09

Cliente do serviço

Transcrição

No vídeo anterior, nós escrevemos a renderização do XML. Porém, às vezes preferimos desenvolver o XML e o JSON simultaneamente, para que sejam enviados a depender do que o cliente pede.

Isso faz sentido quando levamos em consideração que existem clientes diferentes. O Angular, por exemplo, que é uma biblioteca JavaScript, provavelmente irá requisitar o JSON, e o Java dentro do Android talvez trabalhe com XML. Normalmente os dois trabalham com JSON, mas estamos só exemplificando.

A nossa motivação agora é criarmos um cliente Java que saiba enviar uma requisição HTTP e definir se a resposta será em JSON ou XML.

Para começar, no Eclipse, criaremos um novo projeto Java padrão (ao invés de um projeto web). Para isso, selecionaremos "File > New > Project", e em seguida "Java Project > Finish".

Na nova janela, nomearemos esse projeto como `cliente-webservice` e clicaremos em "Finish". Como não queremos criar um módulo (`module-info.java`), basta clicarmos em "Don't create".

Antes de criarmos uma classe no nosso projeto, precisamos de uma biblioteca que faça esse trabalho para nós. Na verdade, poderíamos utilizar as classes Java padrão para enviarmos uma requisição HTTP, mas já existem bibliotecas de mais alto nível que fazem isso por nós.

Nesse curso, usaremos a mais popular delas (apesar de já existir uma mais simples), chamada Java HttpClient 4.5 (http://hc.apache.org/httpclient-3.x/tutorial.html) - uma biblioteca da Apache que faz parte dos HTTP Components.

> Na página do Java HttpClient é possível encontrar tutoriais, guias, documentações e exemplos de utilização dessa biblioteca. Os arquivos que vamos precisar para essa aula podem ser baixados aqui (https://caelum-online-public.s3.amazonaws.com/1001-servlets-parte2/06/libs-httpclient.zip).

Será necessário extrair o conteúdo dessa pasta (uma pasta `lib` com 4 arquivos `.jar`) e copiá-lo para dentro do nosso `client-webservice`. Para fazermos com que o compilador enxergue o conteúdo desses `.jar`, selecionaremos os 4, clicaremos com o botão direito e em seguida em "Build > Build Path".

O Eclipse criará uma nova pasta chamada `Referenced Libraries` com o mesmo conteúdo da pasta `lib`. Agora poderemos criar nossa primeira classe, clicando com o botão direito na pasta `src` e em seguida em "New > Class".

Essa classe, chamada `ClienteWebService`, será criada no pacote `br.com.alura.cliente`. Também selecionaremos a caixa "public static void main(String[]args)" para criarmos automaticamente o método `main()`, e então clicaremos em "Finish".

```
package br.com.alura.cliente;

public class ClienteWebService {

    public static void main(String[] args) {

    }

}
```

COPIAR CÓDIGO

Para construirmos nosso código, começaremos com esse exemplo encontrado na página Quick Start (http://hc.apache.org/httpcomponents-client-4.5.x/quickstart.html) do HttpClient:

```
Request.Post("http://targethost/login")
.bodyForm(Form.form().add("username",  "vip").add("password",
.execute().returnContent();
```

COPIAR CÓDIGO

Tome cuidado na hora de importar! Devemos utilizar a interface fluente `org.apache.http.client.fluent.Request`, com a qual é possível definir a requisição através de várias chamadas de métodos.

Queremos enviar um `Post()` para a url `http://localhost:8080/gerenciador/empresas`, que é o endereço do nosso serviço. Como não queremos enviar nenhuma informação sobre formulários por enquanto, deletaremos essa parte do código. Isso fará com que ele pare de compilar, mas para consertarmos isso basta jogarmos uma exceção ("Add throws declaration"). Assim, teremos:

```
public class ClienteWebService {

    public static void main(String[] args) throws Exception {

        Request
        .Post("http://localhost:8080/gerenciador/empresas")
        .execute()
        .returnContent()
    }
```

```
}
```

COPIAR CÓDIGO

Ou seja, definimos que na requisição será enviado um `Post()`, que será executado e aguardará o conteúdo. Nós queremos que o conteúdo na resposta seja devolvido como uma string. Vamos escrever isso, incluindo um `System.out.println()` desse conteúdo:

```
String conteudo = Request
    .Post("http://localhost:8080/gerenciador/empresas")
    .execute()
    .returnContent()
    .asString();

System.out.println(conteudo);
```

COPIAR CÓDIGO

Esse é o código mais simples possível: temos um `request` que faz um `Post()` para o endereço `http://localhost:8080/gerenciador/empresas`, executa esse `Post()`, aguarda o conteúdo e nos devolve como uma string (isso porque esse conteúdo poderia ser, por exemplo, uma imagem).

Vamos inicializar o Tomcat e testar esse código. Para isso, clicaremos com o botão direito em qualquer ponto dentro do método `main()` e selecionaremos "Run as > Java Aplicattion". No console, teremos:

```
<list>
  <empresa>
    <id>1</id>
    <nome>Alura</nome>
    <dataAbertura>2018-11-01 16:53:16.719 UTC</dataAbertura>
  </empresa>
  <empresa>
```

```
    <id>2</id>
    <nome>Caelum</nome>
    <dataAbertura>2018-11-01 16:53:16.719 UTC</dataAbertura>
  </empresa>
</list>
```

COPIAR CÓDIGO

Ou seja, ele nos devolveu um XML. Perfeito! Nós conseguimos, a partir de um cliente Java, fazer uma requisição Java. Foi fácil, não?

Agora nós queremos que o servidor decida entre JSON e XML na resposta. Para tal, precisaremos enviar alguma informação para o servidor antes do método `execute()`. No caso, adicionaremos um cabeçalho (`addHeader()`). Dentro do protocolo HTTP, já existe um cabeçalho que prevê essa situação: o `accept`, cujo valor define o formato que nosso cliente aceita.

```
.addHeader("accept", "application/json")
```

COPIAR CÓDIGO

Agora esse cabeçalho será enviado junto com a requisição. O servidor lerá esse cabeçalho e devolverá um XML, ou, nesse caso, um JSON.

No entanto, se executarmos esse código, o servidor ainda não será capaz de interpretar esse cabeçalho. Resolveremos esse problema na classe `EmpresasService` do projeto `gerenciador`:

```
public class EmpresasService extends HttpServlet {
    private static final long serialVersionUID = 1L;

    protected void service(HttpServletRequest request, HttpSer

        List<Empresa> empresas = new Banco().getEmpresas();

        XStream xstream = new XStream();
```

```
            xstream.alias("empresa", Empresa.class);
            String xml = xstream.toXML(empresas);

            response.setContentType("application/xml");
            response.getWriter().print(xml);


            Gson gson = new Gson();
            String json = gson.toJson(empresas);

            response.setContentType("application/json");
            response.getWriter().print(json);

        }

    }
```

COPIAR CÓDIGO

A primeira coisa que precisamos fazer é ler o cabeçalho a partir da requisição. Para isso, usaremos o método `getHeader()`. Queremos que a requisição nos devolva o valor do cabeçalho `accept`:

```
String valor = request.getHeader("accept")
```

COPIAR CÓDIGO

Agora precisamos analisar se (`if()`) esse valor está associado a um XML ou a um JSON. Para isso, usaremos um truque: o método `endsWith()`, que irá diferenciar se o cliente mandou um cabeçalho que termina com `json` ou com `xml`:

```
String valor = request.getHeader("accept");

if(valor.endsWith("xml")) {
```

```
        XStream xstream = new XStream();
        xstream.alias("empresa", Empresa.class);
        String xml = xstream.toXML(empresas);

        response.setContentType("application/xml");
        response.getWriter().print(xml);

    } else if(valor.endsWith("json")) {
        Gson gson = new Gson();
        String json = gson.toJson(empresas);

        response.setContentType("application/json");
        response.getWriter().print(json);

    }
```

COPIAR CÓDIGO

Se o cabeçalho finaliza com `xml`, iremos renderizar todo o código associado à concatenação em XML, e a mesma coisa para o `json`.

Ao final, escreveremos mais um `else`: se o cliente não definiu nem `xml`, nem `json` no cabeçalho, devolveremos um `json` com a mensagem `no content`:

```
    } else {
        response.setContentType("application/json");
        response.getWriter().print("{'message':'no content'}");
    }
```

COPIAR CÓDIGO

Agora faz parte da especificação do nosso protocolo como o serviço irá tratar cada caso. Dessa forma, o cliente irá saber como enviar os dados e o que ele pode esperar.

> Dica: É possível melhorar esse código com padrões de projeto, e o Spring MVC faz isso de maneira muito simples.

Vamos reinicializar o servidor e executar nosso cliente novamente com o cabeçalho `application/xml` . No console, teremos:

```
<list>
  <empresa>
    <id>1</id>
    <nome>Alura</nome>
    <dataAbertura>2018-11-01 17:59:02.292 UTC</dataAbertura>
  </empresa>
  <empresa>
    <id>2</id>
    <nome>Caelum</nome>
    <dataAbertura>2018-11-01 17:59:02.292 UTC</dataAbertura>
  </empresa>
</list>
```

COPIAR CÓDIGO

Já quando testamos com o cabeçalho `application/json` :

```
[{"id":1,"nome":"Alura","dataAbertura":"Nov 1, 2018, 2:59:02 PM"},
{"id":2,"nome":"Caelum","dataAbertura":"Nov 1, 2018, 2:59:02 PM"}]
```

Ou seja, a lógica do nosso serviço funcionou. Podemos fazer mais um teste acessando a URL http://localhost:8080/gerenciador/empresas (http://localhost:8080/gerenciador/empresas), que nos retornará:

```
{'message':'no content'}
```

Isso porque o navegador não envia na requisição a especificação de um XML ou JSON. Mas será que é isso mesmo? Vamos analisar a página nas "Ferramentas do desenvolvedor", clicando em "Network > Headers" e enviando uma nova requisição. Em "Request Headers", teremos:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,
```

COPIAR CÓDIGO

Como podemos perceber, o `accept` no nosso cabeçalho deveria ter um A maiúsculo, e não devemos nos esquecer de corrigir isso. Além disso, repare que a requisiçao aceita `text/html` , `application/xhtml+xml` ou `application/xml` , mas nosso serviço não consegue avaliar corretamente todas essas opções, pois não o construímos dessa forma.

Vamos corrigir isso. Primeiro, vamos imprimir o valor desse cabeçalho para acompanharmos o trabalho do nosso serviço:

```
List<Empresa> empresas = new Banco().getEmpresas();

String valor = request.getHeader("Accept");

System.out.println(valor);
```

COPIAR CÓDIGO

Como ainda não corrigimos o problema, continuaremos recebendo a mensagem "no content" ao acessarmos a URL http://localhost:8080/gerenciador/empresas (http://localhost:8080/gerenciador/empresas). Porém, nosso console irá imprimir:

```
text/html,application/xhtml+xml,application/xml;q=0.9,image/wel
```

COPIAR CÓDIGO

É dessa forma que nosso serviço recebe a requisição do navegador. Portanto, precisaremos fazer uma análise mais eficiente desse valor, já que nosso

método `endsWith()` não está conseguindo ler a opção `application/xml` .

O certo seria nós realmente avaliarmos cada tipo de maneira minuciosa, mas uma forma simples de resolvermos o problema é o método `contains()` :

```
if(valor.contains("xml")) {

    XStream xstream = new XStream();
    xstream.alias("empresa", Empresa.class);
    String xml = xstream.toXML(empresas);

    response.setContentType("application/xml");
    response.getWriter().print(xml);
```

COPIAR CÓDIGO

Dessa vez, o navegador irá retornar:

```
This XML file does not appear to have any style information as
<list>
    <empresa>
        <id>1</id>
        <nome>Alura</nome>
        <dataAbertura>2018-11-01 18:15:04.842 UTC</dataAbertura
    </empresa>
    <empresa>
        <id>2</id>
        <nome>Caelum</nome>
        <dataAbertura>2018-11-01 18:15:04.842 UTC</dataAbertura
    </empresa>
</list>
```

COPIAR CÓDIGO

Ou seja, o serviço entendeu que o navegador requisitou um XML, e nosso cliente continua funcionando (tanto com JSON quanto com XML).

Pronto! Criamos um serviço que agora consegue devolver em dois formatos diferentes, e criamos nosso próprio cliente. Quem trabalha com Android precisará fazer esse tipo lógica, por exemplo, com Angular (para enviar as requisições através de JavaScript).

Esse foi um teste interessante para verificarmos como funciona a comunicação de um Web Service. Dê uma olhada nos exercícios e até o próximo vídeo!