02

O protocolo HTTP por baixo dos panos

Nós já sabemos que na web o cliente se comunica com o servidor usando o protocolo HTTP. Então ele faz requisições do tipo GET para receber representações de um recurso, e ele faz uso do método POST para enviar uma representação e fazer algo no servidor, como criar um carrinho.

Mas como isso é feito por baixo dos panos? Como é que o HTTP funciona por trás desse código Java ou de nosso navegador?

No nosso terminal usaremos novamente o `curl` . Assim como fizemos antes, executar o`curl http://localhost:8080/carrinhos/1 (http://localhost:8080/carrinhos/1)' faz com que o programa execute uma requisição do tipo GET equivalente a digitar essa URI em nosso navegador. O resultado é o xml de nosso carrinho.

Legal, não tem segredo uma vez que ele fez o mesmo que o navegador. Mas o curl permite uma opção, a `-v` , que diz que queremos saber o que o cliente está enviando para o servidor e o que o servidor está devolvendo para o cliente: ele mostra tudo o que está sendo comunicado entre um lado e o outro, ele aumenta o nível de verbosidade do nosso código. Executamos então

```
curl -v http://localhost:8080/carrinhos/1
```

e temos toda a nossa comunicação entre cliente e servidor:

```
* Adding handle: conn: 0x7fad8b80ac00
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x7fad8b80ac00) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET /carrinhos/1 HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/xml
< Date: Thu, 24 Apr 2014 14:04:36 GMT
< Content-Length: 568
<
<br.com.alura.loja.modelo.Carrinho>
  <produtos>
    <br.com.alura.loja.modelo.Produto>
      <preco>4000.0</preco>
      <id>6237</id>
      <nome>Videogame 4</nome>
      <quantidade>1</quantidade>
    </br.com.alura.loja.modelo.Produto>
    <br.com.alura.loja.modelo.Produto>
      <preco>60.0</preco>
```

```
      <id>3467</id>
      <nome>Jogo de esporte</nome>
      <quantidade>2</quantidade>
    </br.com.alura.loja.modelo.Produto>
  </produtos>
  <rua>Rua Vergueiro 3185, 8 andar</rua>
  <cidade>S?o Paulo</cidade>
  <id>1</id>
* Connection #0 to host localhost left intact
</br.com.alura.loja.modelo.Carrinho>
```

As primeiras linhas com asterisco mostram que a conexão está sendo efetuada, ainda não conseguimos enviar nem receber nada. Já as linhas que começam com o símbolo de maior significam que o dado foi enviado para o servidor, enquanto as que começam com o símbolo de menor significam que foram recebidas do cliente.

Repare que logo de cara enviamos para o servidor um GET, dizendo o que estamos interessados em pegar e a versão do protocolo HTTP utilizada:

```
> GET /carrinhos/1 HTTP/1.1
```

Depois ele envia três cabeçalhos, note o padrão `Nome do cabeçalho: valor do cabeçalho` , sempre separando o nome do valor através de um `:` . Como sabemos que são só três cabeçalhos? Pois após enviar todos os cabeçalhos, um cliente HTTP envia sempre uma linha em branco:

```
> GET /carrinhos/1 HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
>
```

O primeiro cabeçalho diz que é o agente, quem é o programa cliente que está fazendo a requisição, poderia ser um Firefox, um Internet Explorer, um Chrome, Safari etc. No nosso caso é o curl:

```
> User-Agent: curl/7.30.0
```

Logo depois ele fala qual o servidor e porta que ele está tentando acessar:

```
> Host: localhost:8080
```

E ainda comenta que ele aceita como resultado qualquer tipo de media type e formato:

```
> Accept: */*
```

Foi isso que nosso cliente enviou, duas linhas obrigatórias (GET e Host) e dois cabeçalhos opcionais (User e Accept). O servidor respondeu, o que? Primeiro a linha mais importante, dizendo qual a versão do protocolo HTTP que ele está respondendo, o código de status da requisição e uma mensagem relativa a este status:

```
< HTTP/1.1 200 OK
```

No nosso caso o procolo é o 1.1, o resultado foi 200 e o significado deste 200 é OK. O status code 200 significa que a requisição foi processada com sucesso. Outro status code que conhecemos é o 404, NOt Found, quando não foi encontrando nenhum recurso naquela URI, um status code famoso na internet.

Logo depois ele diz qual o media type utilizado no retorno dos dados do servidor, como ele devolveu application/xml, ele entrega o cabeçalho de resposta:

```
< Content-Type: application/xml
```

O servidor entrega a data da resposta:

```
< Date: Thu, 24 Apr 2014 14:04:36 GMT
```

E também o tamanho da resposta, quantos bytes estão no corpo de nossa resposta:

```
< Content-Length: 568
```

Novamente uma linha em branco, para dizer que os cabeçalhos acabaram. Esses são todos os cabeçalhos que o servidor entregou. Agora que os cabeçalhos acabaram, surge o corpo da resposta que tem exatamente o número de bytes que ele avisou em seu cabeçalho:

```
<br.com.alura.loja.modelo.Carrinho>
  <produtos>
    <br.com.alura.loja.modelo.Produto>
      <preco>4000.0</preco>
      <id>6237</id>
      <nome>Videogame 4</nome>
      <quantidade>1</quantidade>
    </br.com.alura.loja.modelo.Produto>
    <br.com.alura.loja.modelo.Produto>
      <preco>60.0</preco>
      <id>3467</id>
      <nome>Jogo de esporte</nome>
      <quantidade>2</quantidade>
    </br.com.alura.loja.modelo.Produto>
  </produtos>
  <rua>Rua Vergueiro 3185, 8 andar</rua>
  <cidade>São Paulo</cidade>
  <id>1</id>
</br.com.alura.loja.modelo.Carrinho>
```

Então por baixo dos panos é isso que o HTTP faz. Toda vez que você faz um GET através do seu navegador, ele está se conectando com o servidor, enviando cabeçalhos similares aos que vimos, recebendo outros cabeçalhos e o corpo da resposta, provavelmente uma página html, um json, um xml, uma imagem etc.

Sempre que trabalhamos com HTTP estamos fazendo requisições como a exemplificada aqui e recebendo respostas similares. É isso que os programas (curl, servidor, navegadores) estão fazendo a todo instante: trocando representações através do protocolo HTTP.

Como vimos anteriormente, agora quero fazer um POST, como funciona ele? Executamos o mesmo POST que vimos no capítulo anterior, mas agora com a opção -v:

```
* Adding handle: conn: 0x7f989380ee00
* Adding handle: send: 0
* Adding handle: recv: 0
* Curl_addHandleToPipeline: length: 1
* - Conn 0 (0x7f989380ee00) send_pipe: 1, recv_pipe: 0
* About to connect() to localhost port 8080 (#0)
*   Trying ::1...
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 8080 (#0)
> POST /carrinhos HTTP/1.1
> User-Agent: curl/7.30.0
> Host: localhost:8080
> Accept: */*
> Content-Length: 362
> Content-Type: application/x-www-form-urlencoded
>
* upload completely sent off: 362 out of 362 bytes
< HTTP/1.1 200 OK
< Content-Type: application/xml
< Date: Thu, 24 Apr 2014 14:15:30 GMT
< Content-Length: 24
<
* Connection #0 to host localhost left intact
<status>sucesso</status>
```

Assim que a conexão foi estabelecida, temos que ao invés de GET fazemos um POST:

```
> POST /carrinhos HTTP/1.1
```

Os próximos três cabeçalhos são análogos aos que vimos no GET: o User-Agent, o Host e o Accepts. Mas ele, cliente, já falou: o conteúdo que estou enviando são 362 bytes. E ele falou, o conteúdo que estou enviando é do tipo `application/x-www-form-urlencoded`, que parece não fazer sentido certo? É algum errozinho que temos que corrigir em breve, mas o ponto importante é que quando enviamos dados via POST, o nosso corpo contem a mensagem a ser enviada, no nosso caso a representação do carrinho, que vem logo após o último cabeçalho da nossa requisição.

E na resposta temos os cabeçalhos já conhecidos, a versão do protocolo, o 200, o Content-Type, Date e Content-Length. O resultado é de sucesso. A diferença do GET para o POST até aqui é que o POST permite enviar qualquer quantidade de dados em seu corpo, seja texto puro ou dados binários.

Então por baixo do pano é assim que funciona o GET e o POST, ambos se comunicam com o servidor através de TCP, enviando cabeçalhos e corpo, recebendo cabeçalhos e corpo. Vamos para os exercícios?