UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

RAIMUNDO HEITOR SIQUEIRA DE MIRANDA

REST TEST:
TESTES AUTOMATIZADOS PARA PROJETO E EXERCÍCIOS REST

CAMPINA GRANDE - PB

2019

RAIMUNDO HEITOR SIQUEIRA DE MIRANDA

REST TEST:
TESTES AUTOMATIZADOS PARA PROJETO E EXERCÍCIOS REST

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientador: Professor Dr. Dalton Dario Serey Guerrero.

CAMPINA GRANDE - PB

2019

M672r Miranda, Raimundo Heitor Siqueira de.
Rest Test : testes automatizados para projetos e exercícios REST . / Raimundo Heitor Siqueira de Miranda. – 2019.

9 f.

Orientador: Prof. Dr. Dalton Dario Serey Guerrero.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Representational State Transfer. 2. REST. 3. Desenvolvimento de software. 4. Modelo REST. I. Guerrero, Dalton Dario Serey. II. Título.

CDU:004(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

RAIMUNDO HEITOR SIQUEIRA DE MIRANDA

REST TEST:
TESTES AUTOMATIZADOS PARA PROJETO E EXERCÍCIOS REST

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professor Dr. Dalton Dario Serey Guerrero
Orientador – UASC/CEEI/UFCG

Professor Dr. Kyller Costa Gorgônio
Examinador – UASC/CEEI/UFCG

Professor Dr. Tiago Lima Massoni
Examinador – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de novembro 2019.

CAMPINA GRANDE - PB

ResTest: Testes Automatizados Para Projeto e Exercícios REST

Trabalho de Conclusão de Curso

Raimundo Heitor Siqueira de Miranda
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil
raimundo.miranda@ccc.ufcg.edu.br

Resumo

REST é um modelo arquitetural cada vez mais utilizado no desenvolvimento de software. No entanto, seu aprendizado nas universidades é um desafio devido à falta de ferramentas para a verificação e testes de projetos e exercícios REST desenvolvidos pelos alunos. Isto implica em uma lacuna de avaliação em que os professores não podem atuar de forma efetiva e os alunos não têm uma resposta direta sobre suas implementações. Este trabalho visa, portanto, o desenvolvimento de um software que apoie estudantes e professores na avaliação do domínio de conceitos relativos ao modelo REST ao mesmo tempo em que permita ao estudante a avaliação imediata de seus exercícios e projetos.

Palavras-Chave

REST, Test, Automatização.

Repositório

https://github.com/RaimundoHeitorMiranda/restest

1. Introdução

Transferência representacional de estado (REST em inglês) é um modelo de arquitetura para sistemas distribuídos especificado por Roy Fielding em sua tese de doutorado no ano de 2000 [1]. É amplamente utilizado nos mais diversos tipos de serviços na internet, integrando sistemas privados e públicos, internos e externos de forma padronizada e eficiente. Tem como principal ideia a utilização do HTTP como forma de comunicação entre sistemas distribuídos pela Internet [2]. E para tal são definidos diversos princípios que provêm padronização da comunicação, performance e escalabilidade.

Diante desse contexto de utilização do REST torna-se necessário, por parte das universidades e faculdades, um ensino adequado e correto do modelo, partindo desde os princípios do HTTP até a compreensão completa do modelo arquitetural, para a preparação profissional do estudante ao mercado de trabalho. No entanto, a verificação e testes de implementações de sistemas REST é um desafio para os professores.

2. O problema

Devido à grande quantidade de alunos em uma turma se torna inviável por parte do professor o acompanhamento individual das implementações REST do alunos, restando ao professor apenas a verificação de funcionalidades, mas não de como o são implementadas. Essa situação se torna ainda mais complexa quando os alunos têm liberdade de definir os endpoints e o formato dos dados. Esse cenário gera uma lacuna de avaliação que pode induzir o aluno à falsa impressão de que está seguindo o modelo e ao professor de que seus alunos estão aprendendo corretamente.

Este problema é melhor verificado na disciplina de Projeto de Software, do curso de Ciência da Computação da Universidade Federal de Campina Grande (UFCG), na qual os alunos têm contato com o padrão REST pela primeira vez no curso. Nessa disciplina os projetos e exercícios são divididos em dois subprojetos que se comunicam utilizando o padrão, compostos pelo *frontend*, que fornece uma interface visual, e o *backend*, que implementa o servidor.

As avaliações são feitas levando em consideração essencialmente as funcionalidades expostas através do *frontend*, pois não há forma prática do professor acompanhar todas as requisições feitas nos projetos de todos os alunos, o que implica em uma avaliação limitada da implementação do *backend* REST. Por esse motivo é comum encontrar erros graves ao se verificar as requisições HTTP. Dentre os mais comuns estão a utilização de métodos HTTP de forma equivocada, status de respostas incorretos, designs mal elaborados e envio de informações erradas.

Como exemplo temos a figura 1, onde o aluno realiza a consulta sobre o recurso *Campaign* utilizando o corpo da requisição para envio dos parâmetros da consulta. Esta abordagem não segue o padrão estabelecido pelo REST, na qual se utiliza o recurso de *Query Component* para este objetivo [9]. Porém ao se verificar a funcionalidade no *frontend* ela demonstrará funcionar com perfeição. O que explicita a falta de avaliação do *backend* o aprendizado errado pelo aluno.

```
@GetMapping
public ResponseEntity<List<Campaign>>
    getCampaign(@RequestBody @Valid String str, String[] status) {
        return new ResponseEntity<List<Campaign>>
        (campaignService.searchCampaign(str, status), HttpStatus.OK);
}
```

Figura 1: Mostra uma requisição incorreta, para o modelo REST, implementada por um aluno.

Existem softwares que realizam requisições HTTP que são utilizados para testes, porém tais softwares possuem limitações que os inviabiliza para testes de projetos de disciplinas. A principal limitação é a ausência de um teste automático, isto é, as respostas das requisições tem que ser conferidas individualmente e visualmente, o que torna impraticável devido à grande quantidade de requisições de projetos que uma disciplina pode ter.

Outras limitações destes softwares é que não há como o professor acompanhar o domínio dos alunos de forma rápida e não há uma resposta automática indicando os erros cometidos pelos alunos. Com isso, nota-se a necessidade de um software que automatize o processo de testes de exercícios e projetos, que permita ao professor um acompanhamento formal do domínio dos alunos e que forneça aos alunos resultados de avaliação sobre suas implementações.

3. Solução

3.1 Descrição

Neste trabalho apresentamos o *ResTest*. O *ResTest* é um software de linha de comando que tem como principal objetivo o teste automatizado de projetos e exercícios REST em um ambiente acadêmico. Através dele é possível verificar se o aluno está seguindo corretamente todos os conceitos e padrões relativos ao modelo REST.

O *ResTest* atende às necessidades dos alunos e dos professores de forma eficiente. Para os alunos é fornecido um rápido resultado sobre sua implementação, contendo erros cometidos e o resultado esperado. Para os professores, torna possível acompanhar o domínio dos alunos de uma turma de forma rápida e, com isso, tomar medidas necessárias em relação aos assuntos mais deficitários da turma, além de seu uso para a avaliação individual dos alunos.

Para atender às necessidades de testes automáticos foram definidos três requisitos principais:

- Executar os testes e obter resultados automaticamente de forma rápida.
- Os testes devem ser fornecidos pelos professores de acordo com os exercícios propostos e especificações de projetos e devem poder ser escritos pelos próprios estudantes.
- Os resultados dos testes devem expor de forma clara os erros e acertos cometidos pelos alunos

Optamos por implementar o software integrando-o com o TST [4], um sistema para controle de testes já utilizado em algumas disciplinas do curso de ciência da computação da UFCG, incluindo a disciplinas de Projeto de Software para a parte de aprendizado de JavaScript. Ele torna possível a escrita, organização e execução de testes para vários alunos simultaneamente.

3.2 Lógica da aplicação

A lógica de utilização do sistema começa com a criação de testes por parte do professor de acordo com a especificação de um exercício ou projeto. Para tal, um teste (Listagem 1) deve sempre conter uma requisição (Listagem 1, linhas 2 a 7) e um resultado (Listagem 1, linhas 8 a 18). A requisição é formada por uma *path*, que representa o endereço da API REST a ser testada, o método HTTP da chamada, podendo ser: *GET*, *POST*, *PUT* , *DELETE*; Por fim pode ser definido um cabeçalho e um conteúdo a ser enviado pela requisição sendo esses dois de acordo com a necessidade da requisição.

Para o resultado, o professor deve definir o status HTTP da resposta e o conteúdo esperado. Todos os testes devem ser salvos em um arquivo no formato JSON [3] e passados como parâmetro para o programa. Esta medida é útil pois permite aos professores criarem diferentes níveis de testes para diferentes avaliações.

```
1.  {
2.     "requestTest": {
3.     "method": "GET",
4.     "path": "/api/usuarios/200",
5.     "headers": null,
6.     "body": null
7.     },
8.     "responseTest": {
9.       "httpStatus":"200",
10.      "body":{
11.        "id": 200,
12.        "email": "restTest2@gmail.com",
13.        "firstName": "REsTest",
14.        "lastName": "TESTE",
15.        "cardNumber": 2023423420,
16.        "password": "123456"
17.        }
18.    }
19.  }
```

Listagem 1: Formato geral de um Teste. (linhas 2 a 7) A requisição, que no caso é um *GET* para o *path* 'api/usuários' sem cabeçalho e sem conteúdo. (linhas 8 a 18) A resposta onde é possível ver o status HTTP e o conteúdo esperado.

Uma vez criados os testes, os alunos deverão configurar o software. Para isso deve ser editado o arquivo *config.json* (Listagem 2) onde deve ser definir o endereço do servidor onde o projeto ou exercício está hospedado, a porta, os cabeçalhos padrões que serão enviados em todas as requisições e por fim as configurações de segurança onde é possível definir o *path* e as credenciais necessárias para a obtenção de uma chave de autorização ou definir a chave diretamente.

```
1.  {"requestConfig":{
2.          "server":"http://localhost",
3.          "port":"3000",
4.          "headers": {
```

```
5.              "headerKey":"headerValue"} },
6.      "securityConfig":{
7.          "pathLogin":"/login",
8.          "login": {
9.              "user":"admin",
10.             "password":"admin"},
11.         "token":"",
12.         "headerKey":"authorization"
13.     }
14. }
```

Listagem 2: Arquivo de configuração.

Com os testes criados e o arquivo config.json devidamente configurado basta executar o comando 'restest' para inicializar o programa. O software, então, irá realizar as requisições HTTP especificadas no arquivos de testes para o servidor definido pelo aluno no arquivo *config.json*. A cada requisição o software irá esperar a resposta e então verificar se a resposta confere com a especificada pelo professor.

A validação se dá pela comparação do status e do conteúdo da resposta. Caso o status seja diferente é informado o status esperado (Listagem 3, linhas 11 e 27). No caso do contéudo, é executado um algoritmo de de verificação de JSON, onde é retornado: os atributos em excesso, os atributos em falta, os valores errados e respectivos valores corretos (Listagem 3, linhas 5, 6, 8 respectivamente).

Por fim, ao final da validação de todas as requisições é informado o resultado geral, tendo na primeira linha um resumo para compatibilização com o TST (Listagem 3, linha 1) e nas demais linhas os resultados detalhados de cada teste no formato JSON.

```
1.  FFFFFFF.F
2.  [
3.    Result {
4.      valid: false,
5.      attributeExtra: ['email'],
6.      attributeMissing: [ 'id' ],
7.      valuesDiff:
8.       [ 'Attribute Value Diff: .CREDICARD
    have different values:<11111> is
    different of <22222>(expected)' ],
9.      request_verb_path: 'POST in
    /v1/api/users',
10.     status_ok: false,
11.     status_diff: 'Expected 201 but was
    200' },
12.   Result {
13.     valid: true,
14.     attributeExtra: [],
15.     attributeMissing: [],
16.     valuesDiff: [],
17.     request_verb_path: 'GET in
    /v1/api/users/200',
18.     status_ok: true,
19.     body_ok: true },
20.   Result {
21.     valid: false,
22.     attributeExtra: [],
23.     attributeMissing: [],
24.     valuesDiff: [],
25.     request_verb_path: 'GET in
    /v1/api/users',
26.     status_ok: false,
27.     status_diff: 'Expected 200 but was
    400' }...
28.  ]
```

Listagem 3: Mostra o resultado final da execução dos testes. Linha 1 mostra o resultado resumido para compatibilização com o TST; Linha 8 mostra resultado mostra valores diferentes do esperado; Linhas 12 a 19 mostra um resultado válido.

3.3 Tecnologias utilizadas

Devido à natureza da solução, onde sempre serão processados dados no formato JSON, foi escolhida a linguagem *JavaScript* por sua facilidade em manipular tais dados pois, com essa linguagem é simples listar, adicionar ou remover atributos além de realizar comparações. Tais operações seriam demasiadamente trabalhosas de se lidar em outras linguagem de programação, portanto *JavaScript* foi a escolha ideal para o desenvolvimento da solução.

Contudo, *JavaScript* possui limitações que impactam na dificuldade do desenvolvimento, como definição de interfaces e sincronização das operações. A fim de contornar esses problemas foi escolhido o super conjunto *TypeScript* para a codificação [5], tal escolha permitiu a criação de interfaces e realização de operações síncronas de forma simples.

Para a realização de requisições foi escolhida a biblioteca *Request* que facilita o trabalho de configuração da requisição e obtenção da resposta [6]. Como ferramentas externas foram utilizados o software de gerência de rede *WireShark* para a visualização das requisições em rede [7] e *JsonServer* para a criação de servidores para testar o software [8].

3.4 Integração com o TST

O TST é uma plataforma de controle e organização de testes utilizada no curso de computação da UFCG, através dela é possível executar e obter resultados do *ResTest* sobre os projetos e exercícios dos alunos de forma mais simples. Para atender a compatibilização foi necessário ter como primeira informação de retorno de execução um conjunto de caracteres que significam o resultado do testes, sendo o '.' sucesso e o 'F' falha. Também é especificado o arquivo de configuração do TST, denominado *tst.yaml* e pode ser visto abaixo:

```
1.  extensions: [js]
2.
3.  tests:
4.  - type: script
5.    script: node
    ./node_modules/.bin/restest requests.json
```

Listagem 4: tst.yaml mostra a configuração para se executar o ResTest dentro do TST.

3.5 Arquitetura e funcionamento

Com objetivo de facilitar o desenvolvimento, manutenção e evolução do software, a arquitetura da solução é dividida em

módulos, sendo eles o *ResTest*, *Models*, *FileReader*, *HttpTest* e *Comparator*. A visão geral e sua relações podem ser vistas na imagem abaixo:

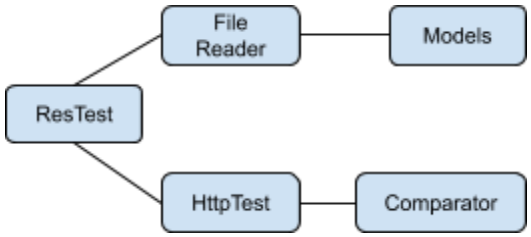

Figura 2: O módulo *ResTest* utiliza o *FileReader* (que utiliza o *Models*) e o *HttpTest* (que utiliza o *Comparator*).

O primeiro módulo é o *ResTest*. Ele é o responsável por iniciar o software, coordenar os demais módulos, garantir a ordem de execução dos testes e por retornar o resultado final.

O módulo *Models* reúne classes e interfaces. Essas classes e interfaces atendem a dois objetivos: o primeiro é o mapeamento dos dados contidos nos arquivos de testes e de configuração e o segundo é padronizá-los, garantindo assim que os objetos terão os mesmos conjuntos de atributos.

O módulo *FileReader* é o responsável pela leitura dos arquivos de testes e configuração, nele são realizados os mapeamentos do arquivos JSON para objetos da linguagem *JavaScript*, utilizando as classes e interfaces definidas no módulo *Models*. Durante o processo de mapeamento é realizado uma verificação sobre todos os dados contidos nos arquivos com objetivo de detectar a ausência de dados ou dados incorretos. Caso haja algum erro nos dados o erro é informado ao usuário e o programa é encerrado.

O módulo *HttpTest* é o responsável por realizar as requisições e delegar os resultados para a validação. Para isso esse módulo utiliza as informações do arquivo de configuração para poder montar as requisições, isto é, é neste módulo onde são adicionados o endereço da requisição, porta, cabeçalho e por fim, a chave de autorização.

Por último, o módulo *Comparator* é o responsável por validar as respostas das requisições em relação às respostas definidas nos arquivos de testes. É neste módulo que se encontra o algoritmo capaz de identificar um conjunto de erros cometidos pelo aluno, sendo eles:

- Erro no status HTTP de resposta.
- Erros de atributos extras, ou seja, o resultado contém mais atributo do que o teste esperava.
- Erros de atributos ausentes, ou seja, o resultado não contém todos os atributos que o teste esperava.
- Erros nos valores dos atributos, ou seja, os dados de um atributo possui valor diferente do que o esperado pelo teste.
- Erros de tipagem, ou seja, o tipo de um dado é diferente do que o esperado pelo teste.

O funcionamento do software segue uma ordem sequencial, primeiro o módulo *ResTest* invoca o módulo *FileReader* que processa os arquivos e, ao fim, retorna os objetos de configuração e de testes. Logo depois, o módulo *HttpTest* é chamado para cada teste, executa a requisição e manda o resultado para o *Comparator*, que ao finalizar a verificação retorna um objeto de resultado. O objeto de resultado é alimentado com informações sobre a requisição no *HttpTest* que o envia como resposta para o módulo principal. Ao término de todos os testes o módulo principal então exibe ao usuário o resultado de todos os testes em ordem de execução (Listagem 3).

4. Uso em projetos acadêmicos

Com o desenvolvimento finalizado foi escolhido cinco projetos da disciplinas de Projeto de Software do período 2019.2 para utilização do *ResTest*. Seguindo a lógica descrita na seção 3.2 foi criado o arquivo de testes que manipula o recurso ‘campanha’, definido na especificação do projeto, com objetivo de verificar se os princípios do REST estavam sendo seguidos.

4.1 Testes

Os testes consistem em consulta, adição, atualização e remoção do recurso ‘campanha’ do projeto. Estas operações foram definidas em uma sequência lógica a fim de se testar o maior número de casos possíveis. As operações e seus objetivos consistem em:

1. Consulta sobre todos as campanhas do sistema para validar o estado inicial e consulta genérica.
2. Adição de uma nova campanha para validar a operação de adição.
3. Busca a nova campanha adicionada para validar a operação de busca de um recurso específico.
4. Consulta sobre todas as campanhas, para validar a mudança do estado.
5. Modificação da campanha adicionada para validar a operação de modificação.
6. Consulta da campanha adicionada para validar as mudanças ocorridas.
7. Remoção da campanha para validar a operação de remoção.
8. Consulta da campanha removida para validar a busca de recurso inexistente.
9. Consulta sobre todos as campanhas do sistema para verificar se o estado final se manteve inalterado após a adição, modificação e remoção do mesmo recurso.

4.2 Resultado

Ao todo 45 testes no total foram executados, dos quais apenas 12 tiveram êxito (26,66%) e 33 apresentaram algum tipo de falha. Dos testes que obtiveram êxito: 5 foram na realização de consulta sobre um recurso inexistente, 4 no estado inicial, 2 sobre consultas e 1 no estado final. Os demais testes apresentaram falhas diversas e são melhor observados no gráfico abaixo:

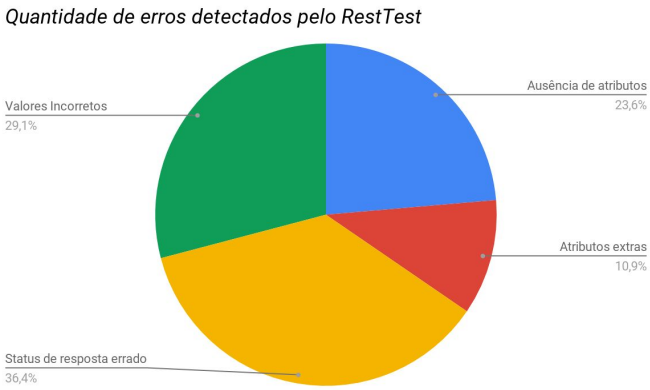

Gráfico 1: Mostra os erros cometidos pelos alunos nas implementações dos projetos. Em verde os valores incorretos, em azul a ausência de atributos, em vermelho os atributos extras, em amarelo os status de resposta incorreto.

Dentre os erros apontados no gráfico observa-se uma grande quantidade de status de resposta errado, atributos ausentes e valores de atributos incorretos. Ao observar o resultado de um dos testes de forma mais detalhada (Listagem 5) podemos notar que o projeto não retornou um atributos, denominado 'comentários' (Listagem 5, linha 3), esperados na consulta do recurso, assim como a resposta possui atributos com valores diferentes do que o esperado (Listagem 5, linha 5 e 6).

```
1.  {   valid: false,
2.      attributeExtra: [],
3.      attributeMissing: [ 'comentarios' ],
4.      valuesDiff:
5.       [ 'Attribute Value Diff:
    .[0].DEADLINE have different
    values:<2019-10-31> is different of
    <2019-11-01>(expected)',
6.         'Attribute Value Diff: .[0].URL
    have different
    values:<comprar-cadeira-de-rodas> is
    different of
    <comprando-cadeira-de-rodas>(expected)'
    ],
7.      request_verb_path: 'GET in
    /campaigns',
8.      status_ok: true }
9.
```

Listagem 5: Resultado de um teste indicando ausência de atributos e valores incorretos.

4.3 Conclusão

Os resultados demonstram que o RestTest torna possível que tanto o professor como o estudante possam ver com clareza os erros cometidos no desenvolvimento das APIs de aplicações baseadas no modelo arquitetural REST. Torna ainda possível que o professor direcione suas aulas para os assuntos mais deficitários, como, por exemplo, a importância da escolha apropriada dos status de resposta para cada tipo de requisição. Pela quantidade de atributos ausentes nota-se também um indício de que os alunos não estão seguindo a especificação do projeto, o que pode levar o professor a tomar medidas para o cumprimento da especificação. Quanto ao aluno, ao ver a figura 4 , saberá quais erros está cometendo de forma clara, concisa e didática auxiliando na autoavaliação e no aprendizado.

5. Experiência

5.1 Processo de desenvolvimento

O processo de desenvolvimento seguiu uma metodologia ágil, onde o software foi desenvolvido buscando atender um objetivo por vez. O primeiro objetivo foi a leitura de arquivos, em seguida foram definidos as classes e interfaces e, por fim, desenvolvido os mapeamentos. Terminado esse objetivo o código foi dividido em dois módulos: o *FileReader* e o *Models*.

Terminada a etapa anterior o próximo objetivo atendido foi a funcionalidade de requisições que, quando terminada, foi exportada para um módulo específico, o *HttpTest*. Por fim, o último objetivo foi o algoritmo de comparação de objetos JSON que ficou em um módulo próprio denominado *Comparator*. O módulo *ResTest* foi implementado em paralelo com os outros módulos, pois este módulo é o responsável pelo controle dos demais.

5.2 Desafios e soluções

Durante o processo de desenvolvimento três problema se tornaram um grande desafio. O primeiro deles foi a comparação de objetos JSON, visto que dentro de um objeto poderia haver outros objetos e listas, que por sua vez, poderiam ter outros objetos e assim indefinidamente. Para resolver este problema foi necessário a concepção de um algoritmo recursivo e inteligente, recursivo, pois, se um dos atributos fosse um segundo objeto o algoritmo teria que ser executado recursivamente para aquele objeto, e inteligente, pois, se fosse uma lista ou um tipo de dado primitivo outro tipo de verificação teria que ser utilizada.

O segundo grande problema se deu pela natureza assíncrona da linguagem *JavaScript*, uma vez que, a lógica do projeto é totalmente síncrona, ou seja, é necessário seguir uma ordem bem definida de operações, por exemplo, a comparação precisa da resposta da requisição, que por sua vez precisa do teste já completamente mapeado. Também é necessário uma ordem na execução dos testes, visto que, um teste de consulta pode depender de um teste de adição anterior. Para a solução desse problema foi utilizado recursos oferecidos pela própria linguagem: *promises*, *async* e *await* para garantir a ordem correta de execução da lógica do projeto.

Por fim, o último desafio foi o tratamento de identificadores sobre respostas das requisições. Quando se adiciona um recurso através de uma *API REST* é retornado um identificador para o recurso adicionado, através deste identificador é possível manipular o recurso. O problema se dá pelo fato desses identificadores serem aleatórios e, por isso, não ser possível escrever testes sem saber quais foram os identificadores utilizados.

Para resolver esse problema foi definido que todo teste de adição de um recurso tenha um identificador em sua resposta, este identificador pode ser utilizado nos demais testes. O software ao realizar a adição do recurso vai receber um identificador real do

servidor e o mapear ao identificador definido no teste, com isso, toda vez que o identificador do teste aparecer em alguma requisição será trocado pelo identificador real, garantindo a realização da requisição com o identificador correto e de forma transparente para quem escreve o teste.

5.3 Limitações

O *ResTest* é limitado a funcionar apenas com dados no formato JSON, não sendo possível testar requisições que envolvam dados de outro tipo como XML e arquivos como fotos, imagens e vídeos. Dentre os métodos HTTP o *ResTest* suporta apenas o GET, POST, PUT e DELETE.

5.4 Trabalhos futuros

Como forma de melhorar o propósito do *ResTest* um conjunto de funcionalidades podem ser implementadas, dentre elas:

- Uma ferramenta que auxilie na criação de testes.
- Uma interface gráfica do próprio *ResTest* para expor de forma mais didática o resultados dos testes.
- A utilização de máquinas virtuais para a execução local do projetos dos alunos e, com isso, executar testes sem restrições, completos e complexos.
- A possibilidade de configurar quais validações devem ser feitas e quais erros podem ser ignorados.
- Uma ferramenta que exporta os resultados dos *ResTest* de forma estatśitica, melhorando a visualização dos erros.

6. Referências

[1] UCI ICS. Fielding Dissertation: CHAPTER 5: Representational State Transfer. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm>. Acesso em 30 de junho de 2019.

[2] Hypertext Transfer Protocol specification(HTTP). Disponível em:<https://www.w3.org/Protocols/rfc2616/rfc2616.html>. Acesso em 20 de Agosto de 2019.

[3] The JavaScript Object Notation (JSON) Data Interchange Format. Disponível em: <https://tools.ietf.org/html/rfc7159>.

Acesso em 30 de Agosto de 2019.

[4] TST. Disponível em <https://github.com/daltonserey/tst>, Acesso em 30 de Agosto de 2019.

[5] TypeScript. Disponível em: <https://www.typescriptlang.org/docs/home.html>. Acesso em 10 de outubro de 2019.

[6] Request. Disponível em:< https://www.npmjs.com/package/request>. Acesso em 10 de outubro de 2019.

[7] Wireshark. Disponível em: <https://www.wireshark.org/docs/>. Acesso em 10 de outubro de 2019.

[8] JsonServer. Disponível em: <https://github.com/typicode/json-server>. Acesso em 10 de outubro de 2019.

[9] Query Params. Disponível em: <https://tools.ietf.org/html/rfc3986#section-3.4>. Acesso em 15 de novembro de 2019.