milhares de arestas. O peso poderia ser representado apenas por um *byte*, mas foi criada a classe com a finalidade de que no futuro possam ser adicionadas mais condições, para além da probabilidade simples.

A classe *Graph* é a que representa o grafo de transições das operações da API. Este grafo está implementado como um mapa de mapas, onde a chave é o identificador das operações, e o valor, um mapa. Este segundo mapa tem como chave um inteiro, que representa um código de estado HTTP, e como valor, um *TreeSet* de objetos do tipo *Edge*. Usando, como exemplo, o grafo apresentado na Figura 5.1, o mapa que o representa tem uma entrada onde a chave é "*getUser*", e o valor é um mapa, com, pelo menos, duas entradas. Uma destas entradas, tem como chave o código HTTP 404 e apenas uma aresta. Logo, quando a operação *getUser* retorna uma resposta com código 404, o vértice sucessor será o vértice destino presente nessa aresta, que, no caso descrito, é o vértice identificado como "*createUser*". A outra entrada tem como chave o código 200, o que significa que, quando a operação *getUser* é bem-sucedida, existem dois possíveis vértices sucessores com diferentes probabilidades de serem escolhidos. O vértice identificado como "*updateUser*" é o mais provável de ser escolhido como sucessor, pois tem um peso de 90%, enquanto que o vértice identificado como "*getUser*" tem uma pequena probabilidade de 10%.

A escolha do uso da classe *TreeSet* está relacionada com a necessidade de manter a ordem do conjunto de arestas, que devem estar organizadas de menor a maior peso, devido à implementação da função que devolve o vértice sucessor. Esta função está presente na Listagem 5.24. Na primeira linha, observa-se que a função recebe como parâmetros o identificador de uma operação e o código de estado da respetiva resposta. Na segunda linha, é gerado um número aleatório, entre 0 e 99, que será usado mais à frente. Na terceira linha, é obtido o mapa que detém a informação sobre os possíveis vértices sucessores, dependendo do código da resposta da operação. Se não existir nenhuma aresta com início no vértice em questão, significa que a operação não era suposto ter esse resultado, e a função retorna o valor *null* (linha 4). Caso exista pelo menos uma, na quinta linha, é obtido o conjunto de arestas respetivas ao código dado nos parâmetros da função. Para escolher o vértice sucessor, o *TreeSet* é percorrido até encontrar uma aresta que tenha um peso maior que o número gerado aleatoriamente no início da função (linhas 7-12). Quando esta aresta é encontrada, o valor que a função retorna é o identificador do vértice de destino da aresta (linha 9). Se as arestas não estivessem ordenadas por peso, poderia acontecer que uma aresta com um peso elevado estivesse no início do conjunto, o que faria com que as arestas com peso menor nunca fossem escolhidas. Esta é a razão principal da escolha do *TreeSet*.

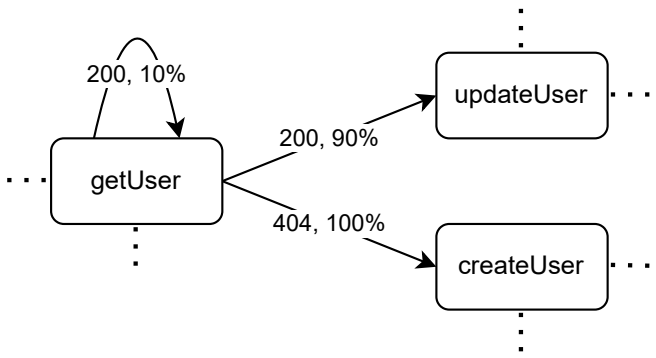


Figura 5.1: Exemplo gráfico de uma parte de um grafo de transições de operações.

Listagem 5.24: Código da operação que decide que caminho tomar no grafo.

```
1   public String getNextVertex(String v, int code) {
2       int r = random.nextInt(100);
3       Map<Integer, TreeSet<Edge>> vertexInfo = graph.get(v);
4       if (vertexInfo == null) return null;
5       TreeSet<Edge> edges = vertexInfo.get(code);
6       if (edges != null)
7           for (Edge edge : edges) {
8               if (r <= edge.getWeight().getWeight()) {
9                   return edge.getDestination();
10              }
11              r -= edge.getWeight().getWeight();
12          }
13      return null;
14  }
```

5.2.2 Geração dos Vértices do Grafo

Inicialmente, esta ferramenta copia todos os documentos interface e objeto da aplicação em teste para ter acesso às anotações, pois contêm toda a informação necessária para a geração dos testes. As anotações relevantes para esta ferramenta são as contidas nos documentos interface, pois determinam quais são as operações disponibilizadas pela API da aplicação em teste, e descrevem como são executadas as operações e o significado dos seus resultados. Os documentos objeto não são relevantes, mas são necessários devido à dependência que os documentos interface têm em relação a eles.

Uma vez tendo acesso a estes documentos, os documentos interface são percorridos, e, por cada operação, é adicionado um vértice novo ao grafo, que terá associado a ele o identificador da operação. No fim desta fase, a ferramenta tem à sua disposição um grafo nulo, onde cada vértice representa uma operação da API.

5.2.3 Geração das Arestas do Grafo

Nesta secção, explica-se como são criadas as arestas do grafo e como são gerados os pesos a elas associados. Primeiro, é descrita a estrutura que o ficheiro de *logs* deve apresentar para poder ser interpretado pela ferramenta. A seguir, são apresentadas as três fases que levam à geração das arestas. Na primeira fase, a ferramenta lê o ficheiro de *logs* da aplicação, e armazena, para cada cliente, a sequência de pedidos que efetuou e as respostas que recebeu. Na segunda fase, a ferramenta calcula o número de vezes que cada sequência de duas operações foi realizada pelos clientes. Na terceira e última fase, usando os dados calculados na fase anterior, a ferramenta define os pesos e adiciona as devidas arestas ao grafo.

5.2.3.1 Estrutura do ficheiro de *logs*

Para definir as ligações entre os vértices do grafo, é preciso saber como é utilizada a aplicação em teste, para, desta maneira, acabar com um grafo que possa gerar sequências de operações similares às dos utilizadores da aplicação. Para isto, são utilizados os *logs* da aplicação, pois contêm todos os pedidos feitos pelos clientes e os respetivos resultados. Ao executar esta ferramenta, deve ser especificado o caminho para o ficheiro de *logs* da aplicação.

Como a estrutura dos ficheiros de *logs* de distintas aplicações pode ser diferente, foi definido um padrão (i.e., uma expressão regular) que deve ser utilizado quando se quiser usar esta ferramenta. Este padrão contém as seguintes informações:

- Identificador do pedido executado pelo cliente.
- IP do cliente que executou o pedido.
- Data em que o pedido/resposta foi realizado/a.
- Método HTTP utilizado no pedido.
- Caminho do pedido.
- Código de estado HTTP da resposta. Somente se o evento representa uma resposta.
- Lista com os parâmetros do caminho do pedido.
- Lista com os parâmetros de pesquisa do pedido.
- Cabeçalhos presentes no pedido/resposta.
- Corpo do pedido/resposta.

A Listagem 5.25 apresenta um exemplo de uma linha do ficheiro de *logs*. Esta linha representa uma resposta de sucesso a um pedido que o cliente, com o IP 192.168.0.2, realizou no dia 2 de setembro de 2024. O pedido foi um simples GET do utilizador com

identificador "*john*" e palavra-passe "*pwd*", e, como pode ser observado na listagem, no corpo da resposta está um objeto JSON que representa esse utilizador.

Listagem 5.25: Exemplo de uma resposta no ficheiro de *logs*.

```
1 [535032503:192.168.0.2] Mon Sep 02 11:03:08 UTC 2024 - GET users/john 200 params={{userId
      ↪ =[john]}} query={pwd=pswd} headers={Content-Type=[application/json]} body={"userId
      ↪ ":"john","pwd":"pwd","email":"johndoe@email.com","displayName":"John Doe"}
```

Como, na versão atual desta ferramenta, os cabeçalhos e o corpo do pedido/resposta não são utilizados para gerar as arestas, foi definido um padrão mais simples, onde estas informações não são apresentadas no ficheiro de *logs*. Desta forma, o ficheiro fica mais legível, como é demonstrado na diferença entre as Listagens 5.25 e 5.26, onde ambas representam a mesma linha do ficheiro.

Listagem 5.26: Exemplo de uma resposta no ficheiro de *logs* sem cabeçalhos nem corpo.

```
1 [535032503:192.168.0.2] Mon Sep 02 11:03:08 UTC 2024 - GET users/john 200 params={{userId
      ↪ =[john]}} query={pwd=pswd}
```

5.2.3.2 Interpretação do ficheiro de *logs*

Nesta fase, a ferramenta lê o ficheiro de *logs* da aplicação e interpreta linha a linha. Cada linha representa um pedido, uma resposta, ou outro tipo de informação não relevante para a ferramenta. Neste último caso, a ferramenta ignora a linha em questão e passa à seguinte. Caso a linha represente um pedido ou uma resposta, a ferramenta transforma a linha, que inicialmente é uma *String*, numa instância da classe *LogEntry* e adiciona-a à lista de *logs* do respetivo cliente. A classe *LogEntry* representa uma entrada no ficheiro de *logs* e guarda toda a informação relativa a essa entrada (e.g., IP do cliente, método HTTP, caminho do pedido). As entradas são armazenadas por cliente, para manter a causalidade das operações. Ou seja, a ferramenta guarda a sequência de pedidos e respostas dos clientes por separado. Isto tem a ver com a forma em como está implementada a fase seguinte.

5.2.3.3 Cálculo da Frequência de Sequências de Operações

Após percorrer todas as entradas do ficheiro de *logs* e obter as sequências de pedidos e respostas, respetivas a cada cliente, a ferramenta calcula a frequência das sequências de operações. Posto de forma mais simples, a ferramenta calcula o número de vezes que duas operações são executadas consequentemente. Este comportamento está exposto na função apresentada na Listagem 5.27. Esta função é executada uma vez por cada cliente e recebe como parâmetro a lista de entradas do respetivo cliente. A função começa por remover a primeira entrada da lista e guardá-la na variável *previousEntry*, que será utilizada como a "entrada anterior" durante cada iteração da análise. A seguir, a função itera sobre o restante da lista de entradas, e, em cada iteração, a "próxima entrada" é processada. Se a

previousEntry for uma resposta, a função procede à atualização do mapa que armazena a contagem de transições entre vértices (*numberOfTimesEdgesOccurred*). Nas linhas 9 e 10, são criados dois mapas auxiliares, *map1* e *map2*. O *map1* tem, como chave, um inteiro que representa um código de estado HTTP, e, como valor, um mapa. Este segundo mapa, tem como chave, os identificadores dos vértices, e, como valor, a quantidade de vezes que a operação contida neles ocorreu depois da operação presente na *previousEntry*. O *map2* é o mapa contido no *map1* em que a chave é o código de estado HTTP presente na resposta *previousEntry*. Em continuação, na linha 11, é somado 1 ao valor do *map2* que tem a chave igual ao identificador do vértice do *succeedingEntry*. Este valor representa a quantidade de vezes que a operação do *succeedingEntry* foi executada depois da operação do *previousEntry*. No fim, nas linhas 12 e 13, o *map2* atualizado é inserido no *map1*, e o *map1* é inserido de volta no *numberOfTimesEdgesOccurred*. Caso a *previousEntry* seja um pedido, a função ignora essa entrada, sem atualizar o mapa de frequência das transições. Isto deve-se a que uma transição tem de ser de resposta para pedido, pois o caso inverso simplesmente representa a resposta ao pedido anterior. No final de cada iteração, a *succeedingEntry* passa a ser a *previousEntry*, preparando-se assim para a próxima iteração.

Listagem 5.27: Código da função que calcula a frequência das sequências de operações.

```
1  private void handleSequentialLogs(List<LogEntry> entries) {
2      ...
3      LogEntry previousEntry = entries.remove(0);
4      ...
5      for (LogEntry succeedingEntry : entries) {
6          String previousVertexId = previousEntry.getVertexId();
7          String succeedingVertexId = succeedingEntry.getVertexId();
8          if (previousEntry.isResponse()) {
9              Map<Integer, Map<String, Integer>> map1 = numberOfTimesEdgesOccurred.
                   ↪ computeIfAbsent(previousVertexId, k -> new HashMap<>());
10             Map<String, Integer> map2 = map1.computeIfAbsent(previousEntry.getStatus(), k ->
                   ↪  new HashMap<>());
11             map2.merge(succeedingVertexId, 1, Integer::sum);
12             map1.put(previousEntry.getStatus(), map2);
13             numberOfTimesEdgesOccurred.put(previousVertexId, map1);
14         }
15         previousEntry = succeedingEntry;
16     }
17 }
```

É importante realçar que os identificadores dos vértices não são os identificadores das operações a eles associadas. O identificador de um vértice é a união da *String* que representa o método HTTP com a *String* que determina o caminho do pedido. Por exemplo, se é realizado um GET com caminho "*users/john-1*", onde "*john-1*" é o valor do parâmetro *userId* do caminho, o identificador do vértice é "*GET /users/{userId}*", e diz respeito à operação com identificador "*getUser*". Portanto, existe uma relação direta entre o identificador de um vértice e o identificador da respetiva operação. Isto é usado para

identificar as operações nas entradas dos *logs*.

5.2.3.4 Definição dos Pesos e Construção das Arestas do Grafo

Depois de calculada a frequência com que cada sequência de operações ocorreu nos *logs* da aplicação, a ferramenta tem todos os dados necessários para gerar os pesos das arestas e ligar os vértices do grafo. Para isto, a ferramenta percorre os vértices que foram sinalizados como sendo vértices fonte de arestas. Para cada um desses vértices, são obtidos os códigos de estado HTTP que a operação, do vértice em questão, retornou nas diferentes respostas. Devido à fase anterior, a ferramenta sabe quantas vezes foi executada cada operação a seguir a uma resposta relativa à operação do vértice fonte. Então, o cálculo dos pesos fica bastante simples. O peso de uma aresta, que liga os vértices fonte e destino, e que tem como condição o código de estado HTTP, é simplesmente a divisão entre o número de vezes em que a operação do vértice destino foi executada após receber uma resposta da operação do vértice fonte com o código presente na aresta, e o número total de vezes em que qualquer operação foi executada após o resultado da operação do vértice fonte conter o código de estado presente na aresta. Após o cálculo do peso da aresta, a mesma é introduzida no grafo.

Este comportamento está implementado na função apresentada na Listagem 5.28.

Listagem 5.28: Código da função que gera o peso das arestas.

```
1  public void fillGraph(Graph graph) {
2      ...
3      for (String vertexId : numberOfTimesEdgesOccurred.keySet()) {
4          ...
5          Map<Integer, Map<String, Integer>> map1 = numberOfTimesEdgesOccurred.get(vertexId);
6          for (Integer status : map1.keySet()) {
7              Map<String, Integer> map2 = map1.get(status);
8              int total = map2.values().stream().mapToInt(Integer::intValue).sum();
9              for (String nextVertexId : map2.keySet()) {
10                 int weight = (int) Math.round((double) map2.get(nextVertexId) / total * 100)
11                 graph.addEdge(vertexId, nextVertexId, status, new Weight((byte) weight));
12             }
13         }
14     }
15 }
```

Nesta fase, é também calculada a probabilidade de cada vértice ser o primeiro da sequência executada por um cliente. Ao contrário da função que gera o vértice seguinte, para obter o primeiro vértice de uma sequência não é necessário enviar qualquer informação sobre operações passadas, até porque ainda não foi executada nenhuma. Assim, a probabilidade de um vértice ser o primeiro de uma sequência está diretamente relacionada com a frequência com que cada operação foi a primeira a ser executada pelos clientes. O cálculo é, portanto, simplesmente a divisão entre o número de vezes que uma operação foi a primeira a ser executada e o número de sessões existentes no ficheiro de *logs*.

6

Avaliação Experimental

Neste capítulo, apresenta-se a avaliação do sistema JepREST em aplicações REST cujas operações da API possuem uma semântica personalizada. Os resultados destes testes são analisados com o objetivo de demonstrar a capacidade que esta ferramenta tem em encontrar problemas de correção neste tipo de aplicações.

6.1 Etapas da Avaliação

A fase de avaliação do JepREST foi realizada através de testes em duas aplicações REST que partilham a mesma API, mas possuem implementações distintas.

Na primeira fase da avaliação, testámos uma aplicação desenvolvida por nós que tem uma implementação simples com três servidores, um para cada recurso. O objetivo desta fase foi verificar que o JepREST detetava problemas de correção que foram introduzidos explicitamente.

Na segunda fase, testámos uma aplicação desenvolvida por terceiros que utiliza várias réplicas e recorre ao Kafka para comunicar as alterações aplicadas ao estado. O objetivo desta fase foi testar o sistema JepREST com aplicações desenvolvidas por outros programadores. Para tal, recorreu-se aos projetos desenvolvidos na unidade curricular de Sistemas Distribuídos do curso de Licenciatura em Engenharia Informática da NOVA FCT. Aquando da entrega dos trabalhos, solicitou-se aos estudantes a autorização da utilização dos trabalhos para este efeito, apenas tendo sido usados aqueles cujos estudantes concordaram explicitamente com esta utilização.

6.2 API das Aplicações

A aplicação testada consiste numa rede social simples onde os utilizadores podem partilhar vídeos curtos (*shorts*). Os recursos usados nesta aplicação são:

- *users* - Representam os utilizadores da aplicação. Cada um destes recursos contém o identificador, a palavra-passe, o email e o nome do utilizador.

- *shorts* - Representam as informações relativas aos vídeos publicados, como o identificador do vídeo, o autor, o *timestamp* da sua publicação, o número de *likes* e o link do vídeo.
- *blobs* - Ficheiros que representam os vídeos

Além destes três recursos principais, a aplicação utiliza dois recursos chamados *following* e *likes*, que dependem dos anteriores. Concretamente, os recursos do tipo *following* dependem dos *users*, enquanto os do tipo *likes* dependem tanto dos *users* como dos *shorts*. Um recurso *following* representa que um utilizador segue outro e guarda os identificadores do seguidor e do utilizador seguido. Um recurso *likes* representa que um utilizador gosta de um *short* de outro utilizador e contém três atributos: o identificador do utilizador que gostou do *short*, o identificador do *short* e o identificador do autor do *short*.

A Figura 6.1 apresenta algumas operações da API utilizadas para testar o JepREST [1] :

- *createUser* - POST que cria um utilizador novo.
- *getUser* - GET que devolve o utilizador com o identificador indicado.
- *updateUser* - PUT que atualiza um utilizador.
- *createShort* - POST que cria um *short*.
- *getShort* - GET que retorna o *short* solicitado.
- *follow* - POST que realiza a ação de seguir ou deixar de seguir um utilizador.
- *followers* - GET que devolve os seguidores de um dado utilizador.

6.2.1 Especificação das Operações

Todas as operações desta aplicação têm um comportamento personalizado e estão especificadas utilizando a linguagem descrita na secção 4.2, exceto a operação *getShort*, que na nossa implementação segue o comportamento esperado de uma operação GET com semântica padrão.

A seguir, são explicadas em mais detalhe as operações relevantes para os testes realizados.

6.2.1.1 UpdateUser

A operação *updateUser* permite atualizador todos os atributos de um utilizador, exceto o seu identificador. Como pode ser observado na Listagem 6.1, a operação *updateUser* tem dois passos internos:

[1] A API definida tem algumas opções discutíveis do ponto de vista de modelação REST, em particular no que diz respeito às operações que definem os seguidores de cada utilizador, mas essa modelação não afeta o resultado da avaliação.

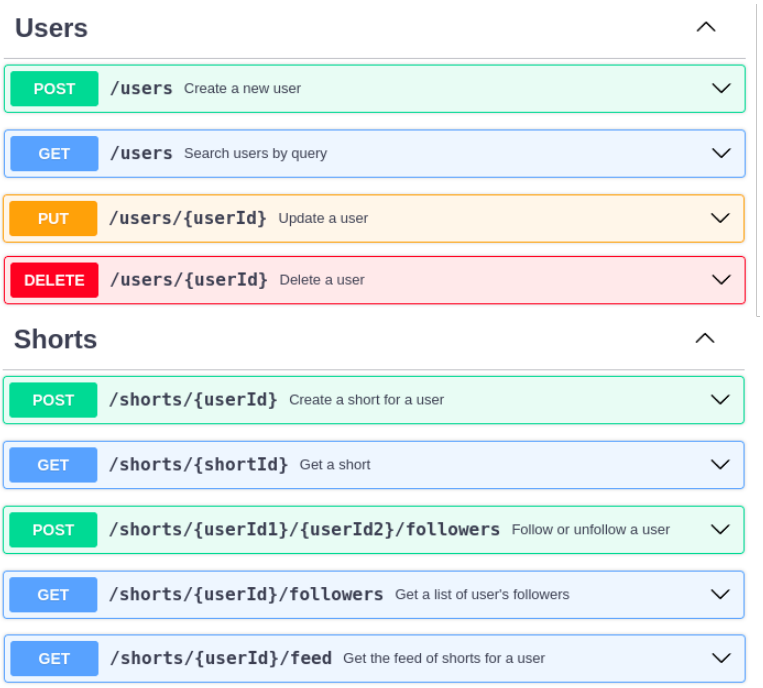


Figura 6.1: Operações da API testadas pelo JepREST.

1. O primeiro passo é um GET que devolve o utilizador com o identificador definido no parâmetro *userId* do caminho do pedido. Caso este não exista, a operação deve retornar 404.

2. O segundo passo trata da autenticação e da atualização do utilizador. Este passo começa por filtrar o utilizador do passo anterior com a palavra-passe determinada no parâmetro de pesquisa *pwd*. Se o primeiro passo devolver 200 e a filtragem não encontrar nenhum recurso, o utilizador existe, mas a palavra-passe enviada no pedido está errada. Logo, a aplicação deve retornar o código 403, pois apenas o próprio utilizador está autorizado a atualizar os seus dados.

   A seguir, se a palavra-passe estiver correta, é efetuada a atualização do utilizador. Os atributos que podem ser modificados são o *pwd*, o *email* e o *displayName*, e vão ser substituídos pelos valores correspondentes enviados no corpo do pedido.

Quando ambos os passos são bem-sucedidos, a operação retorna o recurso que representa o utilizador atualizado.

Listagem 6.1: Especificação da operação *updateUser*.

```
@PUT
@CustomMethod(operation = {
       @CustomMethodStep(operationType = HTTPVerb.GET,
           resource = "users/$request.path#/userId"),
       @CustomMethodStep(operationType = HTTPVerb.PUT, resource = "users",
           filter = @Filter(simpleFilter = {
               @SimpleFilter(attribute = "userId", value = "$request.path#/userId"),
               @SimpleFilter(attribute = "pwd", value = "$request.query#/pwd")}),
           write = @Write(simpleWrite = {
               @SimpleWrite(attribute = "pwd", value = "$request.body#/pwd"),
               @SimpleWrite(attribute = "email", value = "$request.body#/email"),
               @SimpleWrite(attribute = "displayName", value = "$request.body#/displayName")}))},
    statusCodeSeq = {@StatusCodeSequence(opStatus = 404, stepsStatusSeq = {404, -1}),
       @StatusCodeSequence(opStatus = 403, stepsStatusSeq = {200, 404})})
@Path("/{userId}") ...
User updateUser(@PathParam("userId") String id, @QueryParam("pwd") String pwd, User usr);
```

6.2.1.2 Follow

A operação *follow* é responsável por permitir que os utilizadores se sigam ou deixem de seguir. Esta operação recebe um booleano como parâmetro que, quando verdadeiro, indica que um utilizador quer seguir outro, e, quando falso, sinaliza que quer deixar de o seguir (*unfollow*).

A Listagem 6.2 mostra a especificação da operação *follow*. Esta está dividida em cinco passos distintos. Os dois primeiros representam dois GET que verificam se os utilizadores com os identificadores enviados nos parâmetros *userId1* e *userId2* existem. Caso algum destes falhe, a aplicação deve retornar 404. No terceiro passo, é realizado um GET que verifica se a palavra-passe do utilizador do primeiro passo é igual ao valor do parâmetro de pesquisa *pwd*. Se o passo não encontrar nenhum recurso, a palavra-passe está incorreta e o código enviado na resposta da operação deve ser 403.

Os dois últimos passos são mutuamente exclusivos, sendo a sua execução determinada pela condição imposta pelo valor do booleano *isFollowing*, enviado no corpo do pedido.

Quando o *isFollowing* é verdadeiro, o utilizador com *userId1* pretende seguir o utilizador com *userId2*, e é executado o penúltimo passo. Este passo corresponde a um POST que cria um recurso do tipo *following*, onde são guardados os identificadores dos utilizadores. Se o código enviado na resposta da operação é 409, este passo falhou e indica que o utilizador *userId1* já segue o *userId2*.

No cenário em que o *isFollowing* é falso, o utilizador com *userId1* deseja deixar de seguir o utilizador com *userId2*, o que leva à execução do último passo. Este passo envolve um DELETE que elimina a entrada correspondente na tabela *following* que relaciona os dois utilizadores. O resultado deste passo não influência o código enviado na resposta da operação.

Listagem 6.2: Especificação da operação *follow*.

```
@POST
@CustomMethod(operation = {
  @CustomMethodStep(operationType=HTTPVerb.GET, resource = "users/$request.path#/userId1"),
  @CustomMethodStep(operationType=HTTPVerb.GET, resource = "users/$request.path#/userId2"),
  @CustomMethodStep(operationType=HTTPVerb.GET, resource = "users",
      filter = @Filter(simpleFilter = {
         @SimpleFilter(attribute = "userId", value = "$request.path#/userId1"),
         @SimpleFilter(attribute = "pwd", value = "$request.query#/pwd") })),
  @CustomMethodStep(operationType = HTTPVerb.POST, resource = "following",
                  condition = @Condition(bool = "$request.body"),
      write = @Write(simpleWrite = {
         @SimpleWrite(attribute = "follower", value = "$request.path#/userId1", isID = true),
         @SimpleWrite(attribute = "followee", value = "$request.path#/userId2", isID = true)})),
  @CustomMethodStep(operationType = HTTPVerb.DELETE, resource = "following",
                  condition = @Condition(bool = "$request.body", result = false),
       filter = @Filter(simpleFilter = {
           @SimpleFilter(attribute = "follower", value = "$request.path#/userId1"),
           @SimpleFilter(attribute = "followee", value = "$request.path#/userId2")}))},
statusCodeSeq = {
    @StatusCodeSequence(opStatus = 404, stepsStatusSeq = {404, -1, -1, -1, -1}),
    @StatusCodeSequence(opStatus = 404, stepsStatusSeq = {200, 404, -1, -1, -1}),
    @StatusCodeSequence(opStatus = 403, stepsStatusSeq = {200, 200, 404, -1, -1}),
    @StatusCodeSequence(opStatus = 409, stepsStatusSeq = {200, 200, 200, 409, 0}),
    @StatusCodeSequence(opStatus = 204, stepsStatusSeq = {200, 200, 200, 200, 0}),
    @StatusCodeSequence(opStatus = 204, stepsStatusSeq = {200, 200, 200, 0, 1})})
@Path("/{userId1}/{userId2}/followers") ...
void follow(@PathParam("userId1") String userId1, @PathParam("userId2") String userId2,
          boolean isFollowing, @QueryParam("pwd") String password);
```

6.2.1.3 Followers

À primeira vista, a operação *followers* parece seguir a semântica da arquitetura REST, retornando apenas os recursos que representam os seguidores de um utilizador. No entanto, além de devolver uma lista de *Strings* que representam identificadores, a operação também realiza a autenticação do utilizador que solicitou a lista de seguidores. Esta ação está descrita nos dois primeiros passos da especificação, apresentada na Listagem 6.3, e ambos estão definidos como GET. O primeiro confirma se existe um recurso com o identificador enviado no parâmetro *userId*, e o segundo verifica se existe um recurso com esse identificador e com o atributo *pwd* igual ao valor do parâmetro de pesquisa *pwd*. Caso o primeiro passo falhe, o utilizador não existe e a operação deve retornar 404. Caso o segundo passo falhe, a palavra-passe enviada no pedido está errada e a resposta deve conter o código 403.

O terceiro e último passo representa a ação principal da operação. Este passo é um GET que filtra todos os recursos do tipo *following* onde o atributo *followee* contém o *userId*. Estes recursos contêm o atributo *follower* que guarda o identificador do seguidor. O resultado deste passo não é relevante para a operação, uma vez que esta retorna 200 independentemente do utilizador ter seguidores ou não. Os recursos filtrados neste passo,

ou seja, os utilizadores que seguem o utilizador que executou a operação, são armazenados na variável *foll* (linha 9).

No campo *responseCondition*, é definida uma função e os seus parâmetros, que, em conjunto, estabelecem o significado do resultado da operação (linhas 16-18). Concretamente, a função verifica se dois conjuntos são iguais, sendo os parâmetros a lista de identificadores dos seguidores, presentes na variável *foll*, e o corpo da resposta. Assim, sabe-se que a lista de *Strings* enviada no corpo da resposta corresponde ao conjunto de identificadores de utilizadores que seguem um determinado utilizador.

Listagem 6.3: Especificação da operação *followers*.

```
1  @GET
2  @CustomMethod(
3      operation = {
4          @CustomMethodStep(operationType = HTTPVerb.GET, resource = "users/$request.path#/userId"),
5          @CustomMethodStep(operationType = HTTPVerb.GET, resource = "users",
6              filter = @Filter(simpleFilter = {
7                  @SimpleFilter(attribute = "userId", value = "$request.path#/userId"),
8                  @SimpleFilter(attribute = "pwd", value = "$request.query#/pwd")})),
9          @CustomMethodStep(operationType = HTTPVerb.GET, resource = "following", storeInVar = "foll",
10             filter = @Filter(simpleFilter = {
11                         @SimpleFilter(attribute = "followee", value = "$request.path#/userId")}))},
12     statusCodeSeq = {
13         @StatusCodeSequence(opStatus = 404, stepsStatusSeq = {404, -1, -1}),
14         @StatusCodeSequence(opStatus = 403, stepsStatusSeq = {200, 404, -1}),
15         @StatusCodeSequence(opStatus = 200, stepsStatusSeq = {200, 200, 1})},
16     responseCondition = @ResponseCondition(
17         function = "(defn f [l1 l2] (= (set l1) (set l2)))",
18         functionParams = {"$stepvar.foll#/follower", "$response.body"}))
19 @Path("/{userId}/followers" ) ..
20 List<String> followers(@PathParam("userId") String userId, @QueryParam("pwd") String password);
```

6.2.1.4 GetShort

Como foi explicado anteriormente, a operação *getShort* apresenta um comportamento distinto na aplicação dos alunos em comparação com a nossa. Na nossa aplicação, a operação é um simples GET que retorna o recurso correspondente ao *short* com o identificador enviado no pedido.

Na aplicação dos alunos, a operação tem um comportamento diferente do esperado. Nesta, a operação retorna o *short* com o identificador enviado no parâmetro do pedido após aplicar uma atualização no atributo *blobUrl*. A especificação desta operação, exposta na Listagem 6.4, apresenta exatamente este comportamento, onde o primeiro passo é referente a um PATCH que atualiza o atributo correspondente, atribuindo-lhe o valor enviado no corpo da resposta, e o segundo passo representa um GET que devolve o recurso. O único caso em que a operação falha é quando o primeiro passo não encontra o recurso, fazendo com que a operação retorne um erro 404.

Listagem 6.4: Especificação da operação *getShort*.

```
@GET
@CustomMethod(operation = {
   @CustomMethodStep(operationType = HTTPVerb.PATCH, resource = "shorts/$request.path#/shortId",
          write = @Write(simpleWrite = {
                    @SimpleWrite(attribute = "blobUrl", value = "$response.body#/blobUrl")})),
   @CustomMethodStep(operationType = HTTPVerb.GET, resource = "shorts/$request.path#/shortId")},
   statusCodeSeq = {@StatusCodeSequence(opStatus = 404, stepsStatusSeq = {404, -1})})
@Path("/{shortId}" ) ...
Short getShort(@PathParam("shortId") String shortId);
```

6.3 Significado do Output Gerado pelo JepREST

Antes de explicar os testes realizados, é importante perceber o significado do *output* gerado pelo JepREST, para que se consiga perceber que operações causaram o problema de linearização. Este *output* é composto pelos *logs* gerados durante as execuções dos pedidos e pelo *output* gerado pelo Knossos. A última linha do *output* da ferramenta contém uma frase que indica se o teste teve sucesso (“*Everything looks good!*”) ou se foi encontrado algum problema de linearização (“*Analysis invalid!*”).

A primeira parte do *output* mostra os *logs* dos pedidos realizados pelos clientes do JepREST. As informações apresentadas representam operações do Jepsen do tipo *:invoke*, *:ok*, ou *:fail*, e contêm os dados enviados nos pedidos, as informações recolhidas na especificação e os dados recebidos nas respostas. A Listagem 6.5 mostra um exemplo desta parte do *output*.

Listagem 6.5: Excerto da primeira parte do *output* gerado pelo JepREST.

```
INFO [2024-09-19 19:13:30,639] jepsen worker 4 - jepsen.util 4 :invoke :post
   {:custom-op {...}, :typeOp "createUser", :input {:body {:userId "Kadin", ...}}}
INFO [2024-09-19 19:13:30,642] jepsen worker 1 - jepsen.util 1 :invoke :get
   {:custom-op {...}, :input {:typeOp "getUser", :path {:userId "Colten"}, :query {:pwd "JGhb3"}}}
INFO [2024-09-19 19:13:30,645] jepsen worker 0 - jepsen.util 0 :ok :get
   {:custom-op {...}, :input {:typeOp "followers", :path {:userId "Abdullah"}, :query {:pwd "Yzu6"}},
   :url "http://host.docker.internal:8081/", :output {:status 200, :headers {...}, :body ()}}
INFO [2024-09-19 19:13:30,650] jepsen worker 4 - jepsen.util 4 :ok :post
   {:custom-op {...}, :typeOp "createUser", :input {:body {:userId "Kadin", ...}},
   :url "http://host.docker.internal:8080/", :output {:status 200, :headers {...}, :body "Kadin"}}
```

O output gerado pelo Knossos apresenta um mapa que inclui um conjunto de informações associadas às chaves *:valid?*, *:configs*, *:final-paths* e *:model*:

- O valor associado à chave *:valid?* é um booleano que indica se a história gerada durante a fase de teste é linearizável ou não.
- A chave *:configs* armazena um conjunto de configurações que mostram o estado da aplicação no momento em que foi identificada a incoerência. Cada configuração inclui três elementos: a última operação que a aplicação executou corretamente,

o estado do modelo antes da execução da operação incorreta e as operações que ainda estavam pendentes no momento da falha, ou seja, que ainda não tinham sido concluídas.

Como o Knossos analisa várias ordenações das operações para determinar se existe um caminho linearizável, as diferentes configurações podem conter ordens distintas das operações pendentes e também podem diferir na última operação sinalizada como correta.

Se a história é linearizável, não existem operações pendentes e apenas existe uma última operação correta, que é a última da história, logo, o valor associado a esta chave é um conjunto vazio.

- A chave *:final-paths* representa o conjunto de ordenações distintas das operações concorrentes no momento do erro, isto é, as operações pendentes presentes no campo *:configs*, e mostra como cada uma dessas ordenações resultou numa transição para um estado ilegal. Se a história é considerada linearizável, existe pelo menos uma ordem das operações que leva a um estado final correto, o que significa que este conjunto fica vazio.
- A chave *:model* é apresentada apenas quando o Knossos não encontra nenhuma incoerência entre as operações efetuadas na história. Esta representa o estado final da aplicação.

Quando a história analisada pelo Knossos não é linearizável, são apresentadas três informações adicionais, representadas com as chaves *:previous-ok*, *:last-op* e *:op*:

- A chave *:previous-ok* representa a última operação do tipo *:ok* sinalizada como linearizável, que indica que todas as operações anteriores a essa são linearizáveis.
- A chave *:last-op* representa a última operação que a aplicação executou no momento em que o Knossos relata uma incoerência. Muitas vezes coincide com a operação *:previous-ok*.
- A chave *:op* representa a primeira operação *:ok* que o Knossos não conseguiu linearizar.

6.4 Testes e Resultados

Nesta secção, apresentamos os testes realizados pelo JepREST nas aplicações mencionadas anteriormente, bem como a análise dos *outputs* gerados pela ferramenta.

6.4.1 Testes Realizados à Aplicação Instrumentada

A nossa aplicação é implementada com três servidores, sendo cada um responsável por um recurso distinto. Para isso, foram criados três *containers Docker*: um para os recursos

users, outro para os *shorts* e um último para os *blobs*. Estes *containers* foram executados na mesma máquina local que o JepREST. Nos servidores de *users* e *shorts*, os recursos são armazenados numa base de dados HyperSQL [33], com todas as operações de escrita protegidas por transações. Os recursos *blobs*, por sua vez, são armazenados no *file system* do servidor correspondente. Já os recursos *following* e *likes* são armazenados em tabelas de relação um-para-um dentro do servidor de *shorts*. A Figura 6.2 apresenta a arquitetura desta aplicação.

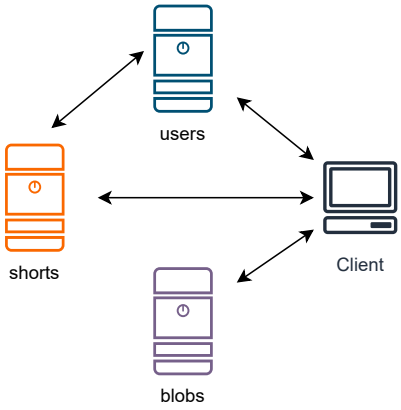


Figura 6.2: Arquitetura da nossa implementação da aplicação.

Para esta aplicação foram realizados três testes que cobrem partes distintas do código. O primeiro teste foi realizado na aplicação com a sua versão final implementada. Nos dois testes subsequentes, foram injetados erros na aplicação para verificar se o JepREST seria capaz de os detetar.

Nas secções seguintes são descritos os testes e os resultados apresentados pela ferramenta.

6.4.1.1 Atualizações Concorrentes de um Utilizador

Neste primeiro teste, utilizámos o ficheiro YAML apresentado na Listagem 6.6, para verificar se existe algum problema quando dois clientes distintos atualizam um mesmo utilizador. O ficheiro apenas contém um cenário onde três clientes realizam pedidos concorrentes à aplicação. Um dos clientes executa a operação *createUser*, enquanto que os outros dois efetuam a operação *updateUser* simultaneamente. Este comportamento é realizado a cada segundo e, como não foi definido um tempo total de execução no argumento "-tl" ou "–time-limit", o teste é efetuado durante 5 segundos, que é o tempo padrão definido pelo JepREST.

Ao executar este teste no JepREST, é detetado um problema de linearização entre duas operações *updateUser* que alteram a palavra-passe de um utilizador. Para demonstrar qual

foi o erro encontrado, é analisada a sequência de operações presente no campo *:final-paths* gerado pelo Knossos, apresentada na Listagem 6.7.

Listagem 6.6: Ficheiro de teste YAML usado para testar a atualização concorrente de um utilizador.

```
1 scenarios:
2   - name: 'Test Update User'
3     weight: 100
4     flow:
5       - createUserData
6       - updateUserData
7       - updateUserData
```

Esta sequência contém duas operações *updateUser* executadas pelos clientes com identificadores 2 e 3. O *updateUser* efetuado pelo cliente 2 encontra-se na oitava posição da história (linha 10), e, logo a seguir, na nona posição (linha 21), aparece o *updateUser* realizado pelo cliente 3. Ambos os clientes pediram à aplicação para atualizar vários dados do utilizador com o identificador "*Gust*", incluindo a palavra-passe, e enviaram no parâmetro de pesquisa *pwd* a palavra-passe correta no momento da invocação dos pedidos, cujo valor era "*26B3Pw8*".

A mensagem de erro associada à operação realizada pelo cliente 3 (linhas 22 e 23) indica que o problema de coerência está relacionado com o segundo passo (os identificadores dos passos começam no 0, por isso, o *STEP* 1 é o segundo passo). Como a operação retornou o código 200, todos os passos deveriam ter sido bem-sucedidos, mas o segundo passo não encontrou nenhum recurso onde o atributo *userId* é "*Gust*" e o *pwd* é "*26B3Pw8*".

O que aconteceu foi que dois clientes pediram para atualizar a palavra-passe do mesmo utilizador, e ambos tiveram sucesso. De acordo com a especificação da operação *updateUser*, definida na Listagem 6.1, a operação só pode ter sucesso quando a palavra-passe está correta. O *updateUser* do cliente 2 foi finalizado primeiro, o que levou ao estado apresentado no campo *:model*, nas linhas 11 e 12. É percetível que a palavra-passe do utilizador com *userId* "*Gust*" já não é "*26B3Pw8*", mas sim "*7rKu9i1*". Logo, o pedido do cliente 3 deveria ter resultado num erro 403, pois a palavra-passe mudou para o valor definido pelo cliente 2. O pedido do cliente 3 só poderia ter sucesso se a palavra-passe enviada no parâmetro de pesquisa *pwd* fosse "*7rKu9i1*".

Isto demonstra que existe um problema de linearização relacionado com a operação *updateUser*. Verificámos, por isso, o código da aplicação e descobrimos que a parte da autenticação é feita fora da transação. O que acontece quando são realizados dois pedidos concorrentes relativos à operação *updateUser* é que ambos os pedidos podem passar com sucesso a parte da autenticação. Depois, um dos pedidos bloqueia o recurso e modifica a palavra-passe, enquanto o outro fica à espera. Após a atualização, o recurso é desbloqueado e a operação é concluída. O problema está em que, a partir desse momento, a palavra-passe do utilizador já não é a mesma, e a autenticação do pedido que ficou à espera deveria falhar. No entanto, como a autenticação já tinha sido realizada, o pedido prossegue com a

atualização, e a palavra-passe é alterada novamente.

Listagem 6.7: Sequência de operações *updateUser* não linearizável.

```
:final-paths
 ([{:op
    {:process 2, :type :ok, :f :put,
     :value{:custom-op {...},
      :input {:typeOp "updateUser", :path {:userId "Gust"}, :query {:pwd "26B3Pw8"},
              :body {:email "koby@hotmail.com", :userId "Gust",
                     :displayName "Krystel Muller", :pwd "7rKu9i1"}},
      :output {:status 200, :headers {...}, :body {:userId "Gust", :pwd "7rKu9i1",
                :email "koby@hotmail.com", :displayName "Krystel Muller"}}},
     :index 8, :time 2336937650},
    :model {:allJsons {"users" {"Gust" {:displayName "Krystel Muller", :pwd "7rKu9i1",
                                        :userId "Gust", :email "koby@hotmail.com"}}}}}
   {:op
    {:process 3, :type :ok, :f :put,
     :value {:custom-op {...},
      :input {:typeOp "updateUser", :path {:userId "Gust"}, :query {:pwd "26B3Pw8"}
      :body {:email "yvette@hotmail.com", :userId "Gust",
             :displayName "Patsy Stoltenberg", :pwd "526PRW"}},
      :output {:status 200, :headers {...}, :body {:userId "Gust", :pwd "526PRW",
                :email "yvette@hotmail.com", :displayName "Patsy Stoltenberg"}}},
     :index 9, :time 2337024794},
    :model {:msg "[STEP 1] PUT: this step was supposed to
                  be successful but there are no resources"}}])
```

6.4.1.2 Bloqueio da Ação de *Unfollow*

Para a realização deste teste, determinámos que a operação *follow* não pode ser usada para deixar de seguir um utilizador. Assim, independentemente do valor do booleano enviado no pedido, a operação apenas executa a ação de seguir. Ou seja, durante este teste, quando um utilizador segue outro, essa ligação nunca desaparece.

Para testar esta operação e verificar se o JepREST consegue identificar o erro injetado, utilizámos a ferramenta que gera o grafo de transições das operações a partir dos *logs* da aplicação (apresentada na secção 4.3), para definir as operações que os clientes devem efetuar. Estes *logs* foram gerados durante uma execução prévia da aplicação. A Figura 6.3 representa visualmente o grafo gerado, que foi usado pelo JepREST na fase de teste para efetuar pedidos à aplicação.

A Listagem 6.8 mostra a parte final do *output* gerado pelo JepREST, que, como o valor correspondente à chave *:valid?* é *false* (linha 9), demonstra que a ferramenta encontrou um erro associado à operação presente no campo *:op*. De facto, a operação que falhou representa um *unfollow*, onde a utilizadora "*Lisette*" pediu para deixar de seguir a "*Clemmie*".

A mensagem de erro associada à operação dentro do campo *:final-paths* é a seguinte: "*[STEP 3] -> The condition of this step was not supposed to fail, but failed.*". Esta mensagem indica que o quarto passo falhou porque a condição definida na especificação não foi cumprida, mas mesmo assim foi executado. Sabemos que foi executado porque, como

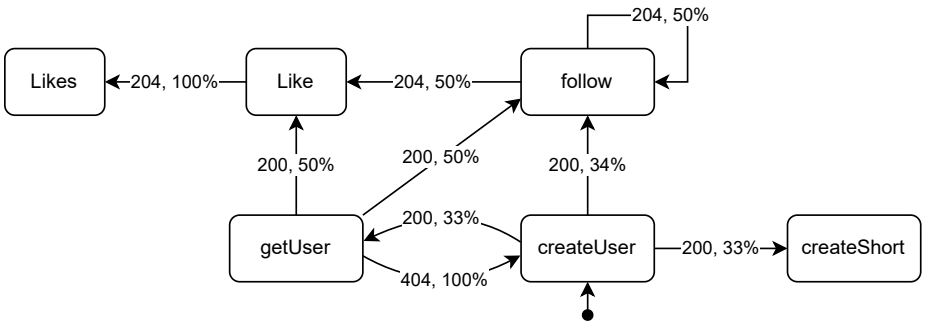


Figura 6.3: 1º Grafo utilizado para gerar as operações executadas pelos clientes.

se pode observar na linha 23 da Listagem 6.2, a única forma da operação retornar 409 é se o quarto passo resultar também num 409. No entanto, este passo não deveria ter sido executado, pois o valor enviado no corpo do pedido é *false*. Deste modo, conseguimos perceber que a implementação da operação não é coerente com a especificação.

Listagem 6.8: Parte final do *output* gerado pelo JepREST.

```
...
:op {:process 2, :type :ok, :f :post,
     :value {:custom-op {...},
        :input {:body false, :typeOp "follow",
               :path {:userId1 "Lisette", :userId2 "Clemmie"}, :query {:pwd "mO7mFEm"}},
        :output {:status 409, :headers {...}, :body ""}},
    :index 419, :time 6094395471}},
:timeline {:valid? true},
:valid? false}
```

Para perceber a razão que levou o JepREST a descobrir este erro, analisámos a parte do *output* que contém as sequências de operações do Jepsen *:invoke* e *:ok* que foram efetuadas durante a fase de teste. A Listagem 6.9 mostra a ordem em que as operações *follow* foram executadas pelo cliente do JepREST com identificador 2, apenas com alguns milissegundos de diferença. Primeiro, às 18:22:30,110, o cliente 2 efetua um pedido para que a "*Lisette*" siga a "*Clemmie*". A seguir, passados 47 milissegundos, a aplicação responde com o código 204, que indica que, a partir desse momento, a "*Lisette*" segue a "*Clemmie*". 206 milissegundos mais tarde, o cliente 2 realiza novamente um *follow* com os mesmos identificadores, mas, desta vez, o corpo do pedido é enviado com o valor *false*, ou seja, o cliente 2 pede para que a "*Lisette*" deixe de seguir a "*Clemmie*". Por fim, o cliente recebe a resposta do *unfollow*, onde está presente o código 409, que, segundo a especificação, nunca poderia acontecer nesta situação.

O Knossos, no campo *:previous-ok*, apresenta uma operação que foi realizada depois do *:ok* do primeiro *follow*. O *:previous-ok* indica qual foi a última operação que o Knossos conseguiu linearizar, o que significa que todas as operações anteriores também são linearizáveis. Deste modo, no momento da verificação do *:ok* correspondente ao *unfollow*, o estado já contém a relação entre a "*Lisette*" e a "*Clemmie*". Neste ponto, o Knossos executa

várias sequências diferentes das operações pendentes, com o objetivo de encontrar uma em que código 409 do *unfollow* seja possível. Contudo, nenhuma é encontrada e o Knossos finaliza a sua execução, explicando que o *unfollow* falhou de maneira inesperada.

Listagem 6.9: Ordem das operações *follow* realizadas pelos clientes do JepREST.

```
INFO [2024-09-22 18:22:30,110] jepsen worker 2 - jepsen.util 2 :invoke :post {:custom-op {...},
       :input {:body true, :typeOp "follow", :path {:userId1 "Lisette", :userId2 "Clemmie"},
               :query {:pwd "mO7mFEm"}}}
...
INFO [2024-09-22 18:22:30,157] jepsen worker 2 - jepsen.util 2 :ok :post {:custom-op {...},
       :input {:body true, :typeOp "follow", :path {:userId1 "Lisette", :userId2 "Clemmie"},
               :query {:pwd "mO7mFEm"}}, :output {:status 204, :headers {..}, :body nil}}
...
INFO [2024-09-22 18:22:30,363] jepsen worker 2 - jepsen.util 2 :invoke :post {:custom-op {...},
       :input {:body false, :typeOp "follow", :path {:userId1 "Lisette", :userId2 "Clemmie"},
               :query {:pwd "mO7mFEm"}}}
...
INFO [2024-09-22 18:22:30,410] jepsen worker 2 - jepsen.util 2 :ok :post {:custom-op {...},
       :input {:body false, :typeOp "follow", :path {:userId1 "Lisette", :userId2 "Clemmie"},
               :query {:pwd "mO7mFEm"}}, :output {:status 409, :headers {...}, :body ""}}
```

Este teste demonstra que o JepREST é capaz de identificar erros na aplicação que fazem com que as operações não se comportem conforme especificado.

6.4.1.3 Deteção de Dados Incorretos nas Respostas

Nesta secção, é apresentado um teste que foi realizado para verificar se o JepREST consegue identificar um problema de linearização quando a resposta de uma operação apresenta dados incorretos.

Para isto, injetámos um erro na operação *followers*. Esta operação retorna uma lista de identificadores dos utilizadores que seguem outro utilizador. A modificação adicionada remove o primeiro elemento da lista de seguidores que é enviada na resposta.

As operações executadas pelos clientes do JepREST, para comprovar se a ferramenta consegue detetar o erro injetado, foram geradas a partir do grafo apresentado na Figura 6.4.

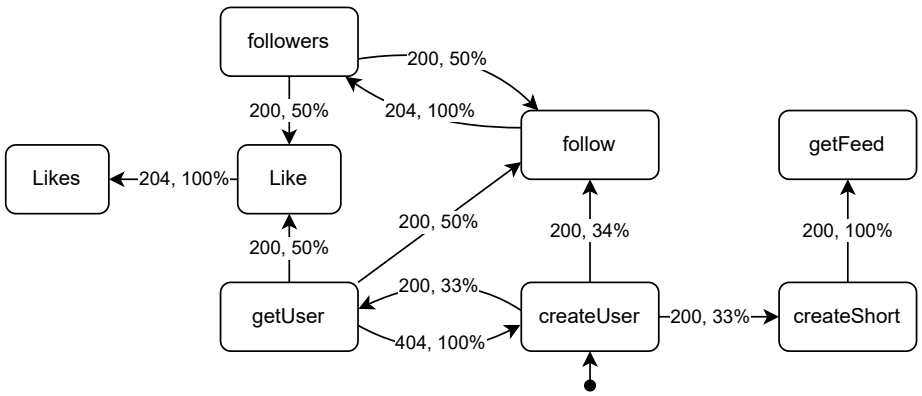


Figura 6.4: 2º Grafo utilizado para gerar as operações executadas pelos clientes.

A Listagem 6.10 apresenta a parte final do campo *:final-paths* gerado pelo Knossos durante a avaliação deste teste, e representa o resultado da operação após executar uma sequência de operações que estavam pendentes de linearizar. Todas as sequências de operações pendentes conduzem exatamente ao estado apresentado na listagem, embora algumas delas mostrem diferenças em determinados recursos. Contudo, todas as sequências levam a um estado em que o utilizador "*Trevor*" tem dois seguidores, "*Clara*" e "*Green*" (linhas 3 e 4).

A operação que falhou foi um *followers* com *userId* igual a "*Trevor*". A razão deste falho é apresentada na mensagem dentro campo *:model* (linhas 9 e 10). De acordo com a mensagem, a verificação encontrou uma incoerência entre os dados presentes na resposta e os armazenados no estado mantido pelo Knossos. Ao analisar o corpo da resposta e o estado gerado pelas operações anteriores, verifica-se que a lista de seguidores do utilizador "*Trevor*" apenas contém um valor, "*Green*", quando o estado contém, além deste, o valor "*Clara*".

Listagem 6.10: Parte final do campo *:final-paths* gerado pelo Knossos.

```
1  :final-paths (...,
2  [{ ... :model {:allJsons {"shorts" {...}, "users" {...}, "likes" {...},
3          "following" {"Clara+Trevor" {:follower "Clara", :followee "Trevor"}, ...,
4                       "Green+Trevor" {:follower "Green", :followee "Trevor"}, ...}}}},
5  {:op {:process 2, :type :ok, :f :get,
6        :value {:custom-op {...},
7            :input {:typeOp "followers", :path {:userId "Trevor"}, :query {:pwd "2zGh13C"}},
8            :output {:status 200, :headers {...}, :body ("Green")}}, :index 115, :time 2709674955},
9   :model {:msg
10   "[RESPONSE CONDITION] - Values in the response don't match the ones stored in the local state."}}])
```

Analisando a sequência de operações Jepsen executadas pelos clientes do JepREST, é fácil perceber a razão do problema de linearização. A primeira operação é um *:ok* relativo à operação *follow* que indica que a utilizadora "*Clara*" começou a seguir o utilizador "*Trevor*". A seguir, aparece outro *:ok* relativo à operação *follow*, que indica que o utilizador "*Green*" também começou a seguir o utilizador "*Trevor*". Logo após estas duas operações bem-sucedidas, é invocada uma operação para obter os seguidores do "*Trevor*". A resposta a esta operação contém apenas o "*Green*" como seguidor, quando o esperado seria que ambos, "*Clara*" e "*Green*", aparecessem na lista.

Este comportamento indica uma falha na linearização, uma vez que, do ponto de vista lógico, ambas as operações *follow* foram executadas com sucesso e todas as operações invocadas a seguir deveriam refletir o estado gerado por essas operações.

O resultado deste teste comprova que o JepREST consegue descobrir incoerências relativas aos dados recebidos nas respostas das operações.

Listagem 6.11: Sequência de operações executadas pelos clientes que descobriu o erro na operação *followers*.

```
[19:13:30,650] jepsen worker 2 - :ok :post {:input {:body true, :typeOp "follow",
              :path {:userId1 "Clara", :userId2 "Trevor"}, ...}, :output {:status 204, ...}}
[19:13:30,887] jepsen worker 2 - :ok :post {:input {:body true, :typeOp "follow",
              :path {:userId1 "Green", :userId2 "Trevor"}, ...}, :output {:status 204, ...}}

[19:13:30,890] jepsen worker 2 - :invoke :get {:input {:typeOp "followers", :path {:userId "Trevor"},
                                                       :query {:pwd "2zGh13C"}}}
[19:13:30,912] jepsen worker 2 - :ok :get {:input {:typeOp "followers", :path {:userId "Trevor"},
                                          :output {:status 200, :body ("Green"), ...}}
```

6.4.2 Teste Realizado à Aplicação de Terceiros

Nesta secção é apresentado o resultado do uso do JepREST para testar a correção de uma aplicação implementada por outros programadores.

A aplicação testada utiliza a mesma API que a aplicação da secção anterior, mas a sua implementação é replicada. Esta recorre ao sistema de mensagens distribuído Kafka, baseado no modelo *publish/subscribe*, para propagar as operações realizadas pelos clientes às diferentes réplicas.

A aplicação consiste em cinco servidores: um para os *users*, um para os *blobs* e três para os *shorts*. Tanto estes servidores como o Kafka foram executados em *containers* Docker na mesma máquina local que o JepREST. Cada um dos servidores de *shorts* é simultaneamente *publisher* e *subscriber* do tópico *shorts*. Tal como na nossa aplicação, os recursos *users*, *shorts*, *following* e *likes* são armazenados em bases de dados HyperSQL e os *blobs* em ficheiros. Dentro dos servidores de *users* e *shorts*, todas as operações de escrita são realizadas no contexto de transações. A Figura 6.5 mostra a arquitetura desta aplicação.

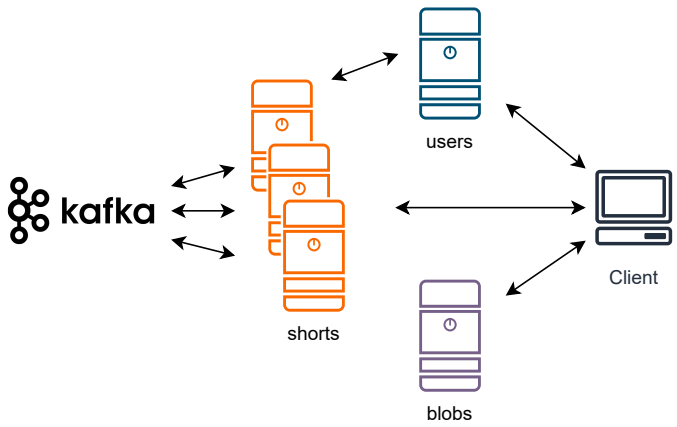


Figura 6.5: Arquitetura da aplicação implementada pelos alunos.

Para gerar as operações que os clientes realizaram à aplicação durante o teste, foi utilizado o grafo da Figura 6.6.

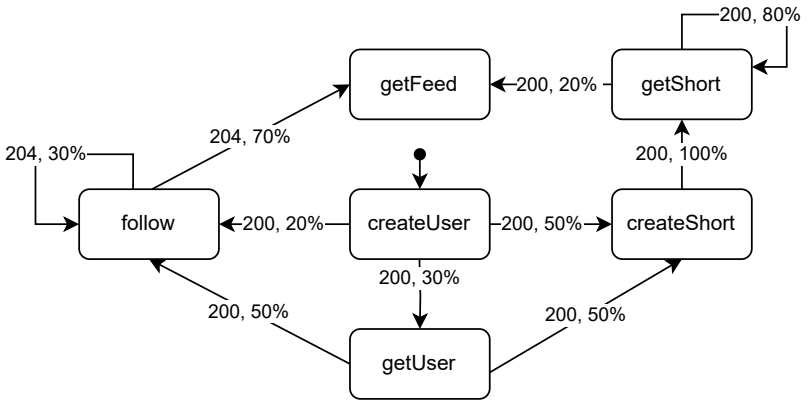


Figura 6.6: 3º Grafo utilizado para gerar as operações executadas pelos clientes.

O problema de coerência detetado pelo JepREST durante este teste está relacionado com a operação *getShort*, mais especificamente, quando esta é executada em várias réplicas imediatamente após a criação do recurso.

A Listagem 6.12 apresenta a parte final de uma das sequências de operações que o Knossos não conseguiu lienarizar. A primeira operação representa um *getShort* com o identificador "*Barbara-4*", realizado pelo cliente 2 ao servidor de *shorts* mapeado para a porta 8089 do *host*. A operação seguinte é um *getShort* idêntico ao anterior, mas realizado pelo cliente 3 ao servidor mapeado para a porta 8087. A ordem das operações é determinada pelos índices apresentados nas linhas 7 e 15.

A incoerência detetada pelo Knossos reside no facto do primeiro *getShort* ter retornado uma resposta de sucesso com o recurso solicitado, enquanto o segundo *getShort* indicou que o recurso não existe. Ou seja, duas operações concorrentes que deveriam devolver o mesmo resultado acabaram por devolver resultados opostos. Concretamente, dois clientes pediram o *short* com o identificador "*Barbara-4*" a duas réplicas distintas e obtiveram resultados diferentes, o que indica que, no momento da execução destas operações, as réplicas têm estados diferentes. Logo, a história gerada pelo JepREST não é linearizável.

De modo a perceber melhor a razão da incoerência detetada pelo Knossos, analisámos a sequência de operações efetuadas pelos clientes do JepREST, apresentada na Listagem 6.13.

Nesta sequência de operações, o cliente 2 começa por realizar um pedido para criar um *short* com o identificador "*Barbara-4*" na réplica mapeada para a porta 8088, recebendo uma resposta de sucesso que confirma a criação do recurso. Logo após, o mesmo cliente 2 faz um pedido para recuperar o *short* na réplica mapeada para a porta 8089, enquanto o cliente 3 faz um pedido semelhante na réplica mapeada para a porta 8087. A réplica da porta 8089 responde com sucesso, devolvendo o *short* criado, mas a réplica da porta 8087 responde com um erro 404, indicando que ainda não está sincronizada com a criação do recurso. Isto apresenta uma incoerência entre os estados de cada réplica num dado momento, infringindo assim as regras de linearizabilidade.

Essencialmente, a operação de criação de um *short* não garante que todas as leituras

subsequentes, efetuadas em qualquer outra réplica, reflitam o recurso criado.

O resultado deste teste permite concluir que o JepREST é capaz de detetar problemas de correção em aplicações independentes apenas fornecendo a sua API.

Listagem 6.12: Parte final do campo *:final-paths* gerado pelo Knossos.

```
:final-paths (..., [...,
{:op
 {:process 2, :type :invoke, :f :get,
  :value {:custom-op {...}, :url "http://host.docker.internal:8089/",
          :input {:typeOp "getShort", :path {:shortId "Barbara-4"}},
          :output {:status 200, ..., :body {:shortId "Barbara-4", :ownerId "Barbara", ...}}},
  :index 24, :time 2809024517},
 :model {:allJsons
  {"shorts" {"Barbara-4" {:shortId "Barbara-4", :ownerId "Barbara", ...}, ...}, "users" {...}}}}}
{:op
 {:process 3, :type :ok, :f :get,
  :value {:custom-op {...}, :url "http://host.docker.internal:8087/",
          :input {:typeOp "getShort", :path {:shortId "Barbara-4"}},
          :output {:status 404, :headers {...}},
  :index 33, :time 3026532279},
 :model {:msg
  "[STEP 0] GET: the step returned status code 404 but the resources exist: {\"Barbara-4\" {:shortId
      ↪ \"Barbara-4\", :totalLikes 0, :blobUrl \"http://7013b3c42026:5678/rest/blobs/Barbara-4?vrf
      ↪ =-1592911125_1726584607218\", :timestamp 1726584606406, :ownerId \"Barbara\"}}"}}])
```

Listagem 6.13: Sequência de operações que gerou uma incoerência entre réplicas.

```
[14:50:06,201] worker 2 - :invoke :post {:input {:typeOp "createShort",
                 :path {:userId "Barbara"}, :query {:pwd "cVnRf5G"}
                 :url "http://host.docker.internal:8088/"}}
[14:50:06,684] worker 2 - :ok :post {:input {:typeOp "createShort",
                 :path {:userId "Barbara"}, :query {:pwd "cVnRf5G"}},
                 :url "http://host.docker.internal:8088/",
                 :output {:status 200, ..., :body {:shortId "Barbara-4", :ownerId "Barbara", ...}}}
[14:50:06,685] worker 2 - :invoke :get {:input {:typeOp "getShort", :path {:shortId "Barbara-4"}}
                 :url "http://host.docker.internal:8089/",}
[14:50:06,831] worker 3 - :invoke :get {:input {:typeOp "getShort", :path {:shortId "Barbara-4"}}
                 :url "http://host.docker.internal:8087/",}
[14:50:06,902] worker 3 - :ok :get {:input {:typeOp "getShort", :path {:shortId "Barbara-4"}},
                 :url "http://host.docker.internal:8087/",
                 :output {:status 404, ..., :body ""}}
[14:50:07,236] worker 2 - :ok :get {:input {:typeOp "getShort", :path {:shortId "Barbara-4"}},
                 :url "http://host.docker.internal:8089/",
                 :output {:status 200, ..., :body {:shortId "Barbara-4", :ownerId "Barbara", ...}}}
```

7

Conclusão

Neste trabalho, estendemos as funcionalidades do JepREST com o objetivo de testar a correção de aplicações REST que oferecem operações com funcionalidades personalizadas. Como a especificação OpenAPI não tem capacidade de descrever este tipo de operações, implementámos uma linguagem própria para esse fim. Para realizar testes mais realistas, criámos um sistema de geração de *workloads* baseado nos *logs* da aplicação em teste.

7.1 Considerações Finais

O JepREST é uma ferramenta de teste *black-box* destinada a verificar a correção de aplicações REST. Esta ferramenta simplifica o processo de teste e a verificação dos resultados gerados. Para tal, são criados múltiplos clientes que executam pedidos concorrentes, simulando a execução da aplicação num ambiente real. Uma vez terminada a fase de execução dos testes, a ferramenta analisa os resultados para identificar possíveis erros.

Numa aplicação que oferece operações com funcionalidades personalizadas, saber se o resultado das operações está correto não é uma tarefa trivial. É necessário considerar não apenas os resultados esperados, mas também como essas funcionalidades personalizadas afetam o estado da aplicação. Neste trabalho, criámos uma linguagem capaz de especificar o comportamento interno das operações de uma API REST. Esta linguagem permite descrever as operações da API como sequências de operações REST simples e especificar o resultado que estas devem ter para cada código HTTP da resposta.

A história analisada pelo JepREST é gerada pelos clientes durante a fase de teste. Como o objetivo desta ferramenta é testar uma aplicação de forma semelhante à utilização esperada, as sequências de pedidos executadas pelos clientes devem ser coerentes. Por isso, desenvolvemos uma ferramenta que gera um grafo de transições de operações a partir dos *logs* da aplicação. Esse grafo define, com base nos resultados de cada operação, as probabilidades de execução das operações subsequentes.

Para determinar as operações que os clientes do JepREST devem executar em cada momento, definimos um novo gerador de operações do Jepsen. Este gerador interage com o grafo de transições de operações sempre que um cliente recebe uma resposta,

fornecendo o identificador da operação e o código HTTP. Com base nessas informações, o grafo retorna o identificador da próxima operação, que é então atribuída ao cliente correspondente, garantindo uma sequência coerente e dinâmica de operações durante o teste.

A junção do grafo de transições de operações da API com o novo gerador de operações do Jepsen permite simular o comportamento real dos clientes da aplicação em teste. Porém, há uma contrapartida: quando o utilizador do JepREST não tem acesso aos *logs* da aplicação, o grafo não pode ser gerado. Nestes casos, utiliza-se o ficheiro YAML estilo Artillery, conforme definido na primeira versão da ferramenta.

Para analisar a correção dos resultados, o JepREST utiliza um *checker* do Jepsen chamado Knossos, que verifica se uma história é linearizável com um modelo predefinido. Para que o Knossos pudesse testar aplicações REST com operações personalizadas, ampliámos o modelo criado na primeira versão do JepREST para a suportar a linguagem que definimos. Desta forma, o *checker* é capaz avaliar cada passo interno das operações da API e entender como a sua execução altera o estado da aplicação.

A avaliação realizada ao JepREST focou-se em comprovar que a ferramenta consegue detetar problemas de coerência numa aplicação REST com funcionalidades personalizadas. Para esse efeito, utilizámos uma aplicação com duas implementações distintas: uma desenvolvida por nós e outra criada por outros programadores. Os resultados dos testes efetuados demonstram que a ferramenta consegue identificar problemas de correção causados por variados tipos de erros.

A capacidade do JepREST para detetar incoerências torna-o uma ferramenta útil para identificar erros de correção em aplicações REST.

7.2 Trabalho futuro

No decurso deste trabalho identificámos algumas melhorias e adições que podem ser implementadas em trabalhos futuros:

- O *checker* utilizado pelo JepREST verifica se uma história é linearizável, o que limita a quantidade de aplicações que podem ser testadas pela ferramenta. Em trabalhos futuros, poderiam ser definidos novos *checkers* que validem outros níveis de coerência.

- Muitas aplicações optam por usar algum protocolo de autenticação para garantir a segurança da API. Uma melhoria possível seria implementar um mecanismo de suporte de protocolos de segurança, como o OAuth 2.0 ou o JWT (JSON Web Tokens). Para isso, o utilizador poderia definir, num ficheiro de configurações, todas as informações sobre segurança necessárias para que os clientes do JepREST as incluam nos pedidos realizados.

- A linguagem implementada neste trabalho permite especificar operações de aplicações com complexidade semelhante à das utilizadas na avaliação da ferramenta. Seria interessante utilizar esta linguagem para especificar aplicações mais complexas, a fim de explorar os seus limites atuais e expandi-la, de modo a abranger o maior número possível de aplicações.

- O sistema de geração de *workloads* pode ser expandido com diversas melhorias, que tornariam os testes executados pelos clientes do JepREST ainda mais realistas. Uma dessas melhorias seria permitir que os clientes utilizassem dados contidos nas respostas de operações anteriores nas operações subsequentes. Além disso, a probabilidade de transitar de uma operação para outra poderia depender de mais fatores além do código da resposta, como o número total de operações realizadas, dados específicos presentes nas respostas, ou, até, o tempo de resposta das operações.

Bibliografia

[1] *2023 State of the API Report*. Rel. téc. Postman, Inc., 2023. URL: https://www.postman.com/state-of-api/api-technologies/#api-technologies (ver p. 2).

[2] M. Andronache. *JAX-RS is just an API!* Jul. de 2022. URL: https://www.baeldung.com/jax-rs-spec-and-implementations (acedido em 3 de dez. de 2023) (ver p. 7).

[3] *Apache CXF™: An Open-Source Services Framework*. URL: https://cxf.apache.org/ (acedido em 3 de dez. de 2023) (ver p. 7).

[4] A. Arcuri. «RESTful API Automated Test Case Generation». Em: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 2017, pp. 9–20. DOI: 10.1109/QRS.2017.11 (ver p. 18).

[5] A. Arcuri e J. P. Galeotti. «SQL data generation to enhance search-based system testing». Em: jul. de 2019, pp. 1390–1398. ISBN: 978-1-4503-6111-8. DOI: 10.1145/3321707.3321732 (ver p. 18).

[6] *Artillery*. URL: https://www.artillery.io/ (acedido em 14 de jan. de 2024) (ver p. 16).

[7] *Cheshire*. URL: https://github.com/dakrone/cheshire (acedido em 9 de set. de 2024) (ver p. 58).

[8] *clj-http*. URL: https://clojars.org/clj-http (acedido em 9 de set. de 2024) (ver p. 57).

[9] Codecademy Team. *What is REST?* URL: https://www.codecademy.com/article/what-is-rest (acedido em 15 de nov. de 2023) (ver p. 6).

[10] *Consistency Models. Histories*. URL: https://jepsen.io/consistency#histories (acedido em 18 de jan. de 2024) (ver p. 17).

[11] D. Corradini et al. «RestTestGen: An Extensible Framework for Automated Black-box Testing of RESTful APIs». Em: out. de 2022, pp. 504–508. DOI: 10.1109/ICSME55016.2022.00068 (ver p. 20).

[12] B. Crist. *Practice Jepsen Test Framework in Nebula Graph*. 2021. URL: https://morioh.com/a/5298629fafb4/practice-jepsen-test-framework-in-nebula-graph (acedido em 18 de jan. de 2024) (ver p. 17).

[13] *cURL*. URL: https://curl.se/ (acedido em 9 de jan. de 2024) (ver p. 14).

[14] *cURL: Command line HTTP*. URL: https://everything.curl.dev/http (acedido em 10 de jan. de 2024) (ver p. 14).

[15] *cURL: What does curl do?* URL: https://everything.curl.dev/project/does (acedido em 9 de jan. de 2024) (ver p. 14).

[16] G. Deng et al. *PentestGPT: An LLM-empowered Automatic Penetration Testing Tool*. 2023. arXiv: 2308.06782 [cs.SE] (ver p. 26).

[17] B. Dijkstra. *How to perform API testing with REST Assured*. URL: https://techbeacon.com/app-dev-testing/how-perform-api-testing-rest-assured (acedido em 12 de jan. de 2024) (ver p. 15).

[18] H. Ed-douibi, J. Canovas Izquierdo e J. Cabot. «Automatic Generation of Test Cases for REST APIs: A Specification-Based Approach». Em: out. de 2018, pp. 181–190. DOI: 10.1109/EDOC.2018.00031 (ver pp. 19, 21).

[19] H. Ed-douibi, J. Canovas Izquierdo e J. Cabot. «Example-Driven Web API Specification Discovery». Em: jul. de 2017, pp. 267–284. ISBN: 978-3-319-61481-6. DOI: 10.1007/978-3-319-61482-3_16 (ver p. 19).

[20] *Dropbox*. URL: https://www.dropbox.com/ (acedido em 14 de fev. de 2024) (ver p. 3).

[21] *Dropbox API v2: /move*. URL: https://www.dropbox.com/developers/documentation/http/documentation#files-move (acedido em 29 de jan. de 2024) (ver p. 3).

[22] *Eclipse Jersey*. URL: https://eclipse-ee4j.github.io/jersey/ (acedido em 3 de dez. de 2023) (ver p. 7).

[23] *Everything you need to know about code coverage*. URL: https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-coverage/ (acedido em 13 de dez. de 2023) (ver p. 11).

[24] *EvoMaster*. URL: https://github.com/EMResearch/EvoMaster (acedido em 26 de dez. de 2024) (ver p. 18).

[25] R. T. Fielding e R. N. Taylor. «Principled Design of the Modern Web Architecture». Em: *ACM Trans. Internet Technol.* 2.2 (mai. de 2002), pp. 115–150. ISSN: 1533-5399. DOI: 10.1145/514183.514185. URL: https://doi.org/10.1145/514183.514185 (ver p. 6).

[26] R. T. Fielding. «Architectural Styles and the Design of Network-based Software Architectures». Tese de doutoramento. University of California, Irvine, 2000 (ver pp. 1, 6).

[27] *Fuzz Testing: How does fuzz testing work?* URL: https://www.synopsys.com/glossary/what-is-fuzz-testing.html (acedido em 14 de dez. de 2023) (ver p. 12).

[28] S. Galkin et al. *Faker - Clojure Adaptation*. URL: https://github.com/paraseba/faker (acedido em 10 de set. de 2024) (ver p. 63).

[29] D. Graham et al. «Test Types: The Targets of Testing». Em: *Foundations of Software Testing: ISTQB Certification*. 3ª ed. Cengage Learning, 2013. ISBN: 978-1408044056 (ver p. 11).

[30] T. Hamilton. *REST Assured Tutorial for API Automation Testing (Example)*. Dez. de 2023. URL: https://www.guru99.com/rest-assured.html (acedido em 12 de jan. de 2024) (ver p. 15).

[31] T. Hamilton. *State Transition Testing - Diagram & Technique (Example)*. URL: https://www.guru99.com/state-transition-testing.html (acedido em 19 de dez. de 2023) (ver p. 13).

[32] *HTTP request methods*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods (acedido em 16 de nov. de 2023) (ver p. 6).

[33] *HyperSQL DataBase*. URL: http://hsqldb.org/ (acedido em 21 de set. de 2024) (ver p. 95).

[34] *JaCoCo Java Code Coverage Library*. URL: https://www.eclemma.org/jacoco/ (acedido em 13 de dez. de 2023) (ver p. 10).

[35] *Jepsen*. URL: https://jepsen.io/ (acedido em 18 de jan. de 2024) (ver pp. 2, 16).

[36] *JUnit 5*. URL: https://junit.org/junit5/ (acedido em 13 de dez. de 2023) (ver p. 10).

[37] R. N. Kacker, E. S. Lagergren e J. J. Filliben. «Taguchi's Orthogonal Arrays Are Classical Designs of Experiments». Em: *Journal of research of the National Institute of Standards and Technology* 96.5 (1991), pp. 577–591. DOI: doi:10.6028/jres.096.034 (ver p. 12).

[38] S. Karlsson et al. *Exploring API Behaviours Through Generated Examples*. 2023. arXiv: 2308.15210 [cs.SE]. URL: https://arxiv.org/abs/2308.15210 (ver p. 23).

[39] M. Khan. «Different Approaches To Black box Testing Technique For Finding Errors». Em: *International Journal of Software Engineering & Applications* 2.4 (out. de 2011). DOI: 10.5121/ijsea.2011.2404 (ver p. 11).

[40] M. Kim et al. «Automated test generation for REST APIs: no time to rest yet». Em: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '22. ACM, jul. de 2022. DOI: 10.1145/3533767.3534401. URL: http://dx.doi.org/10.1145/3533767.3534401 (ver p. 18).

[41] K. Kingsbury. *Jepsen - Generator*. URL: https://jepsen-io.github.io/jepsen/jepsen.generator.html (acedido em 3 de set. de 2024) (ver pp. 3, 27).

[42] K. Kingsbury e P. Alvaro. *Elle: Inferring Isolation Anomalies from Experimental Observations*. 2020. arXiv: 2003.10554 [cs.DB] (ver p. 17).

[43] *Knossos*. URL: https://github.com/jepsen-io/knossos (acedido em 18 de jan. de 2024) (ver pp. 17, 27).

[44] D. Liarokapis, E. ONeil e P. ONeil. *HISTEX (HISTory EXerciser) : A tool for testing the implementation of Isolation Levels of Relational Database Management Systems*. 2019. arXiv: 1903.00731 [cs.DB] (ver p. 17).

[45] *Limitations of the GET method in HTTP*. URL: https://dropbox.tech/developers/limitations-of-the-get-method-in-http (acedido em 21 de ago. de 2024) (ver p. 43).

[46] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (ver p. i).

[47] H. Maltby et al. *Markov Chains*. URL: https://brilliant.org/wiki/markov-chains/ (acedido em 18 de ago. de 2024) (ver p. 54).

[48] A. Martin-Lopez, S. Segura e A. Ruiz-Cortés. «A Catalogue of Inter-parameter Dependencies in RESTful Web APIs». Em: out. de 2019, pp. 399–414. ISBN: 978-3-030-33701-8. DOI: 10.1007/978-3-030-33702-5_31 (ver p. 21).

[49] A. Martin-Lopez, S. Segura e A. Ruiz-Cortés. «RESTest: Automated Black-Box Testing of RESTful Web APIs». Em: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '21. Association for Computing Machinery, 2021 (ver p. 23).

[50] K. Meinke, F. Niu e M. Sindhu. «Learning-Based Software Testing: A Tutorial». Em: vol. 336. Out. de 2011. ISBN: 978-3-642-34780-1. DOI: 10.1007/978-3-642-34781-8_16 (ver p. 13).

[51] A. G. Mirabella et al. «Deep Learning-Based Prediction of Test Input Validity for RESTful APIs». Em: jun. de 2021. DOI: 10.1109/DeepTest52559.2021.00008 (ver p. 23).

[52] *Mocha*. URL: https://mochajs.org/ (acedido em 24 de jan. de 2024) (ver p. 24).

[53] *OpenAPI Specification*. URL: https://swagger.io/specification/ (acedido em 14 de fev. de 2024) (ver pp. 1, 7).

[54] Pawarsuraj. *What is Postman?* Jul. de 2022. URL: https://medium.com/@pawarsuraj614/what-is-postman-fb9774284ab6 (acedido em 12 de jan. de 2024) (ver p. 15).

[55] *Postman*. URL: https://www.postman.com/ (acedido em 12 de jan. de 2024) (ver p. 14).

[56] pp_pankaj. *State Transition Testing*. URL: https://www.geeksforgeeks.org/state-transition-testing/ (acedido em 19 de dez. de 2023) (ver p. 13).

[57] *REST-assured*. URL: https://rest-assured.io/ (acedido em 12 de jan. de 2024) (ver p. 15).

[58] *RESTEasy*. URL: https://resteasy.dev/index.html (acedido em 3 de dez. de 2023) (ver p. 7).

[59] A. Ribeiro. «Invariant-Driven Automated Testing». Tese de doutoramento. Fev. de 2021. DOI: 10.13140/RG.2.2.31110.65603 (ver p. 33).

[60] T. Ridnik, D. Kredo e I. Friedman. *Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering*. 2024. arXiv: 2401.08500 [cs.LG] (ver p. 26).

[61] H. M. G. Rodrigues, N. Preguica e F. Freitas. *Causal Consistency Verification in Restful Systems*. Dez. de 2022. URL: http://hdl.handle.net/10362/158999 (ver p. 4).

[62] *Run Your First Artillery Test*. URL: https://www.artillery.io/docs/get-started/first-test (acedido em 14 de jan. de 2024) (ver pp. 16, 37).

[63] M. Schäfer et al. *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. 2023. arXiv: 2302.06527 [cs.SE] (ver p. 24).

[64] *Scripting in Postman*. URL: https://learning.postman.com/docs/writing-scripts/intro-to-scripts/ (acedido em 12 de jan. de 2024) (ver p. 15).

[65] *Selenium*. URL: https://www.selenium.dev/ (acedido em 13 de dez. de 2023) (ver p. 10).

[66] S. Simoes et al. *JepREST: Functional tests for distributed REST applications*. 2023. arXiv: 2303.14104 [cs.DC] (ver pp. 4, 27, 32, 33, 35, 37, 49, 55, 66).

[67] Y. Song et al. *RestGPT: Connecting Large Language Models with Real-World RESTful APIs*. 2023. arXiv: 2306.06624 [cs.CL] (ver p. 25).

[68] M. E. de Sousa França et al. *Testing Geo-Replicated Aplications*. Nov. de 2023. URL: http://hdl.handle.net/10362/167025 (ver p. 4).

[69] *Swagger Codegen*. URL: https://swagger.io/tools/swagger-codegen/ (acedido em 4 de dez. de 2023) (ver p. 8).

[70] *Swagger Core*. URL: https://github.com/swagger-api/swagger-core (acedido em 4 de dez. de 2023) (ver p. 8).

[71] *Swagger Editor*. URL: https://swagger.io/tools/swagger-editor/ (acedido em 4 de dez. de 2023) (ver p. 8).

[72] *Swagger Petstore*. URL: https://github.com/swagger-api/swagger-petstore (acedido em 4 de dez. de 2023) (ver p. 8).

[73] H. Tan et al. *PROXYQA: An Alternative Framework for Evaluating Long-Form Text Generation with Large Language Models*. 2024. arXiv: 2401.15042 [cs.CL] (ver p. 26).

[74] *Test your API using the Collection Runner*. URL: https://learning.postman.com/docs/collections/running-collections/intro-to-collection-runs/ (acedido em 12 de jan. de 2024) (ver p. 15).

[75] *The Java EE 6 Tutorial. Building RESTful Web Services with JAX-RS*. URL: https://docs.oracle.com/javaee/6/tutorial/doc/giepu.html (acedido em 4 de dez. de 2023) (ver p. 7).

[76] The Postman Team. *What is OpenAPI?* Ago. de 2023. URL: https://blog.postman.com/what-is-openapi/ (acedido em 3 de dez. de 2023) (ver p. 7).

[77] H. Veldstra. *Using Artillery for Your Functional Testing*. Ago. de 2021. URL: https://www.artillery.io/blog/using-artillery-for-your-functional-testing (acedido em 14 de jan. de 2024) (ver p. 16).

[78] E. Viglianisi, M. Dallago e M. Ceccato. «RESTTESTGEN: Automated Black-Box Testing of RESTful APIs». Em: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 142–152. DOI: 10.1109/ICST46399.2020.00024 (ver pp. 20, 21).

[79] *Visão Geral do IBM JAX-RS*. Fev. de 2023. URL: https://www.ibm.com/docs/pt-br/was/8.5.5?topic=SSEQTP_8.5.5/com.ibm.websphere.nd.multiplatform.doc/ae/cwbs_jaxrs_overview.htm (acedido em 3 de dez. de 2023) (ver p. 7).

[80] *Web Components*. URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_components (acedido em 15 de nov. de 2023) (ver p. 6).

[81] *Welcome to Faker's documentation!* URL: https://faker.readthedocs.io/en/master/ (acedido em 6 de fev. de 2024) (ver pp. 31, 33).

[82] *What is Swagger?* URL: https://swagger.io/tools/open-source/getting-started/ (acedido em 4 de dez. de 2023) (ver p. 8).

[83] *White Box Testing*. URL: https://www.imperva.com/learn/application-security/white-box-testing/ (acedido em 13 de dez. de 2023) (ver p. 10).

[84] M. Zhang, B. Marculescu e A. Arcuri. «Resource and dependency based test case generation for RESTful Web services». Em: *Empirical Software Engineering* 26 (jul. de 2021), p. 76. DOI: 10.1007/s10664-020-09937-1 (ver p. 18).