

Documentação | Desafio de automação de Testes de API



Controle de versões do documento				
Versão	Data	Autor	Revisor	Notas da Revisão
1.0	13/01/24	Vanessa Pinheiro		Documentação do Desafio de Testes API Confluence

Introdução

Este documento apresenta a documentação referente aos testes implementados para o projeto, com base nos requisitos especificados no documento [Desafio de Automação de Testes de API](#). Ele inclui:

- Informações dos testes
- Instruções sobre como rodar os testes
- Testes implementado e explicação dos casos cobertos

Os cenários a seguir incluem testes organizados conforme os princípios estabelecidos pelo ISTQB (International Software Testing Qualifications Board). O objetivo é assegurar que a qualidade e a cobertura dos testes atendam plenamente aos requisitos estabelecidos.

Informações dos testes

1. No projeto, foi utilizado o **Postman** para configurar variáveis de **pre-request** e **post-response**, permitindo a personalização e reutilização de dados entre as requisições. As requisições foram organizadas em **arquivos JSON**.
2. A execução foi automatizada com **Newman**, que permitiu rodar múltiplas iterações dos testes, além de gerar relatórios em **JSON** e **HTML**.
3. A aplicação **Serverest** foi executada em um ambiente **Docker**, garantindo consistência e portabilidade.
4. O projeto foi gerenciado com **npm** e **Node.js**, incluindo dependências do **Newman** e scripts para automação.
5. O controle de versão foi feito com **Git** e **GitHub**, utilizando **GitHub Actions** para integração contínua.
6. A edição do código foi feita no **VSCode**, em um ambiente **Linux**.
7. A documentação foi organizada no **Confluence**.
8. A metodologia **BDD** foi aplicada utilizando **Cucumber**.

No **Postman**, foram utilizados diversos recursos para otimizar e automatizar os testes de API, descritos da seguinte forma:

1. **Variáveis:**
 - Foram configuradas variáveis globais e de ambiente, como `email`, `senha`, e `nomeDoUsuario`.
2. **Pre-request Script:**
 - Scripts foram implementados na seção de Pre-request para atualizar dinamicamente valores das requisições antes de sua execução.

3. Tests:

- No campo de **Tests**, foram definidos scripts que validaram as respostas das requisições. Esses testes verificaram se os dados retornados atendiam aos critérios esperados, como status code, estrutura do JSON, e valores específicos.

4. Arquivo JSON para Exportação:

- As coleções de requisições criadas foram exportadas como arquivos JSON. Isso permitiu a integração com o **Newman**, ferramenta de linha de comando do Postman, para execução em pipelines de CI/CD e automação de testes em massa.

O **Newman** foi utilizado para executar os testes no terminal, a partir do arquivo exportado pelo Postman. Isso permitiu a execução de múltiplas iterações, viabilizando a automação e a execução em **pipelines CI/CD**, garantindo a execução contínua e automatizada dos testes de API. **Testes implementados**

Docker para executar o projeto serverlest localmente sem depender do servidor externo

Instruções sobre como rodar os testes

Como configurar o ambiente?

Informações de configuração esta no [README](#)

i Executar os Testes Localmente [↗](#)

Para gerar os relatórios localmente, execute o seguinte comando:

```
1 npm run newman
```

Resultado: será gerado no console.

```
1 npm run newman:reports
```

Arquivos gerados: `./reports/report.json` e `./reports/report.html`.

Executar os Testes com Docker [↗](#)

Para executar os testes com Docker, utilize os seguintes comandos:

Testes em Múltiplas Interações [↗](#)

```
1 npm run newman:docker
```

Resultado: será gerado no console.

```
1 npm run newman:docker:report
```

Arquivos gerados: `./reports/report.json` e `./reports/report.html`.

Testes com 120 Iterações [↗](#)

```
1 npm run newman:docker:multiple
```

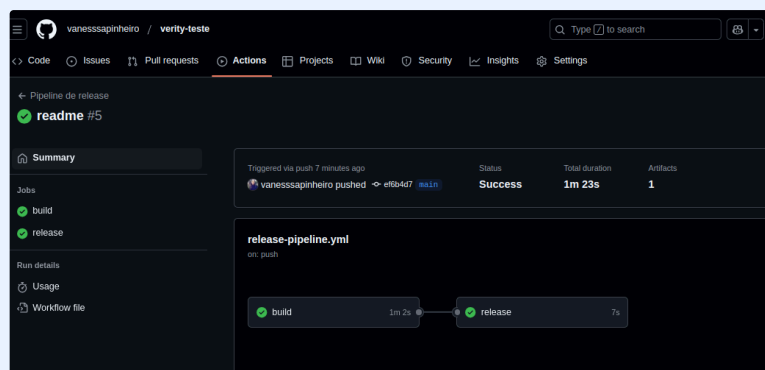
Resultado: será executado no Docker e gerado no console com 120 iterações.

```
1 npm run newman:docker:multiple:report
```

Resultado: será executado no Docker e gerado no console com 120 iterações, além de gerar os arquivos `./reports/report.json` e `./reports/report.html`.

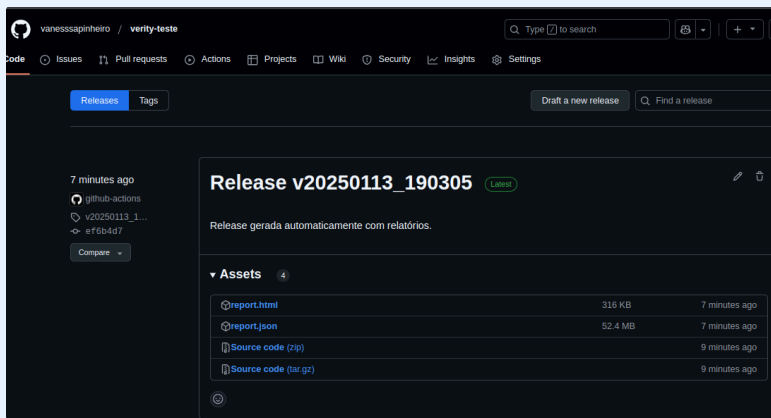
i Instruções da automação com GitHub Actions [↗](#)

- i** 1. Foi gerada uma pipeline no **GitHub Actions** que é executada quando ocorre uma alteração na branch main.



- Os testes são executados utilizando docker + newman com 120 iterações
- No término é gerada uma release com a geração dos arquivos de relatórios nos formatos JSON e HTML com resultado dos testes.

Arquivos ficam disponíveis para downloads, segue link da [Release](#).



Testes implementados [↗](#)

Informações Iniciais

Foram criados testes iniciais de cadastro de usuário, considerando que o **Newman** executa os testes de forma sequencial. A primeira requisição é responsável pelo cadastro do usuário, utilizando informações válidas e aleatórias. Os dados desse usuário são então empregados nas requisições subsequentes, garantindo a continuidade e a consistência dos testes.

Headers: [↗](#)

Content-Type: application/json

Accept: application/jsonheader

Authorization: Bearer Bearer

Exemplo de Corpo da Requisição: [↗](#)

```
1 {
2   "nome": "{{currentName}}",
3   "email": "{{currentEmail}}",
4   "password": "{{currentPassword}}",
5   "administrador": "{{currentAdministrador}}"
6 }
```

Cenários [↗](#)

Login [↗](#)

Testes de Caixa Preta | Funcionais - POST /users

1. Validar login do usuário.

- Objetivo: Validar se ao acessar com dados válidos login é realizado com sucesso.
- Passos: Enviar uma requisição POST para o endpoint `/login` com credenciais válidas.
- Dados de entrada:

```
1 {
2   "email": "usuario@exemplo.com",
```

```
3  "password": "senha123"
4  }
```

a. Resultado esperado:

```
1  {
2    "message": "Login realizado com sucesso",
3    "authorization": "Bearer <token>"
4  }
```

PASS Status code é 200 [🔗](#)

PASS Mensagem é correta [🔗](#)

PASS Token de autorização está presente [🔗](#)

2. Validar E-mail Inválido ou Senha Incorreta:

- Objetivo: Validar se ao informar dados invalidos é retornado corretamente para o usuario.
- Passos: Enviar credenciais inválidas.
- Dados de entrada:

```
1  {
2    "email": "email_invalido@exemplo.com",
3    "password": "senha_errada"
4  }
```

a. Resultado esperado:

```
1  {
2    "message": "Email e/ou senha inválidos"
3  }
```

PASS Status code é 401 [🔗](#)

PASS Mensagem de erro está correta [🔗](#)

3. Validação de Campos | Campos Obrigatórios Ausentes: [🔗](#)

- Obejtivo: Validar se ao realizar o login sem passar os campos obrigatorios, e informado usuario corretamente.
- Passos: Não enviar campos obrigatorios

1. email .
2. password .
3. Ambos os campos ausentes.

- Resultado esperado:

```
1  {
2    "email": "email é obrigatório",
3    "password": "password é obrigatório"
4  }
```

PASS Status code é 400 [↗](#)

PASS Status code é 400 para falta de ... [↗](#)

PASS Menssagem de erro por falta de password correta [↗](#)

▶ Usuários | Lista [↗](#)

Testes de Caixa Preta | Funcionais [↗](#)

1. Listar usuários cadastrados, sem filtro

- Objetivo: Validar se esta retornando corretamente quantidade com todos os usuarios cadastrados sem opção de filtro.
- Passos: Enviar requisição GET sem filtros.
- Resultado esperado:

```
1 {
2   "quantidade": 1,
3   "usuarios": [
4     {
5       "nome": "João Silva",
6       "email": "joao@exemplo.com",
7       "password": "senha123",
8       "administrador": "true",
9       "_id": "<generated_id>"
10    }
11  ]
12 }
```

PASS Status code é 400 [↗](#)

PASS quantidade é um número [↗](#)

PASS Usuarios é um array [↗](#)

PASS usuarios contém itens válidos [↗](#)

2. Listar usuários cadastrados, com filtro [↗](#)

- Objetivo: Validar se esta retornando corretamente todos os usuarios com as opções de filtro.
- Passos: Testar com filtros válidos `_id`, `nome`, `email`, e `administrador`.
- Resultado esperado:

- ```
1 {
2 "quantidade": 1,
3 "usuarios": [
4 {
5 "nome": "João Silva",
6 "email": "joao@exemplo.com",
7 "password": "senha123",
8 "administrador": "true",
9 "_id": "<generated_id>"
10 }
11]
12 }
```

**PASS** Status code é 400 [↗](#)

**PASS** quantidade é um número [↗](#)

**PASS** Usuarios é um array [↗](#)

**PASS** usuarios contém um item [↗](#)

**PASS** Query params são iguais ao primeiro usuário na resposta [↗](#)

### 3. Consultar usuários passando filtros inválidos [↗](#)

- Objetivo: Validar se esta retornando corretamente todos os usuarios com as opções de filtro.
- Passos: Testar com filtros inválidos para `_id`, `nome`, `email`, e `administrador`.
- Resultado esperado:

```
1 {
2 "email": "email deve ser um email válido",
3 "administrador": "administrador deve ser 'true' ou 'false'"
4 }
```

**PASS** Status code é 400 [↗](#)

**PASS** Email invalido [↗](#)

**PASS** Administrador invalido [↗](#)

### 4. Verificar usuarios não encontrados [↗](#)

- Objetivo: Validar se esta retornando corretamente todos os usuarios com as opções de filtro.
- Passos: Testar com filtros válidos e inválidos para `_id`, `nome`, `email`, e `administrador`.
- Resultado esperado:

```
1 {
2 "quantidade": 0,
3 "usuarios": []
4 }
```

**PASS** Status code é 400 [↗](#)

**PASS** quantidade é igual 0 [↗](#)

**PASS** usuarios é um array vazio [↗](#)

## ▶ Usuários | Cadastro [↗](#)

Testes de Caixa Preta | Funcionais [↗](#)

### 1. Validar cadastro de usuário [↗](#)

- Objetivo: Validar se cadastro de usuários é realizado com sucesso.
- Passos: Enviar Requisição POST com dados válidos:

```
1 {
2 "nome": "João Silva",
3 "email": "joao@exemplo.com",
4 "password": "senha123",
5 "administrador": "true"
6 }
```

```

7 // Exemplo utilizando variavel
8 {
9 "nome": "{{randomFullName}}",
10 "email": "{{timestamp}}_{{randomEmail}}",
11 "password": "{{randomPassword}}",
12 "administrador": "{{randomBoolean}}"
13 }

```

- Resultado esperado:

```

1 {
2 "message": "Cadastro realizado com sucesso",
3 "_id": "<generated_id>"
4 }

```

**PASS** Status code é 201 [↗](#)

**PASS** Mensagem é correta [↗](#)

**PASS** \_id presente [↗](#)

## 2. Validar cadastrar com e-mail já utilizado

- Objetivo: Validar ao cadastrar novo usuario, utilizando email já existente se é notificado usuario
- Passos: Enviar requisição com um email já registrado.
- Resultado esperado:

```

1 {
2 "message": "Este email já está sendo usado"
3 }

```

**PASS** Status code é 400 [↗](#)

**PASS** Mensagem é correta [↗](#)

## 3. Testes de Validação de Campos [↗](#)

- Objetivo: Validar se ao tentar cadastrar usuario sem passar dados nos campos, é retornado ao usuario o campo e informando que é obrigatorio.
- Passos: Testar omissões dos campos `nome`, `email`, `password` e `administrador`.
- Resultado esperado:

```

1 {
2 "nome": "nome é obrigatório",
3 "email": "email é obrigatório",
4 "password": "password é obrigatório",
5 "administrador": "administrador é obrigatório"
6 }

```



**PASS** Status code é 400 [↗](#)

**PASS** nome é obrigatorio [↗](#)

**PASS** email é obrigatorio [↗](#)

**PASS** password é obrigatorio [↗](#)

**PASS** administrador é obrigatorio [↗](#)

#### 4. Propriedade email invalido

Objetivo: Validar se ao tentar cadastrar usuario com email invalido, é retornado ao usuario que email esta invalido.

Passos: Testar envio campo `email` invalido.

1. Resultado esperado:

```
1 {
2 "email": "email deve ser um email válido"
3 }
```

**PASS** Status code é 400 [↗](#)

**PASS** email invalido [↗](#)

#### 5. Propriedade administrador invalido

Objetivo: Validar se ao tentar cadastrar usuario com campo administrador invalido é retornado ao usuario informando que o administrador deve ser verdadeiro ou falso.

Passos: Testar envio campo `administrador` invalido.

1. Resultado esperado:

```
1 {
2 "administrador": "administrador deve ser 'true' ou 'false'"
3 }
```

**PASS** Status code é 400 [↗](#)

**PASS** adminstrador invalido [↗](#)



### Testes de Caixa Preta | Funcionais [↗](#)

#### 1. Buscar por ID GET

- Objetivo: Validar se buscando o usuarios cadastrados por ID.
- Passos: Enviar requisição GET com ID valido
- Resultado esperado:

```
1 {
2 "nome": "teste001",
3 "email": "teste001@teste.com",
4 "password": "teste",
5 "administrador": "true",
6 "_id": "hZhIFvf5cpySGGT0"
7 }
```

**PASS** Status code é 200 [↗](#)

**PASS** propriedades validas [↗](#)

**PASS** \_id da url deve ser igual \_id da resposta [↗](#)

## 2. Buscar por ID GET

- Objetivo: Validar se buscando o usuarios com ID invalido, deve informar que usuario não foi encontrado.
- Passos: Enviar requisição GET com ID invalido
- Resultado esperado:

```
1 {
2 "message": "Usuário não encontrado"
3 }
```

**PASS** Status code é 200 [↗](#)

**PASS** Mensagem usuário não encontrado [↗](#)

## 3. Editar por ID PUT

- Objetivo: Editar usuarios cadastrados por ID.
- Passos: Enviar requisição PUT editando informações do usuario
- Observação: no Postman em **pre-request script** criar auto encremento para editar em massa.
- Resultado esperado:

```
1 {
2 "message": "Registro alterado com sucesso"
3 }
```

**PASS** Status code é 200 [↗](#)

**PASS** propriedades validas [↗](#)

**PASS** Mensagem é correta [↗](#)

## 4. Editar por ID, Parametros invalidos PUT

- Objetivo: Validar editar usuarios cadastrados por id com email, administrador invalidos
- Passos: Enviar requisição PUT editando informações do usuario, com email e administrador invalido
- Resultado esperado:

```
1 {
2 "email": "email deve ser um email válido",
3 "administrador": "administrador deve ser 'true' ou 'false'"
4 }
```

**PASS** Status code é 200 [↗](#)

**PASS** Email invalido [↗](#)

**PASS** Administrador invalido [↗](#)

## 5. Editar por ID, Parametros obrigatorios PUT

- Objetivo: Validar editar usuarios cadastrados não passando parametros, deve retornar informando campo obrigatorio.
- Passos: Enviar requisição PUT sem parametros.

- Resultado esperado:

```
1 {
2 "nome": "nome é obrigatório",
3 "email": "email é obrigatório",
4 "password": "password é obrigatório",
5 "administrador": "administrador é obrigatório"
6 }
```

**PASS** Status code é 400 [↗](#)

**PASS** nome é obrigatorio [↗](#)

**PASS** email é obrigatorio [↗](#)

**PASS** password é obrigatorio [↗](#)

**PASS** administrador é obrigatorio [↗](#)

## 5. Excluir por ID invalido DEL

- Objetivo: Validar EXCLUIR usuarios cadastrados não passando parametros, deve retornar informando nenhum registro excluido.
- Passos: Enviar requisição DEL sem parametros.
- Resultado esperado:

```
1 {
2 "message": "Nenhum registro excluído"
3 }
```

**PASS** Status code é 200 [↗](#)

**PASS** Mensagem é correta [↗](#)

## 5. Excluir por ID, valido DEL

- Objetivo: Validar EXCLUIR usuarios cadastrados dados validos, deve retornar informando registro excluido.
- Passos: Enviar requisição DEL parametro ID valido
- Resultado esperado:

```
1 {
2 "message": "Registro excluído com sucesso"
3 }
```

**PASS** Status code é 200 [↗](#)

**PASS** Mensagem é correta [↗](#)

## Tipos de Testes Complementares [↗](#)

### Testes de Desempenho [↗](#)

#### 1. Logins simultâneos

- Objetivo: Validar os **testes de carga** com muitos logins simultâneos

#### 2. Registros simultâneos

- Objetivo: Validar criação dos registros de usuários em massa.

### 3. Edição simultâneas

- Objetivo: Validar a edição de registros de usuários em massa.

### 4. Consultas simultâneas

- Objetivo: Validar consulta dos registros de usuários em massa.
- Passos:
  - A API foi executada utilizando o Docker, configurada para ser acessada localmente através da URL `localhost:3000`
  - Para a execução dos testes, utilizou-se o Newman, ferramenta de linha de comando para executar coleções do Postman.
  - A flag `-n` do Newman foi utilizada para definir a quantidade de iterações a serem realizadas, sendo configurada para 120 iterações neste caso
- Resultado esperado:
  - **Não deve** ser permitido mais que 101 interações

## Testes de Segurança [↗](#)

### 1. Segurança do token

- Objetivo : Validar nas funcionalidades principais, como criação, leitura, atualização e exclusão de usuários o Token JWT não validado
- Passos: Nos testes das funcionalidades principais, como criação, leitura, atualização e exclusão de usuários não informar token valido
- Resultado esperado:
  - Todas as operações **não devem** ser permitidas sem um token válido.