02

Status code e a Interface Uniforme

Já sabemos como funciona um pouco o protocolo HTTP: quando faço uma requisição para o carrinho 1, ele faz uma requisição do tipo GET para `/carrinhos/1` e é o status code que me diz que foi tudo ok: `200`. O status code é quem indica se a requisição foi um sucesso ou um fracasso.

Quais outros status codes podemos conhecer? Se lembra que se colocarmos um número inválido, dá um erro no nosso servidor? Um erro interno? Vamos tentar um carrinho que não existe:

```
curl -v http://localhost:8080/carrinhos/10
```

O resultado é o status code 500, de erro interno:

```
< HTTP/1.1 500 Internal Server Error
< Date: Thu, 24 Apr 2014 14:40:22 GMT
< Connection: close
< Content-Length: 0
```

O que significa ele? Que o problema ocorreu no servidor, o culpado é o servidor, e não o cliente. Nosso código deu uma exception: erro 500. Legal, o status code já serve para dizer se deu sucesso ou falha. Mas olha que estranho, no nosso servidor fizemos que quando recebemos o post, devolveremos o status de sucesso ou erro através do xml, dentro do xml:

```
public String adiciona(String conteudo) {
    Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
    new CarrinhoDAO().adiciona(carrinho);
    return "<status>sucesso</status>";
}
```

Estranho! O http já faz isso para nós, qual o motivo de colocar isso dentro do XML, do corpo da nossa resposta, sendo que o cabeçalho já disse que foi 200, sucesso? Não devo fazer isso dentro do meu xml, devo usar o HTTP com os seus status code. Se eu devolver 200 e uma mensagem de erro não faz sentido para o HTTP, então é importante usarmos o status code como indicador de sucesso ou erro.

Quais outros códigos podemos aprender? Neste caso do POST específico, estamos criando um novo recurso, então além de ser um sucesso, o recurso foi criado no servidor: `Created`. Olhando a documentação do HTTP encontramos que o código `201` significa que um recurso foi criado, `Created`.

Como falo então no JAX-RS para devolver 201? Não quero devolver só um int. Quero devolver uma resposta, um `Response`:

```
public Response adiciona(String conteudo) {
```

Mas qual response quero devolver? Com o control+espaço, percebo que existe o

```
return Response.ok()
```

Mas não quero ok, 200, quero created, 201, existe? Sim, e ele recebe um parâmetro, qual a URI do recurso criado. Isto é, quando devolvemos um código 201 de criação de um recurso, devemos dizer ao cliente onde esse recurso foi criado, para que ele possa depois alterar, pesquisar ou até mesmo remover esse recurso, se suportarmos tais operações.

Qual a URI do nosso carrinho? `/carrinhos/id` . Mas queremos colocar o id do carrinho real, então:

```
String uri = "/carrinhos/" + carrinho.getId();
```

Mas o método não recebe uma String, ele recebe uma URI, então:

```
public Response adiciona(String conteudo) {
    Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
    new CarrinhoDAO().adiciona(carrinho);
    URI uri = URI.create("/carrinhos/" + carrinho.getId());
    return Response.created(uri);
}
```

A URI é, `/carrinhos/5` se o carrinho for o quinto. E agora que estou criando a minha resposta, mando finalizá-la, invocando o método `build` :

```
public Response adiciona(String conteudo) {
    Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
    new CarrinhoDAO().adiciona(carrinho);
    URI uri = URI.create("/carrinhos/" + carrinho.getId());
    return Response.created(uri).build();
}
```

Estou falando, ao invés de devolver o resultado da minha ação dentro do XML, devolvo o resultado no status code. Qual a outra vantagem de usar algo que já é padrão? É que todo mundo entende status code. Todos os programas e servidores da internet entendem o que um código 200, 201, 404 e 500 significam. Mas somente o seu programa entende o que significa `<status>sucesso</status>` . A única pessoa capaz de entender o conteúdo do seu xml é quem foi programado para entendê-lo, enquanto ao usarmos o padrão de status code do HTTP, essa interface uniforme de comunicação, a internet inteira é capaz de entendê-la.

Legal, estamos usando esse padrão uniforme. Reiniciamos o servidor, vamos no terminal e fazemos o post de um carrinho novo:

```
curl -v -d "<br.com.alura.loja.modelo.Carrinho>  <produtos>    <br.com.alura.loja.modelo.Produto
```

E temos o resultado com o 201

```
< HTTP/1.1 201 Created
< Location: http://localhost:8080/carrinhos/2
< Date: Thu, 24 Apr 2014 14:51:58 GMT
```

```
< Content-Length: 0
<
```

Note que o cabeçalho `Location` está indicando a URI de nosso novo carrinho. Não precisamos concatenar a parte do URI relativa ao nosso servidor e porta, o próprio servidor JAX-RS fez isso para nós. Sempre que devolvemos o resultado 201 e passamos a URI, o servidor devolve um header chamado `Location` que indica a localização do nosso recurso lá no servidor.

Se fizemos agora uma requisição GET para essa URI, teremos uma representação desse carrinho. Repara que agora passamos a trabalhar com mais partes dessa Interface Uniforme de comunicação, o status code, assim como os cabeçalhos, faz parte dessa interface. O POST e o GET também são características dessa interface, eles também são um padrão que toda a internet que usa HTTP entende o que eles significam. Se eu inventasse outros usos para o GET e o POST, outros programas não entenderiam o que eles fazem, ou pior ainda entenderiam de maneira errada. Ao usar a interface uniforme de maneira adequada, como ela foi especificada, todos os programas entendem o que está acontecendo.

Que outros códigos podemos usar? Não estamos mais devolvendo um XML na resposta, certo? Então devemos remover a anotação de `Produces` do nosso método de adiciona:

```
@POST
public Response adiciona(String conteudo) {
    Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
    new CarrinhoDAO().adiciona(carrinho);
    URI uri = URI.create("/carrinhos/" + carrinho.getId());
    return Response.created(uri).build();
}
```

Porém, além de não entregarmos nada, estou consumindo algo. No caso do POST estamos consumindo `APPLICATION_XML` enquando no GET estamos produzindo `APPLICATION_XML` . Se no GET tivemos que anotar para dizer ao servidor o que suportávamos, devemos fazer o mesmo no POST:

```
@POST
@Consumes(MediaType.APPLICATION_XML)
public Response adiciona(String conteudo) {
    Carrinho carrinho = (Carrinho) new XStream().fromXML(conteudo);
    new CarrinhoDAO().adiciona(carrinho);
    URI uri = URI.create("/carrinhos/" + carrinho.getId());
    return Response.created(uri).build();
}
```

Estamos agora falando a verdade: o que produzimos e o que consumimos. Reiniciamos o servidor e testamos novamente o POST:

```
< HTTP/1.1 415 Unsupported Media Type
< Date: Thu, 24 Apr 2014 14:57:03 GMT
< Content-Length: 0
```

A resposta agora é um 415, o media type utilizado para o envio (application/x-www-form-urlencoded) não é suportado pelo nosso servidor, que sabe interpretar somente `application/xml` . Note a importância do status code, ele foi

padronizado para dizer que o tipo de dado que você enviou eu, servidor, não consigo entender.

Claro, nós como clientes não enviamos o `Content-Type` adequado. Estamos enviando um xml mas não notificamos o servidor disso. Vamos adicionar o cabeçalho `Content-Type` à nossa requisição:

```
curl -v -H "Content-Type: application/xml" -d "<br.com.alura.loja.modelo.Carrinho>  <produtos>
```

Agora sim, o resultado volta a ser 201, o recurso foi criado com sucesso no nosso servidor.

Vamos ao nosso código de teste, o teste que suporta criar um novo carrinho. Nós enviamos APPLICATION_XML, mas ele assume que o resultado do conteúdo é status sucesso. Paramos o servidor, rodamos o teste e o mesmo falha. Claro, ele espera um xml de status, e não é mais isso, ele deve esperar agora um 201.

Portanto ao invés de recebermos a nossa `String` que é o conteúdo, queremos a resposta inteira:

```
Response response = target.path("/carrinhos").request().post(entity);
```

E agora podemos fazer um assert do código de status:

```
Assert.assertEquals(201, response.getStatus());
```

Legal. Apagamos o assert antigo, salvamos e rodamos o teste, tudo verde. O que falta agora? O 201 indica que ele criou, mas será que ele devolveu a URI certa no Location? Vamos conferir:

```
String location = response.getHeaderString("Location");
```

No caso do nosso exemplo isso poderia ser `http://localhost:8080/carrinhos/2` . Agora pego o nosso client e peço para acessar via GET essa URI:

```
String conteudo = client.target(location).request().get(String.class);
Assert.assertTrue(conteudo.contains("Microfone"));
```

Em nosso teste fazemos agora duas requisições: a primeira que posta a representação de um novo carrinho para o servidor, onde ele é criado, e é retornado 201 com a URI do carrinho no cabeçalho `Location` , e a segunda requisição que pega esse cabeçalho e busca a representação armazenada no servidor, conferindo que ela contem o Microfone que procuramos.

Não importa qual a URI que você me devolveu no `Location` , se ela fica no `localhost` , no Brasil, na China, na Coreia, em qualquer lugar da internet, eu vou seguir e verificarei se a representação do carrinho será devolvida, se o carrinho está sendo apontado por essa URI.

Todos esses status code servem para indicar ao cliente diversos tipos de situações que ocorrem ao se trabalhar com recursos remotos, desde o sucesso (200), até a criação (201), dados não encontrados (404), erro no servidor (500) e media type enviado não suportado (415). Vamos aos exercícios?