KIT DE SOBREVIVÊNCIA EM TESTES DE API REST

7 dicas importantes para você que está sem rumo e já deveria ter começado a testar

Escrito por Antonio Montanha e Júlio de Lima

KIT DE SOBREVIVÊNCIA EM TESTES DE API REST

Antonio Montanha

about.me/ammontanha

Antonio Montanha é Engenheiro QA Sênior, atuando na área de testes há mais de 5 anos. Formado em Engenharia de Produção com ênfase em software, já trabalhou com aplicações Web, Desktop e especialmente com APIs Rest, entusiasta da parte de arquitetura de software, sempre tentando inserir a qualidade desde a concepção dos projetos.

Júlio de Lima

about.me/juliodelimas

Júlio de Lima é Engenheiro QA Principal, possui experiência em testes envolvendo aplicações Web, Desktop, Mobile e Serviços. Atuando desde 2012 com testes em APIs Rest. É formado em Tecnologia em Informática, especialista em Docência no Ensino Superior e é mestrando em Engenharia Elétrica e Computação com foco em Inteligência Artificical no Mackenzie com o intuito de solucionar problemas de teste com IA e ML.

Hoje é 11 de maio de 2020 e essa é a primeira edição desse e-book.

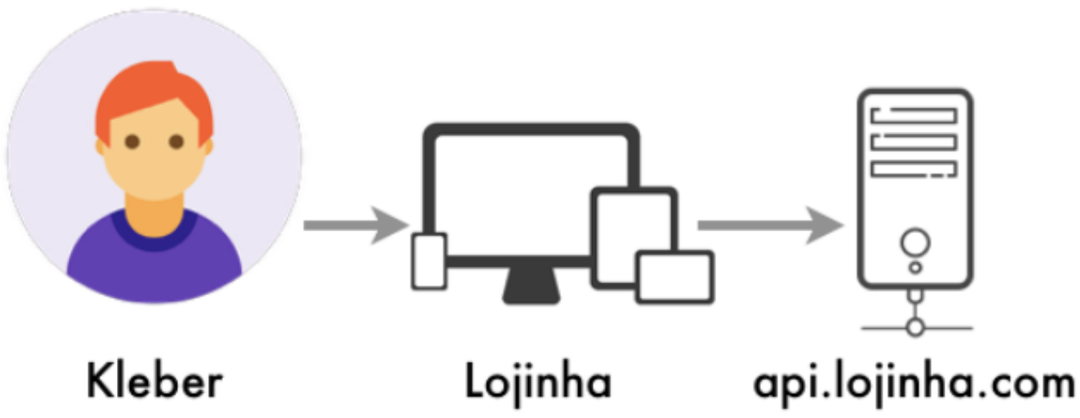
São Paulo, Brasil

Download realizado a partir do link bit.ly/kitSobrevivenciaTestesAPIRest

1

A partir de agora você é o App, sua vida de Kleber chegou ao fim!

Antes de nos preocupar com o REST, vamos entender o que é uma API, o nome é uma sigla para Application Programming Interface, que em português seria: Interface de Programação de Aplicações. É comum as pessoas terem dificuldade de entender o que é uma API, isso porque ela não é visível, ela se trata de várias rotinas e padrões para que as mais diversas aplicações possam se integrar com ela. Logo, podemos dizer que uma API é uma espécie de ponte.



Aqui, vemos que o Kleber usa o App da Lojinha, nossa loja virtual disponível no iOS, Android, Web e em SmartTVs e todos conectam-se à API.

Em meus testes, sempre fui o Kleber!

Infelizmente, em um contexto de testes de API, você não será mais o Kleber. Invés disso, você será o App! Exatamente, terá de aprender a conversar com a API da Lojinha em seu idioma, por meio de bilhetinhos, chamados Requisição e Resposta. Você, App, pede e a API pensa e te responde. Você então vê se ela está se comportando direitinho ;)

2

Requisição, Resposta, Método, Status Code, Token, JSON e essas coisas de APIs Rest

APIs podem ser implementadas de diversas formas, umas delas é seguindo uma arquitetura chamada Rest (Transferência Representacional de Estado), que é o foco desse e-book! Os itens do título dessa lição descreve alguns itens contidos em APIs Rest.

Essa caixa é uma Requisição, lembre-se dela!

A Lojinha API vai armazenar as regras de negócio da Lojinha e será responsável pelos recursos que a lojinha possui, como Produtos, Vendas, etc. Logo, se quero testar se a Lojinha é capaz de permitir que alguém com os dados do Kleber consiga fazer login e ver os produtos dele, preciso ser o App e conversar com a API Rest da Lojinha enviando os dados do Kleber e esperar que ela me dê um Token se eu informar os dados do Kleber corretamente. Quando o App quer requisitar algo à API Rest da Lojinha, ele envia uma Requisição. A API Rest da Lojinha, por sua vez processa a informação da Requisição, então devolve uma Resposta. Ambas tem Cabeçalho e Corpo, mas apenas a Requisição tem URI e Método.

Para que o App consiga se comunicar com a API Rest da Lojinha, é necessário saber qual é o endereço de onde ela está localizada e também qual recurso queremos utilizar. Já sabemos essas informações. O endereço é http://api.lojinha.com e o recurso é o login, logo, acabamos de ter nossa URI: http://api.lojinha.com/login. Assim, preencheremos ela naquela caixa da Requisição da página anterior.

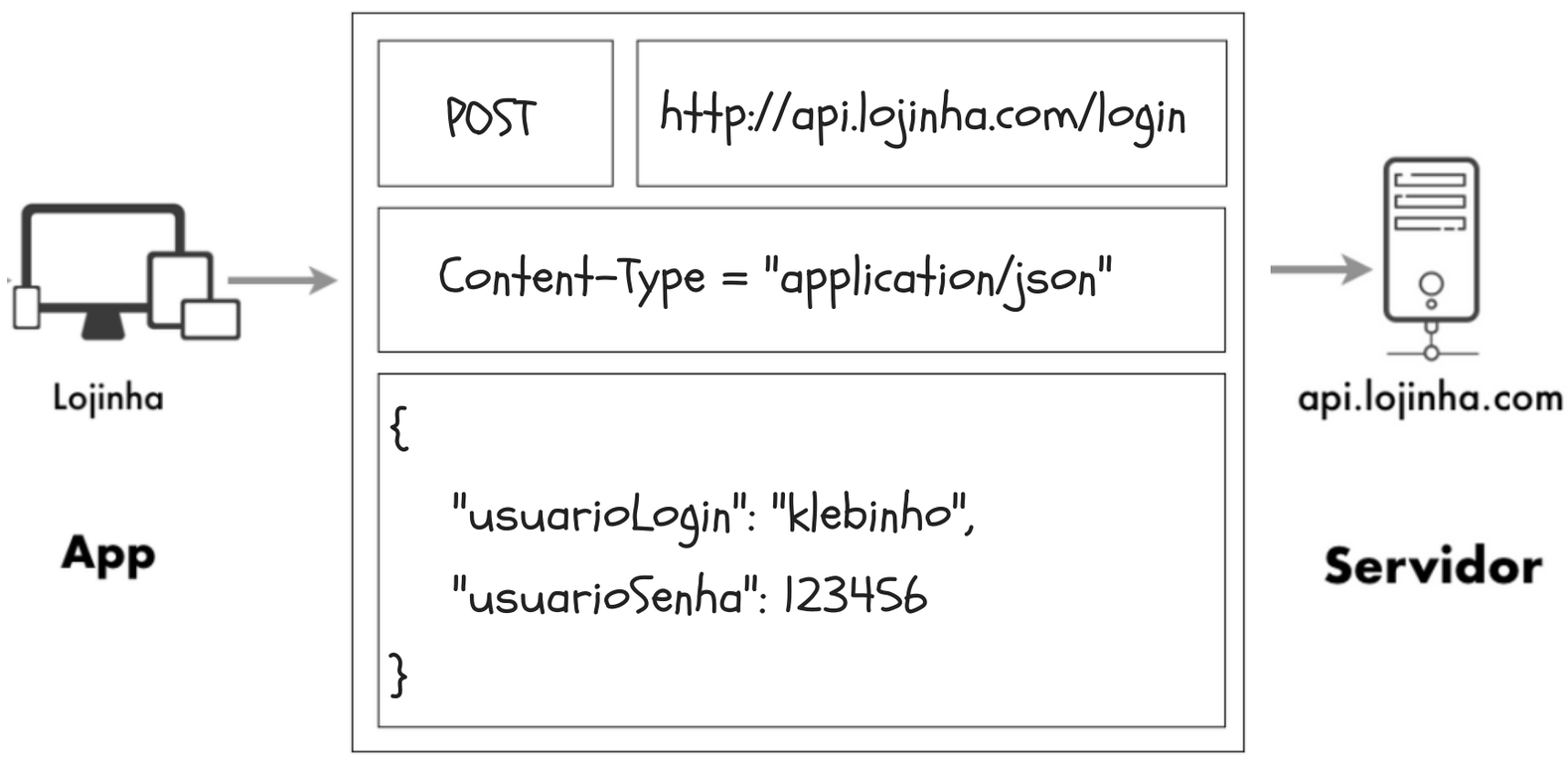
Além da URI também precisamos do Método, ele serve para dizer à API Rest qual ação ela precisa executar, por exemplo, "Registrar". Há 4 métodos muito conhecidos em Rest: Buscar, representado por GET; Registrar, representado por POST; Alterar, representado por PUT; e Remover, representado por DELETE. Logo, se nosso teste envolve registrar o Login, o método que colocaremos na caixa de Requisição é o POST.

Por fim, antes de enviarmos a Requisição para a API Rest da Lojinha, teremos que informar os dados do Kleber. Para isso, preciso descrever os dados dele no formato JSON:

```
{
   "usuarioLogin": "klebinho",
   "usuarioSenha": 123456
}
```

No formato JSON temos chaves representando um objeto. Um objeto tem atributos, no caso, usuarioLogin e usuarioSenha. Atributos tem valores, "klebinho" e 123456.

Veja que os valores tem tipos. O usuarioLogin possui aspas duplas, o que significa que permite texto com números, ou seja, alfanumérico. Já o usuarioSenha é um atributo numérico. Como alguém que testa aplicações, eu poderia tentar enviar um tipo diferente de dado para entender se a API Rest da Lojinha seria capaz de lidar com isso, mas não vamos fazer isso agora. Como estamos enviando um JSON no corpo, precisamos informar isso no cabeçalho, usando um Content-Type com o valor "application/json". O JSON fica no corpo da Requisição. Nesse momento, nossa Requisição está assim:



Agora ela já pode ser enviada! Ao fazer isso, a API Rest da Lojinha recebe, converte e extrai os dados do corpo da Requisição e pesquisa no banco de dados se existe um usuário klebinho com senha 123456, se sim, ela devolve uma Resposta que possui um Cabeçalho e um Corpo. No cabeçalho, um código chamado Status Code, que é uma lista de números que descrevem a natureza da Resposta. O 201 significa "Criado", no caso, o registro de login do Kleber. No corpo, veio o Token do Kleber. O Token serve para lembrar a API Rest da Lojinha quem é que a está chamando. Logo, guarde bem o Token, caso contrário nunca verá os produtos que pertencem ao Kleber!

```
{
    "token": "761b69db-ace4-49cd-84cb"
}
```

Bom, agora você já viu que seu teste teve sucesso, ou seja, você informou os dados do Kleber, teve o seu 201 e conseguiu o Token do Kleber. Um próximo teste poderia ser o de buscar os produtos do Kleber e sabemos que o único produto que o coitado possui é uma bolacha Trakinas, sim isso mesmo BOLACHA não BISCOITO. Lembre-se, para Buscar, usaremos o GET. Mas agora, como queremos apenas os produtos do Kleber, temos que informar seu Token no Cabeçalho. Como nada será registrado desta vez, não precisamos de um corpo, veja:

O Status Code que representa o sucesso da busca é o 200, e veja que legal, o teste passou, foi o que eu obtive como Resposta, juntamente com o JSON abaixo no corpo:

```
[
    {
        "produtoNome": "Trackinas",
        "produtoValor": 3.59,
        "produtoFavorito": true
    }
]
```

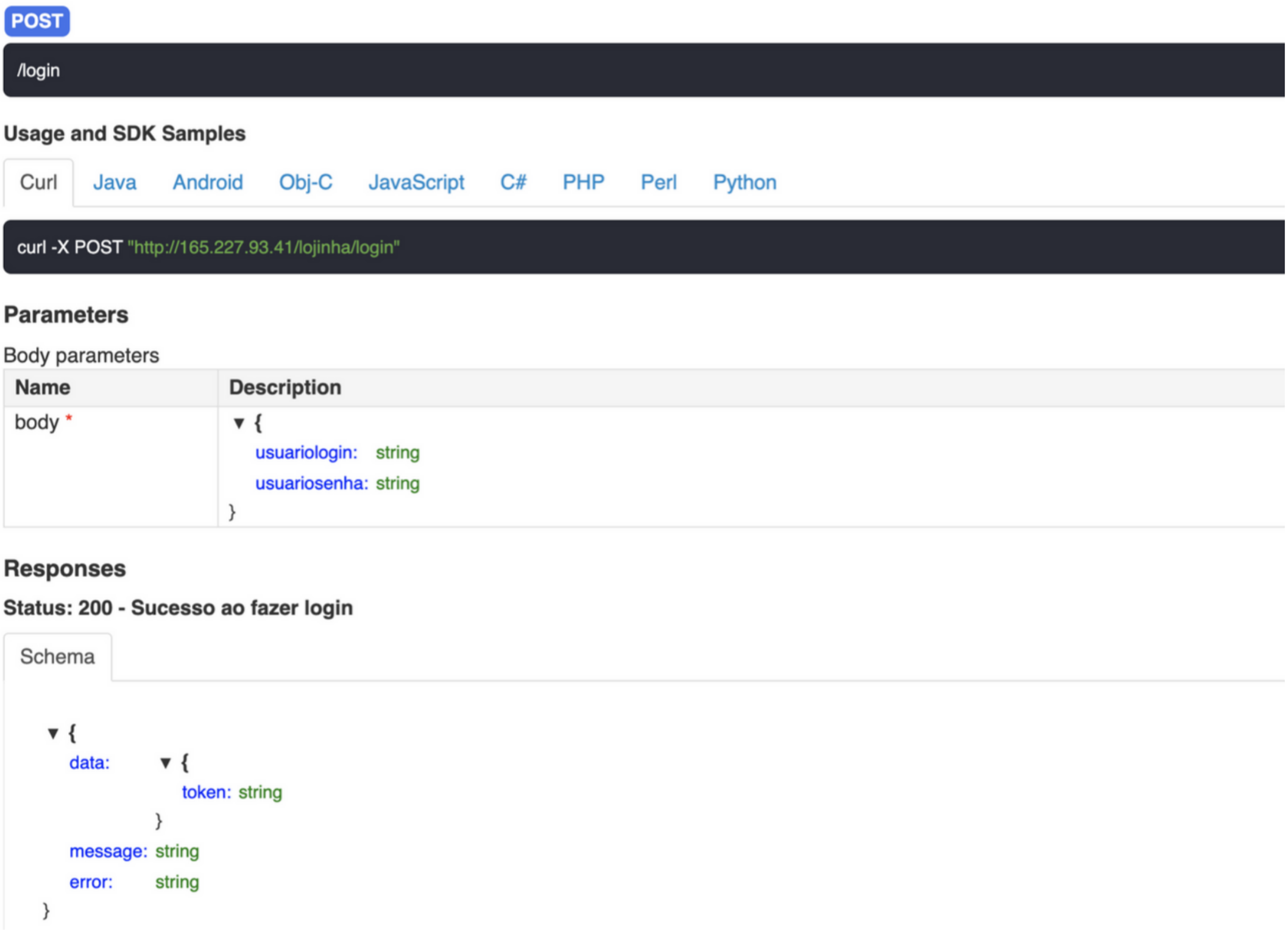
O colchetes servem para criar uma lista, bacana né? Além disso, vemos dois dados novos: Decimal (3.59) e Booleano (true), ambos possíveis em JSONs.

Mas espera ai, Júlio... Antonio!!!

De onde vocês tiraram essas URIs, Recursos, Métodos, Corpos de Requisições e de Respostas?

Há um modelo de documentação de contrato da API Rest, chamado Swagger. Nele não temos as regras de negócio, mas temos a lista de todas as URIs, Recursos, Métodos, Corpos de Requisições e de Respostas. Com base no Swagger é que eu identifico se, além de tratar bem as regras de negócio, se a API Rest da Lojinha também tem uma boa arquitetura.

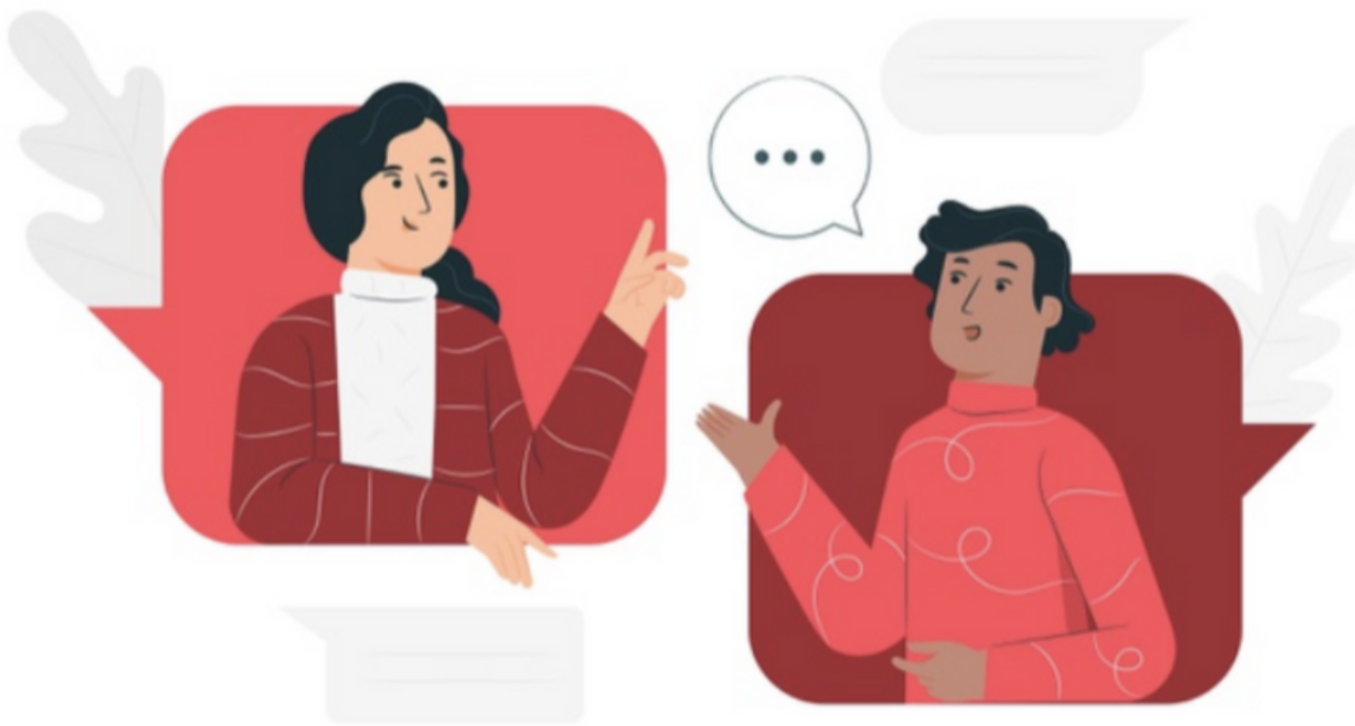
Veja abaixo um exemplo:



Um exemplo extraído da API Rest da Lojinha.

3

Foque nas regras, mas nunca abandone o não funcional, ele também é essencial!



Mídia social vetor criado por stories - br.freepik.com

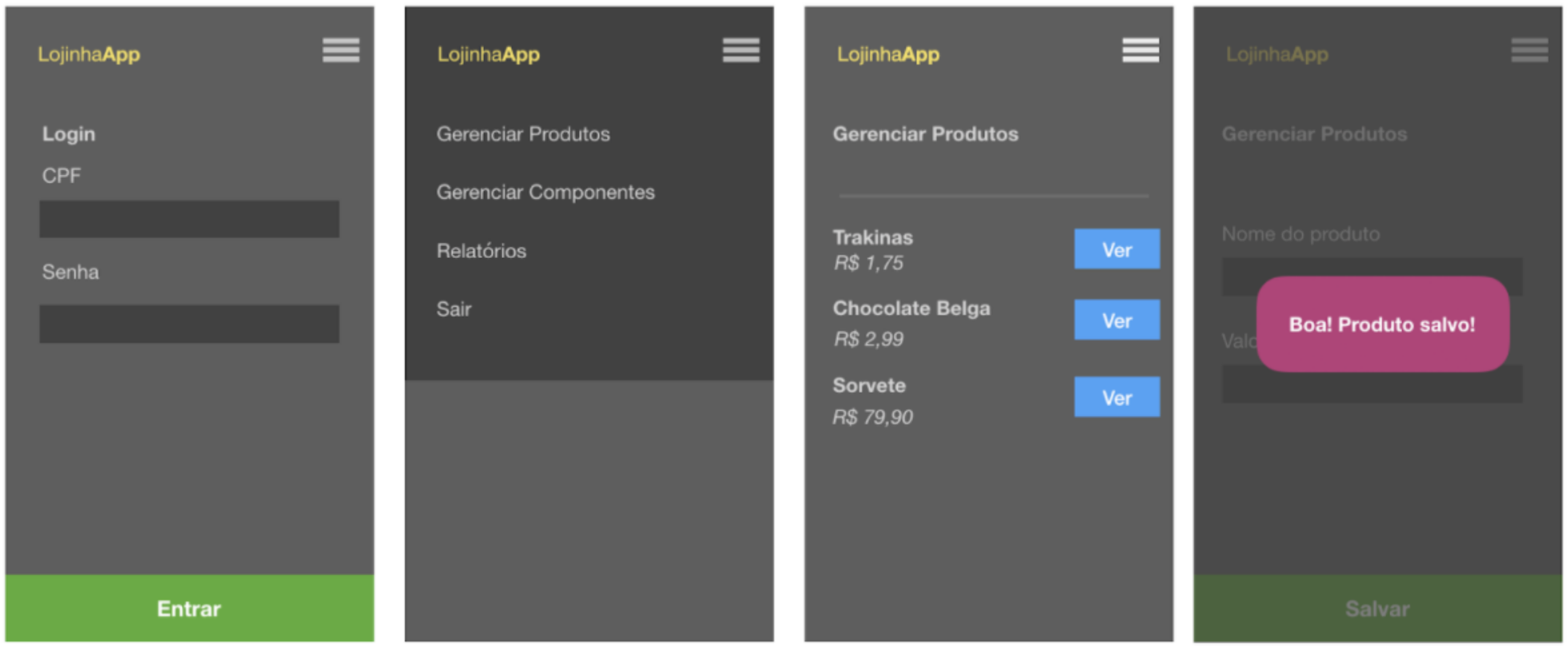
Em uma conversa entre duas pessoas pode ocorrer que informações certas sejam trocadas de maneira errada. Por exemplo, quando um fala baixo e outro alto demais, ou lento e rápido demais, etc. Podemos comparar essa conversa com a que fazemos entre um App e uma API Rest. A Requisição pode ser correta, a Resposta também, mas a resposta que era esperada como Alfanumérico, vem em Numérico. Ou que esperou-se 20 segundos até obter a resposta. Ou que eu tenha buscado Produtos do Kleber, mas tenha vindo Produtos da Fernanda.

Esses comportamentos revelam problemas não funcionais, de Interoperabilidade, Performance e Segurança que podem ser exercitados durante o envio de Requisições e obtenção de Respostas.

Há ferramentas que te ajudam nesses tipos de teste, como: pact-js, Apache JMeter e SoapUI.

4

Antecipe o máximo possível e deixe Myers orgulhoso



POST /login GET /produto POST /produto

Se você tivesse que testar o App da Lojinha no cenário de cadastro de produto e não tivesse acesso a API Rest da Lojinha, provavelmente iria ter que aguardar até que a maioria das telas estivessem criadas e conectadas para começar seus testes, não é? Mas, ao testar APIs Rest você não precisa esperar tanto para ter acesso a um recurso e poder testá-lo, logo, use e abuse desse recurso. Assim que identificar qualquer um dos recursos utilizados nesse cenário (POST login, GET produto ou POST produto) ficaram prontos, comece seus testes!

Você pode testar a busca por produtos sem que o registro de produtos esteja pronto, basta cadastrar os registros na base de dados, de modo que não seja necessário aguardar até que todos estejam prontos!

5

Use alguma ferramenta pra suportar seus testes e outra para automatizá-los

Okay, Antonio.. beleza Julio... entendi agora como funciona uma API Rest e já sei até como testá-la, mas qual ferramenta posso usar para enviar a Requisição e avaliar a Resposta se não vou fazer isso pelo App?

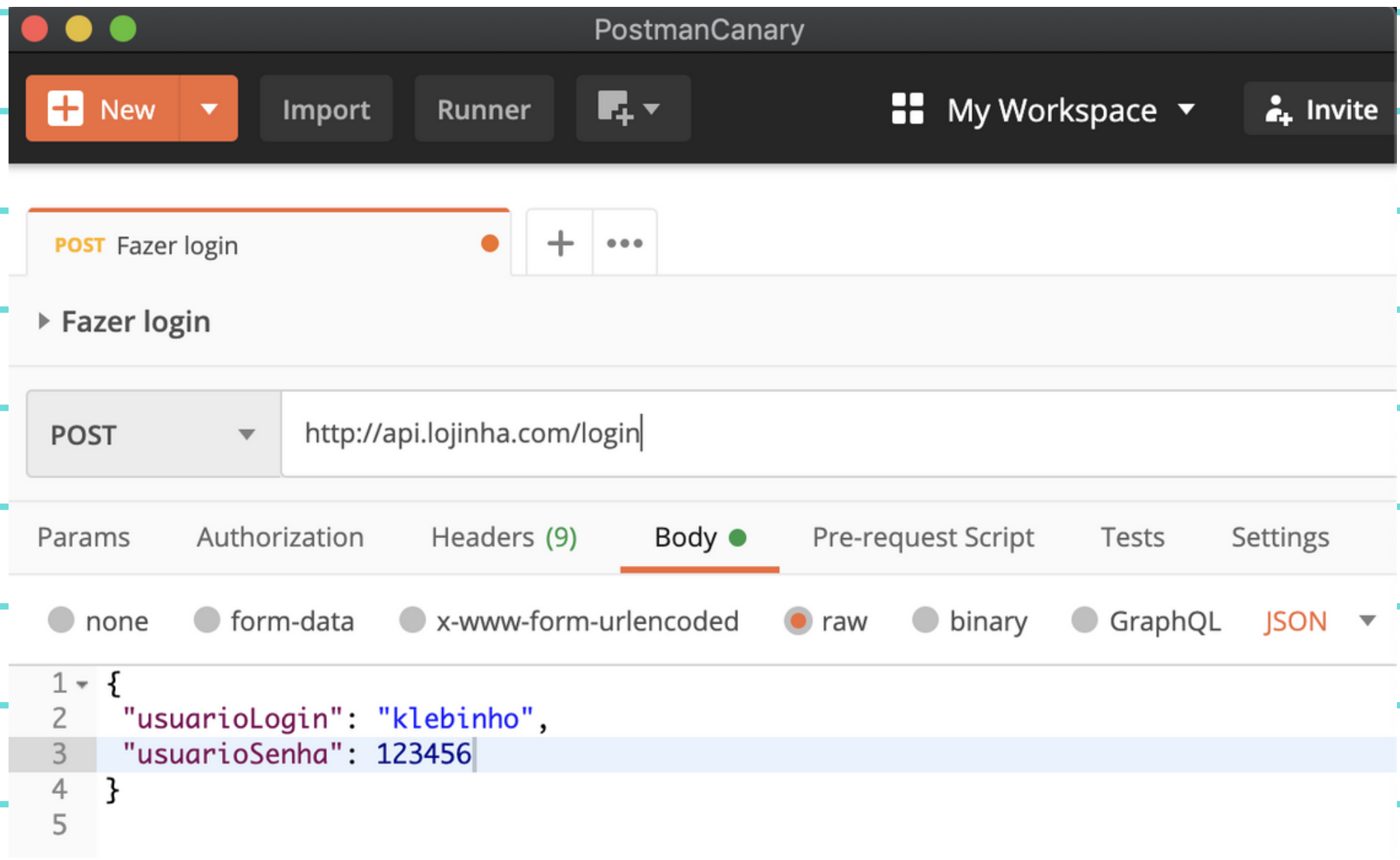
cURL no Unix

```
curl -d '{ "usuarioLogin": "klebinho", "usuarioSenha": 123456 }' -H
"Content-Type: application/json" -X POST http://api.lojinha.com/login
```

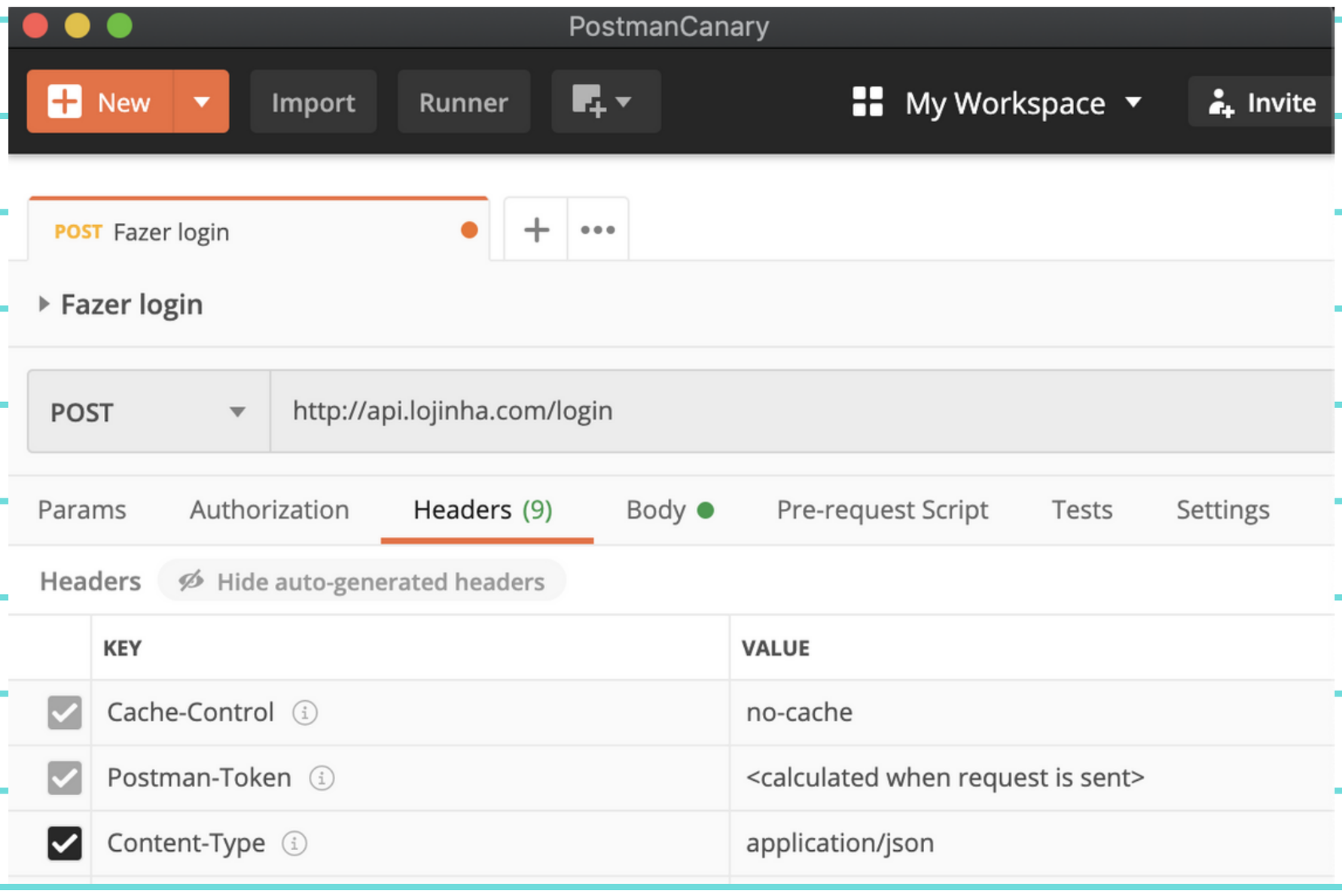
O cURL é uma aplicação nativa nos sistemas operacionais Unix, chamada via linha de comando, que pode ser utilizada para enviar requisições HTTP. Usá-la para fazer requisições a uma API Rest é algo bem simples, como vimos. O parâmetro -d descreve o Corpo da Requisição, o -H descreve o Cabeçalho e o -X descreve o método. Depois disso tudo, você adiciona a URI do recurso para onde irá enviar a requisição e boom! Dá um ENTER e avalia a resposta.

Okay, talvez seja muito roots...

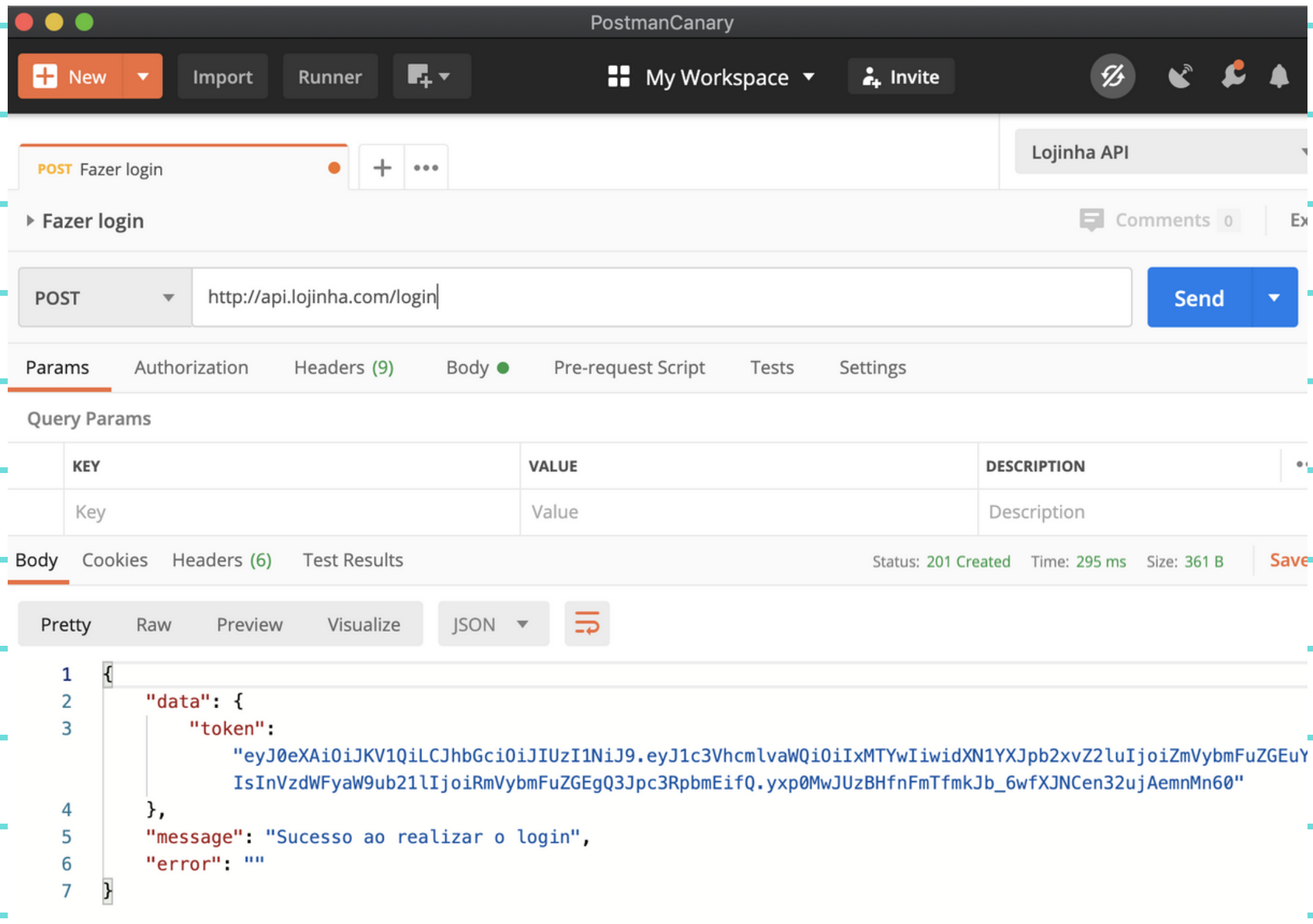
No Postman



Pois é, como vemos, no Postman você tem os mesmos recursos que o cURL, mas com uma interface gráfica, o que te ajuda a escolher o Método, apontar a URI, colocar o Corpo e ainda escolher o tipo do conteúdo contido no corpo (ali em laranja, ao lado da palavra GraphQL) e isso cria automaticamente um item de Content-Type na aba "Headers". Olha só...



E o que acontece depois de enviar a Requisição?



Essa é a tela que você vai ver após enviar uma Requisição de login com sucesso. Vemos muitas coisas aqui, dentre elas, o corpo da resposta com o Token (na aba Body) e o Status Code 201, dizendo que o registro do login foi criado com sucesso. Também vemos o tempo de resposta, nesse caso, 295 milissegundos.

No corpo, por exemplo, validar alguns quesitos como:

- Regras de negócio foram atendidas?
- Tipagem de dados atendem ao Swagger?
- Cada método faz o que deveria fazer?
- A estrutura da resposta segue o Swagger?
- Os código dos estados HTTP estão configurados corretamente?
- E o Tempo de Resposta, tá bacana?

O próximo passo seria automatizar os testes

Bibliotecas como o restAssured podem servir justamente para isso e você pode escrever seus testes em Java ou C#, por exemplo.

Seus scripts são definidos com requisições pre-definidas e validações automáticas, vejamos abaixo um exemplo em Java:

```
@Test
public void testFazerLoginComKleberRetornaSucesso( ) {
   RestAssured.given( )
      .contentType(ContentType.JSON)
      .body("{ 'usuarioLogin': 'klebinho', 'usuarioSenha': 123456 }")
   .when()
      .post("http://api.lojinha.com/login")
   .then()
      .assertThat( statusCode(201);
```

A esse momento você já deve ser capaz de perceber o que está acontecendo. Usamos o método given( ) da classe RestAssured para informar o Content-Type como Cabeçalho e o Corpo como um parâmetro do método body( ). Além disso, usamos o método when( ) para determinar para qual método e URI iremos enviar nossa requisição. Por fim, usamos o método then( ).assertThat( ) para validar que o Status Code recebido é o 201.

Após ter escrito esse teste, poderemos executar ele sempre que o código da API Rest da Lojinha for alterado. Sendo utilizado para validar que aquele cenário continua funcionando mesmo após as alterações no código, bacana né?

6

Analise os Logs para ter visibilidade, pois o teste não termina quando surge um 500

Respostas de APIs Rest vem com um cabeçalho e um corpo, como vimos anteriormente. O corpo tem algo que já conhecemos, o JSON. Mas no cabeçalho, tem algo que tivemos um gostinho de o que é, o Status Code. Vimos que 200 mostra que foi possível buscar (GET) e que o 201 mostra que foi possível registrar (POST). Outro código bem comum que representa sucesso é o 204, geralmente usado quando removemos um recurso.

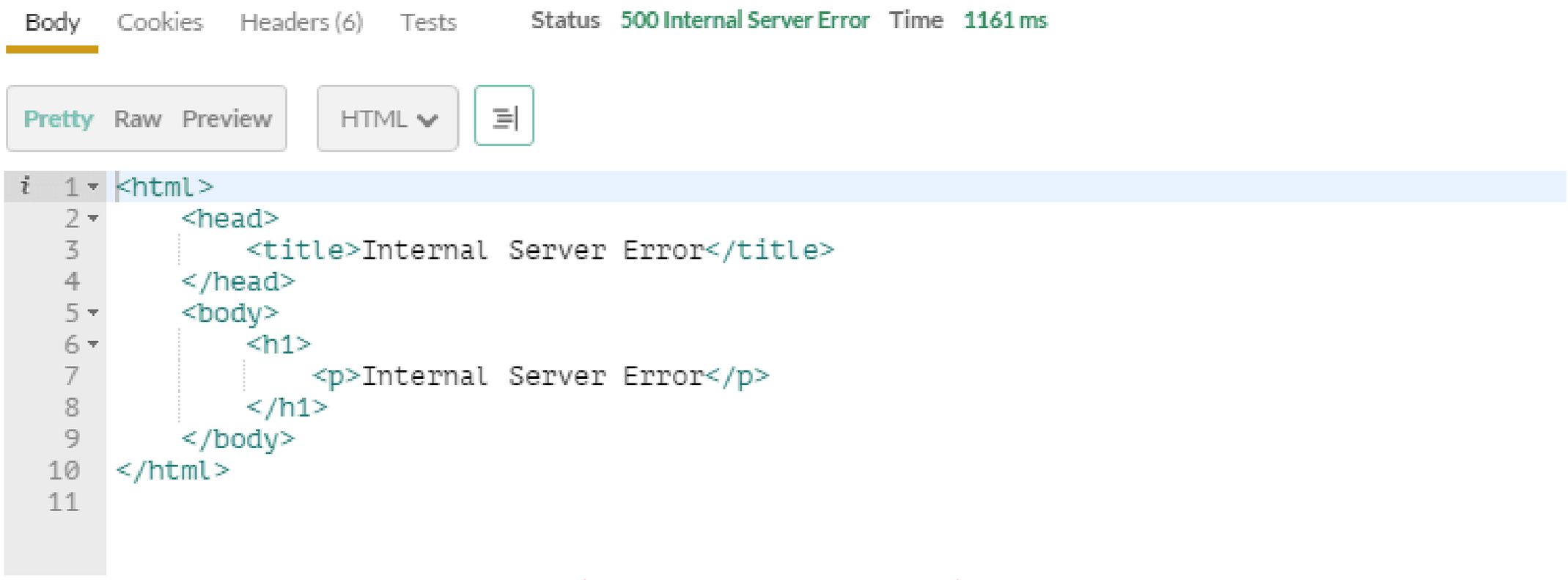
Mas nem tudo são flores, quero dizer...
Status Codes 200...

Quando a API Rest da Lojinha não reconhece seu Token, ela te responde um 401. Se você tenta acessar um recurso que não existe, por exemplo, um produto da Lojinha, o PlayStation, então ela te responde 404. Se você enviou usuarioLoin, invés de usuarioLogin, provavelmente ela irá responder um 400, dizendo que sua requisição não foi das melhores.

Mas, ninguém é tão temido no mundo de APIs Rest quanto os famosos erros 500. Eles ocorrem quando algo inesperado aconteceu e não foi tratado de maneira apropriada...

Hahaha, achei um 500!

É um bug!



Erro acontecido em 11/12/13 por volta das 01:00:43

Bem, no mundo de APIs Rest não é assim, você precisa ir além de apenas "reportar" um 500 que você viu na Resposta. Pois o resultado vai ser algo muito simplista e incerto. Quer confirmar que aquilo realmente é uma inconsistência? Olhe para os logs. Se olhar bem atentamente verá a causa raíz dos erros.

```
[Wed Dec 11 01:00:40 2013] [error] [client 192.168.11.11]

PHP Fatal error:

Call to undefined function Lojinha\\login\\getUsuarioGit()

in /lojinha/Controller/LoginController.php on line 249
```

No trecho do Log acima, vemos que por detrás daquela resposta de erro 500 que aconteceu as 01:00:40 do dia 11/12/13, na verdade, foi uma chamada a uma função inexistente (a getUsuarioGit( )) na linha 249 do arquivo LoginController.php. Será que quem desenvolvedor esqueceu de implementar este método? Ou está chamando o método errado? Ou ele está implementado, mas não foi importado corretamente? Sua colaboração com seu time vai aumentar muito!

KIT DE
SOBREVIVÊNCIA EM
TESTES DE API REST

7

Use esse checklist para te ajudar a testar

1. Regras de negócio foram atendidas?
2. Há campos obrigatórios na resposta para continuidade dos fluxos de tela?
3. Tipagem de dados atendem ao Swagger?
4. Passagem de parâmetros funcionam corretamente?
5. Sendo o Kleber, consigo ver os recursos da Fernanda?
6. Cada método faz o que deveria fazer?
7. A API Rest funciona certinho ao buscar 0, 1 e muitos recursos?
8. A estrutura da resposta segue o Swagger?
9. Os código dos estados HTTP estão configurados corretamente?
10. E o Tempo de Resposta, tá bacana?

KIT DE SOBREVIVENCIA EM TESTES DE API REST

Agradecimento especial aos revisores alunos do TSPI:
Bruna Fernandes, Diego Rocha, Matheus Barcelos, Paula Senna,
Priscila Alves, Rafael Costa e Tiago Pereira