05

Criando métodos nas views e usando o contexto

Vamos abordar por último duas funcionalidades que ainda não vimos durante o curso, mas que podem ser bastante úteis. A primeira delas vamos usar para deixar o código das views um pouco mais legível e formatar melhor os dados das listas de usuários e produtos.

Vamos criar funções dentro de uma view. A lógica é bem semelhante ao que fizemos ao usar a view `main`, que é declarada em um arquivo e usada como se fosse uma view parcial, uma *tag*. Porém, declaramos tudo dentro da própria view onde usaremos a *tag*, tornando o escopo local, como um método privado. Comecemos refatorando a lista de usuários. Ao invés de fazer concatenação de métodos ao mostrar o número de acessos, podemos criar um método que faça isso.

Para utilizar código Scala puro dentro da view é necessário abrir um bloco precedido por um `@`.

```
@numeroDeAcessos(usuario: Usuario) = @{
  usuario.getAcessos().size()
}
```

Faremos o mesmo para a data do último acesso, que é ainda mais complicada. No caso, criaremos uma lógica que retorna a data somente se esta existir.

```
@ultimoAcesso(usuario: Usuario) = @{
    var ultimo = numeroDeAcessos(usuario) - 1
    if (ultimo >= 0) {
        usuario.getAcessos().get(ultimo).getData()
    }
}
```

Substituímos então as chamadas na tabela pelos novos métodos.

```
<td>@numeroDeAcessos(usuario)</td>
<td>@ultimoAcesso(usuario)</td>
```

Na listagem de produtos temos algo mais crítico. Um produto com preço de 0,999999999 centavos mostraria a dízima inteira, inclusive quebrando a formatação da tabela. Podemos formatar de modo que ele mostre somente as duas primeiras casas decimais. Bem semelhante ao `String.format()` de Java, porém em Scala ocorre um arredondamento ao invés de truncar o conteúdo. Já refatoramos na tabela também para utilizar a formatação adequada.

```
@formataEmReais(preco: Double) = @{
    String.format("R$%.2f", preco)
}

<td>@formataEmReais(produto.getPreco())</td>
```

E por fim veremos um modo e utilizar o contexto, objeto já visto antes diversas vezes, para acessar objetos em locais diversos do nosso projeto. No caso, vamos refatorar o cabeçalho da página para mostrar alguns links (de listagem de

usuários e produtos) somente caso o usuário seja administrador.

Como fazemos isso sem ter acesso ao usuário na nossa view? Temos acesso ao contexto na view, e podemos inserir um usuário dentro do contexto, como se fosse um cabide. Dentro do contexto temos um mapa, e podemos inserir o usuário ali. Faremos isso no momento da autenticação, em ambos `UsuarioAutenticado` e `AdminAutenticado`, respectivamente.

```
if (possivelUsuario.isPresent()) {
    Usuario usuario = possivelUsuario.get();
    context.args.put("usuario", usuario);
    return usuario.getEmail();
}
return null;
```

```
if (possivelUsuario.isPresent()) {
    Usuario usuario = possivelUsuario.get();
    // if (usuario.isAdministrador()) {
    context.args.put("usuario", usuario);
    return usuario.getEmail();
    // }
}
return null;
```

Agora, para acessar o contexto na view e recuperar o usuário do cabide, é mais complicado, portanto criaremos um método que servirá de atalho. Altere o arquivo `main.scala.html`. Como não sabemos que tipo de objeto vem do contexto, precisamos indicar que é um **Usuario**.

```
@usuario() = @{
    Http.Context.current().args.get("usuario").asInstanceOf[Usuario];
}
```

Com isso temos acesso a um usuário completo! Enfim, vamos substituir a barra de navegação anterior, utilizando o usuário para garantir que ele esteja autenticado e que é um administrador.

```
<ul class="nav navbar-nav navbar-right">
    @if(usuario() != null) {
        <li><a href="@routes.UsuarioController.painel">Painel</a></li>
        @if(usuario().isAdmin()) {
            <li><a href="@routes.AdminController.usuarios">Usuários</a></li>
            <li><a href="@routes.AdminController.produtos">Produtos</a></li>
        }
        <li><a href="@routes.UsuarioController.fazLogout">Logout</a></li>
    }
</ul>
```