09

Obtendo a Capa do FileSystem

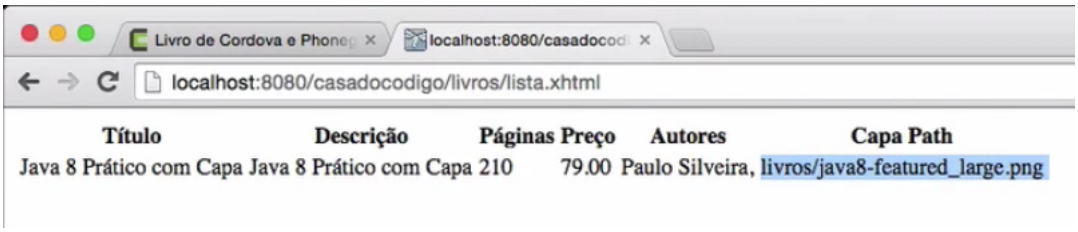
Transcrição

Uma vez que já estamos enviando o arquivo para o servidor, precisamos de uma forma de recuperar os dados do mesmo.

Acessaremos a página `lista.xhtml`, e em `<dataTable>`, adicionaremos mais uma coluna, esta será o *Path* da capa do Livro. Antes de , vamos inserir o seguinte código:

```
<h:column>
    <f:facet name="header">Capa Path</f:facet>
    #{livro.capaPath}"
</h:column>
```

Faça *Full Publish* da aplicação e veja como ficou a exibição da capa na tela.



Aparentemente a foto foi salva, mas na hora de exibir, não faz sentido exibir apenas o path relativo da foto. O ideal é exibir a própria foto. Mas acabamos caindo em uma limitação, pois estamos salvando a foto fora do *Application Server*, em uma pasta específica do S.O.. Então mesmo que usássemos o caminho relativo, ele não iria funcionar.

Para que o sistema pegue esse arquivo de dentro do S.O., precisaremos tratar isso dentro da aplicação. Faremos isto, utilizando `Servlet` normal da especificação de *Servets*. Em seguida, criaremos uma nova classe no pacote `br.com.casadocodigo.servlets` e daremos o nome da classe de `FileServlet`. Faça a nova classe herdar de `HttpServlet` e já sobrescrever o método `service()`. Todo `Servlet` precisa de um mapeamento, no nosso caso, faremos via *annotation* usando o `@WebServlet` e informando o caminho que desejamos mapear. No nosso caso, usaremos o mapeamento `/file`.

Um detalhe interessante é a forma como nosso servlet será chamado. Do jeito que fizemos até aqui, para chamar nosso *Servlet*, precisaremos passar algum parâmetro com o nome do arquivo que queremos carregar, por exemplo:

```
http://localhost:8080/casadocodigo/file?p=livros/java-8-featured_large.png
```

Mas desta forma, o carregamento ficará estranho . Seria melhor fazer:

```
.../file/livros/java-8-featured_large.png
```

Assim, ficaria parecendo que estamos acessando realmente o arquivo. Conseguiremos esse resultado simplesmente adicionado um `/*` (barra asterisco) no fim do mapeamento já feito. Assim a base de nosso *Servlet* ficará assim:

```
@WebServlet("/file/*")
public class FileServlet extends HttpServlet {

    @Override
    protected void service(HttpServletRequest req, HttpServletResponse res)
                throws ServletException, IOException {
        // código do tratamento do arquivo irá aqui!
    }

}
```

Dentro do método `service()`, conseguimos pegar o que vier pelo `*` por meio do objeto `req.getRequestURI()`. O método nos retorna toda a URL, mas só nos interessa o que vier depois de `/file`, então faremos um `split("/file")` e do Split teremos dois lados, valor `[0]` será o que vem antes do `/file` e `[1]` o que vem depois. Queremos este último, uma vez que é isso que temos salvo no banco de dados. Desta forma, já teremos o *path*.

```
String path = req.getRequestURI().split("/file")[1];
```

Para que o arquivo seja enviado no `response` do `Servlet`, precisaremos informar o `contentType` do arquivo. Neste caso específico, sabemos que se trata de uma imagem, porém queremos deixar o `FileServlet` mais genérico. Usaremos a API nova do **Java** de **NIO** para pegar o `contentType` direto do arquivo `Paths.get("caminho completo do arquivo")`. No entanto, temos que acessar de fato o arquivo e, até o momento, só temos o caminho relativo do arquivo. Onde foi que guardamos o arquivo dentro do servidor?

Podemos juntar o `path` que temos com o caminho fixo do nosso `FileSaver` para conseguirmos o caminho completo do arquivo. Passaremos as duas informações para o `Paths` e assim obtemos um `Path` como sendo a fonte do nosso arquivo.

```
Path source = Paths.get(FileSaver.SERVER_PATH + "/" + path);
```

O `Path` servirá como fonte para o `FileNameMap` conseguir chegar no arquivo e obter o `contentType`. O `FileNameMap` pode ser obtido usando a classe `URLConnection` chamando o método estático da classe `getFileNameMap`.

```
FileNameMap fileNameMap = URLConnection.getFileNameMap();
```

Pelo `fileNameMap`, chamaremos o método `getContentTypeFor()` passando nosso `source`. Só temos antes que informar que o protocolo de acesso é `file:` para que o `FileNameMap` possa pegar o arquivo corretamente. Assim sendo, temos:

```
String contentType = fileNameMap.getContentTypeFor("file:"+source);
```

Agora sim, poderemos setar o `contentType` no nosso `response`:

```
res.setContentType(contentType);
```

Mas parece que tivemos tanto trabalho apenas para setar um valor no `response`. Será que valeu a pena?

Todo navegador verifica sempre o *Header* da resposta do servidor para saber o que ele deve fazer. Quando você abre um PDF no `Chrome` por exemplo, ele possui um *leitor* interno de PDF's, e já é possível ao usuário ler o arquivo sem ter que abri-lo no seu computador na mão. Isso vale para outros tipos de arquivos, imagens, vídeos, e etc. Assim, é muito importante dizer ao navegador qual o tipo de conteúdo que estamos enviando para ele, e ele se ajustará a esse tipo de conteúdo.

Outro *Header* que é importante informar, é o tamanho do arquivo ou `Content-Length`, o que também ajuda o navegador a baixar corretamente o arquivo. Usaremos outra classe da API de **NIO** do **Java**, que é a classe `Files`.

```
res.setHeader("Content-Length", String.valueOf(Files.size(source)));
```

E por fim, o *Header* que coloca o nome correto do arquivo que estamos baixando, que é o `Content-Disposition`.

```
res.setHeader("Content-Disposition",
                "filename=\""+source.getFileName().toString() + "\"");
```

Ainda usaremos esse `Content-Disposition` mais adiante. Por enquanto, são esses os *Headers* que precisamos informar. Parece trabalhoso fazer tudo isso manualmente, mas não temos escolha... Até o momento, o **JavaEE** não possui nenhuma forma de obter esses dados automaticamente.

Usando **JSF** ainda temos mais uma coisa a fazer, que é limpar o `response` antes de setar qualquer cabeçalho. Lembre-se que o **JSF** pode ter recebido esse `request` e assim, já podemos ter colocado alguma informação no `response`, por isso é importante limpá-lo sempre que usarmos o `response`, evitando resultados inesperados.

```
res.reset();
// Antes de setar qualquer Header.
```

Agora estamos prontos. Mas ainda não transferimos o arquivo de fato, apenas preparamos o `response` para isso. Usaremos o `FileSaver` que já temos e ele cuidará da operação. Como queremos transferir o arquivo do servidor para o `response`, criaremos um método estático dentro de `FileSaver` chamado `transfer()`. Esse método tem a seguinte assinatura:

```
public static void transfer(Path source, OutputStream outputStream) {
    // código que manipula o arquivo
}
```

Já podemos chamá-lo no `FileServlet`, então vamos adicionar a chamada dele e logo depois veremos como implementar a transferência. Nosso método `service()` do *Servlet* ficou assim:

```
protected void service(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
    String path = req.getRequestURI().split("/file")[1];

    Path source = Paths.get(FileSaver.SERVER_PATH + "/" + path);
    FileNameMap fileNameMap = URLConnection.getFileNameMap();
    String contentType = fileNameMap.getContentTypeFor("file:"+source);

    res.reset();
```

```
    res.setContentType(contentType);
    res.setHeader("Content-Length", String.valueOf(Files.size(source)));
    res.setHeader("Content-Disposition",
            "filename=\""+source.getFileName().toString() + "\"");
    FileSaver.transfer(source, res.getOutputStream());
}
```

Agora, voltando ao nosso `FileSaver`, o primeiro passo da transferência é realizar a entrada do arquivo, do servidor para o sistema. Usaremos a classe `FileInputStream` para isso. Passaremos o `FileInputStream` para a classe `Channels.newChannel` que abre um canal direto com o arquivo, retornando o objeto `ReadableByteChannel`.

Além do canal de entrada, precisaremos de um canal de saída. Vamos usar o `Channels.newChannel` novamente e passaremos o `OutputStream` que recebemos como parâmetro. O canal nos retornará um `WritableByteChannel`.

Como os dois recursos `ReadableByteChannel` e `WritableByteChannel` são canais ligados direto ao arquivo e ao `response` do servidor, precisaremos fechar os recursos. Desde o **Java 7**, temos a possibilidade de usar o `try-with-resources` que automaticamente já fecha um recurso após o fim do `try`. O código ficará assim:

```
FileInputStream input = new FileInputStream(source.toFile());
try( ReadableByteChannel inputChannel = Channels.newChannel(input);
            WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
    // código que transfere o arquivo.
}
```

Além da entrada e saída, a transferência de um lado para o outro deve ser feita sempre usando um `Buffer`. Nesta, os arquivos serão transferidos em pedaços. No caso, vamos transferir 10kb por vez. Para criar nosso `Buffer` usaremos a classe `ByteBuffer.allocateDiret` passando como parâmetro `1024 * 10` que representa os `10Kb`.

```
public static void transfer(Path source, OutputStream outputStream) {
    try {
        FileInputStream input = new FileInputStream(source.toFile());
        try( ReadableByteChannel inputChannel = Channels.newChannel(input);
                WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
            ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);

        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Vamos começar a ler do nosso canal de entrada e transferir para o buffer. Faremos isto, enquanto existirem bytes para serem lidos. Usaremos um `while` para isso, e o próprio `inputChannel` possui um método chamado `read()` que nos retornará o valor `-1` quando não houver mais bytes a serem lidos.

```
while(inputChannel.read(buffer) != -1) {
    outputChannel.write(buffer);
    buffer.clear();
}
```

Depois de adicionarmos o `while()`, o trecho ficará da seguinte maneira:

```
public static void transfer(Path source, OutputStream outputStream) {
    try {
        FileInputStream input = new FileInputStream(source.toFile());
        try( ReadableByteChannel inputChannel = Channels.newChannel(input);
                  WritableByteChannel outputChannel = Channels.newChannel(outputStream)) {
              ByteBuffer buffer = ByteBuffer.allocateDirect(1024 * 10);

        while(inputChannel.read(buffer) != -1) {
            outputChannel.write(buffer);
            buffer.clear();
        }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    } catch (FileNotFoundException e) {
        throw new RuntimeException(e);
    }
}
```

Com os bytes no buffer, podemos enviar para o `outputChannel`, só que dessa vez não queremos ler, e sim escrever na saída. Faremos isto, usando o método `write()` e passando para ele o `buffer` com os bytes lidos. Depois que escrevermos no `buffer`, queremos que ele seja limpo usando o `buffer.clear()`. Desta forma, ele poderá voltar a ler mais informações do arquivo.