**NOS Automation Testing**

NOTA: A API esteve em baixo durante a execução do teste o que dificultou a execução de alguns testes

1. **Escolher uma ferramenta para automatizar testes à seguinte Api REST, explica o porquê dessa escolha. https://gorest.co.in/**

Optei pelo **Postman** porque:

* É uma ferramenta que tem uma interface intuitiva e que permite a fácil criação, execução e a gestão de testes nas APIs.
* Suporta vários tipos de HTTP Requests como GET, POST, PUT, DELETE e outros menos utilizados como PATCH permitindo, assim, testar de forma abrangente uma API.
* Permite a utilização/criação de scripts de testes personalizados que podem correr tanto no pré-request ou pós-response
* Permite efetuar uma rápida validação de schema para verificar se a estrutura dos dados enviados pela API está conforme pretendido.
* Permite a partilha de collections e environments com colegas, contribuindo assim para uma consistente e rápida testagem por outros colegas
* O postman é bastante conhecido e a comunidade online é grande, podemos então encontrar muita informação online e perguntas e respostas por parte da comunidade, ajudando assim a pesquisar de forma mais fácil soluções para certos testes caso necessário.

1. **Explica os use case de teste**

**Testes Funcionais**

**Operações CRUD:  
 Nota: É necessário um Bearer Token para a Authorization, foi criada uma variável {{gorest}} para** [**https://gorest.co.in/public**](https://gorest.co.in/public)**, exemplo de {id} para teste pode ser 7409746, {post\_id} como 156025, {comment\_id} como 120545,**

**USERS:**

**Teste 1:** **Criar utilizador** 🡪 Enviar **POST Request** para {{gorest}}/v2/users com o devido Bearer Token e dados de utilizador válidos.  
 **Output esperado:** Novo utilizador criado e código de status **201 Created.**

**Teste 2:** **Obter utilizador(es)** 🡪 Enviar **GET Request** para {{gorest}}/v2/users/{id} para obter um utilizador ou enviar **GET Request** para {{gorest}}/public/v2/users para obter todos os utilizadores.  
 **Output esperado:** API devolve os dados do utilizador {id} ou de todos os utilizadores e um código de status de **200 OK.**

**Teste 3:** **Atualizar utilizador** 🡪 Enviar **PUT ou PATCH Request** para {{gorest}}/v2/users/{id} com devidos dados atualizados.  
 **Output esperado:** Dados do utilizador atualizados e um código de status de **200 OK.**

**Teste 4:** **Eliminar utilizador** 🡪 Enviar **DELETE Request** para {{gorest}}/v2/users/{id}.  
 **Output esperado:** Utilizador eliminado e a API devolve um código de status de **204 No Content.**

**POSTS:**

**Teste 5:** **Obter Posts de utilizador** 🡪 Enviar **GET Request** para {{gorest}}/v2/users/{id}/posts para obter posts de um utilizador ou {{gorest}}/v2/posts/ para obter todos os posts  
 **Output esperado:** API retorna a lista de posts com um código de status de **200 OK.**

**Teste 6:** **Criar post** 🡪 Enviar **POST Request** para {{gorest}}/v2/users/{id}/posts com os dados necessários para publicar um post tais como **title** ou **body**  
 **Output esperado:** É criado um post para o utilizador {id} e a API retorna um código de status de **201 Created.**

**Teste 7:** **Atualizar post** 🡪 Enviar **PUT ou PATCH Request** para {{gorest}}/v2/posts/{post\_id} com devidos dados atualizados tais como **title** ou **body**.  
 **Output esperado:** Dados do post atualizados e um código de status de **200 OK.**

**Teste 8:** **Eliminar post** 🡪 Enviar **DELETE Request** para {{gorest}}/v2/posts/{post\_id}.  
 **Output esperado:** Post eliminado e a API devolve um código de status de **204 No Content.**

**COMMENTS:**

**Teste 9:** **Obter comentários de posts** 🡪 Enviar **GET Request** para {{gorest}}/v2/posts/{post\_id}/comments para obter comentários de um post ou {{gorest}}/v2/comments para obter todos os comentários de vários posts  
 **Output esperado:** API retorna a lista de comentários com um código de status de **200 OK.**

**Teste 10:** **Criar comentário** 🡪 Enviar **POST Request** para {{gorest}}/v2/posts/{post \_id}/comments com os dados necessários para publicar um comentário tais como **name, email** e **body**  
 **Output esperado:** É criado um novo comentário para o post e a API retorna um código de status de **201 Created.**

**Teste 11:** **Atualizar comentário** 🡪 Enviar **PUT ou PATCH Request** para {{gorest}}/v2/comments/{comment\_id} com devidos dados atualizados tais como **bod, name** ou **email**.  
 **Output esperado:** Dados do comentário atualizados e um código de status de **200 OK.**

**Teste 12:** **Eliminar comentário** 🡪 Enviar **DELETE Request** para {{gorest}}/v2/comments/{comment\_id}.  
 **Output esperado:** Comentário eliminado e a API devolve um código de status de **204 No Content.**

**User Todos:**

**Teste 13:** **Obter ‘Todos’ de utilizador** 🡪 Enviar **GET Request** para {{gorest}}/v2/users/{id}/todos para obter posts de um utilizador ou {{gorest}}/v2/todos/ para obter todos os posts  
 **Output esperado:** API retorna a lista de ‘Todos’ com um código de status de **200 OK.**

**Teste 14:** **Criar ‘Todo’** 🡪 Enviar **POST Request** para {{gorest}}/v2/users/{id}/todos com os dados necessários para publicar uma todo tais como **title** ou **status**  
 **Output esperado:** É criada uma ‘todo’ para o utilizador {id} e a API retorna um código de status de **201 Created.**

**Teste 15:** **Atualizar ‘Todo’** 🡪 Enviar **PUT ou PATCH Request** para {{gorest}}/v2/todos/{todo\_id} com devidos dados atualizados tais como **status** ou **title**.  
 **Output esperado:** Dados da ‘todo’ atualizados e um código de status de **200 OK.**

**Teste 16:** **Eliminar ‘Todo’** 🡪 Enviar **DELETE Request** para {{gorest}}/v2/todos/{todo\_id}.  
 **Output esperado:** ‘Todo’ eliminada e a API devolve um código de status de **204 No Content.**

**Paginação:**

**Teste 17:** **Verificação da paginação predefinida** 🡪 Enviar **GET Request** para {{gorest}}/v2/users e registar número predefinido de users por página (neste caso foram 10).

Enviar **GET** para {{gorest}}/v2/users?page=2 e registar número de users recebido que tem de ser 10 users diferentes da primeira página.

**Teste 18:** **Verificação da quantidade de users recebidos** 🡪 Enviar **GET Request** para {{gorest}}/v2/users?per\_page=20 e registar número de users obtido por página

Repetir o teste para diferentes valores de users por página tais como 1, 5,15, 20, 25, 30 … até 100.   
  
Enviar **GET Request** para {{gorest}}/v2/users?page=2&per\_page=10 e registar número de users obtido por página

Repetir o teste para diferentes valores de users por página tais como 1, 5, 15, 20, 25, 30 … até 100. Confirmar que os users da primeira página não estão presentes nas próximas páginas 2, 3, 4 ..  
  
NOTA: Para os próximos testes, confirmar em **Headers**, se os valores de **X-Pagination-Total (número de resultados)**, **X-Pagination-Pages (número de páginas)**, **X-Pagination-Page (número de página atual)** e **X-Pagination-Limit (resultados por página)** correspondem com o que a API devolveu

**Teste 19:** **Verificação da última página**  🡪 Enviar **GET Request** para {{gorest}}/v2/users?page=32&per\_page=90 e confirmar que a última página tem menos que o **per\_page=90** se o número total de itens não for um múltiplo de **90  
  
Teste 20:** **Verificação de página vazia**  🡪 Enviar **GET Request** para {{gorest}}/v2/users?page=9999 (um número muito alto) e confirmar que a API retorna uma lista vazia  
  
**Teste 21:** **Verificação de per\_page=1**  🡪 Enviar **GET Request** para {{gorest}}/v2/users?per\_page=1 e confirmar que a API retorna apenas uma lista com 1 user.  
  
**Teste 22:** **Verificação de per\_page=100 (valor máximo)**  🡪 Enviar **GET Request** para {{gorest}}/v2/users?per\_page=100 e confirmar que a API retorna uma lista de 100 users.

**Teste 23:** **Verificação de valores incorretos para per\_page** 🡪 Enviar **GET Request** para {{gorest}}/v2/users?per\_page=-**1** e per\_page com **valores não numéricos** e confirmar que a API retorna um erro.

**Teste 24:** **Verificação de valores incorretos para page** 🡪 Enviar **GET Request** para {{gorest}}/v2/users?page=-**1** e page com **valores não numéricos** e valor **0** e confirmar que a API retorna um erro.

**Teste 25:** **Verificação de média dos tempos de resposta** 🡪 Enviar **GET Request** para {{gorest}}/v2/users?page=**100** e efetuar 50 vezes o request para obter uma pequena amostra e saber a média de tempo de resposta.

**Teste 26:** **Página não encontrada, erro 404** 🡪 Enviar **GET Request** para {{gorest}}/v2/teste404 (ou qualquer link que não existe) para verificar se obtemos um erro **404 Not Found**.

**Teste 27:** **Parâmetros inválidos** 🡪 Enviar **GET Request** para {{gorest}}/v2/users?testex (ou qualquer parâmetro não suportado) para verificar como a API mostra estes erros e se envia um erro de **400 Bad Request** ou **422 Unprocessable Entity.**

**Dados:**

**Teste 28:** **Dados com formatos errados** 🡪 Enviar **POST Request** para criar um novo utilizador e enviar dados com formato errado ou em falta, como por exemplo um email, para verificar como a API mostra estes erros e se envia um erro de **400 Bad Request** ou **422 Unprocessable Entity e/ou uma mensagem de erro de dados em falta**

**Teste 29:** **Credenciais inválidas (API Key)** 🡪 Enviar **GET Request** para {{gorest}}/v2/users sem ter uma API key implementada para verificar como a API mostra estes erros e se envia um erro de **401 Unauthorized.**

**Teste 30:** **Envia de dados de grande porte** 🡪 Enviar **POST Request** para criar um novo utilizador e enviar strings grandes para verificar como a API lida com dados de grande porte, e.g “email”: “fdfsdfsdjmfdiofjmsdifmsdijmfsdmfisdjfmisdfjis(…)”

**Carga:**

Podemos efetuar testes de carga com o Jmeter, porém, sem avisar o host ou sem fazer estes testes num ambiente isolado (servidor remoto de testes ou semelhante), poderá ser considerado um ataque DdoS o que não é ideal se não soubermos que tipo de infraestrutura está por trás da API.  
Um exemplo de teste de carga pode ser:  
  
**Teste 31:** **Enviar um grande número de GET Requests** 🡪 Enviar vários **GET Request num curto espaço de tempo** para {{gorest}}/v2/users e verificar como a API lida e se envia um erro do tipo **429 Too Many Requests**

1. **Em automação, com resposta desta API gorest.co.in/public/v2/todos:**

**3.1 Aplica uma validação de schema ao resultado;**

A computer screen with colorful text

Description automatically generated

**3.2 Valida se todos os resultados têm status completed;**

A screen shot of a computer code

Description automatically generated

* 1. **Interpreta e valida o valor “due\_on”**

A screen shot of a computer code

Description automatically generated  
  
**Repositório github: https://github.com/JoaoCBOliveiraa/SchemaValid\_Gorest**

1. **DevOps, CI/CD:**

**4.1 Explica e justifica uma implementação de testes de carga a esta API**

Os testes de carga são essenciais para perceber como uma API funciona sob vários níveis de tráfego. O objetivo deste tipo de testes é garantir que a API consegue aguentar um determinado nível (muitas das vezes de teor elevado) de tráfego sem que este sofra quaisquer problemas de performance e, consequentemente, períodos de inatividade. Sendo esta API maioritariamente para aprendizagem e explorar testes em APIs, um teste de carga nesta API não irá afetar outras webapps. Para tal, podemos efetuar os testes com uma ferramente chamada **JMeter** que é muito usada para efetuar testes de carga em APIs.  
Com estes testes de carga podemos então, **verificar se a API aguenta períodos de tráfego elevado mantendo assim a sua funcionalidade, podemos identificar possíveis bottlenecks e facilitar a implementação de melhorias e prevenir períodos de inatividade estando preparados com o conhecimento de quando estes bottlenecks podem acontecer.**

**4.2 Como implementarias uma solução de Continuous Testing, justifica**

A implementação duma solução de Continuous Testing na pipeline é essencial para obter uma resposta imediata sobre possíveis erros de software.

Para tal, podemos utilizar ferramentas como o **Jenkins** ou Gitlab para adicionar etapas à pipeline tais como testes de unidade, testes de integração, testes de API e testes de carga. O Jenkins é a ferramenta mais conhecida para criar pipelines de CI/CD e tem uma grande comunidade, tornando assim fácil a obtenção de informação online. Podemos assim implementar uma robusta pipeline que automatiza todo o ciclo de desenvolvimento de software e integração de código no comissionamento.  
Com a instalação do Jenkins, podemos instalar plugins como o Git e Docker (Version control e containerization).

Podemos:

1. Customizar uma pipeline de Jenkins com **Groovy**
2. Automatizar builds e testes com **Postman**, **Junit**, **Selenium** e conseguimos, então, compilar código, correr e integrar testes e gerar relatórios.
3. Adicionar etapes de comissionamento de código com o **Docker**
4. Gerir infraestruturas com o **Terraform**
5. Integrar ferramentas de monitorização e agrupar e analisar relatórios
6. Configurar notificações de possíveis erros ou detalhes dos testes para serem enviadas para ferramentas de comunicação com a equipa como o **Slack**
7. Implementar mecanismos de backup/segurança