01

Pesquisando por nome

Transcrição

Neste momento, nosso cliente solicitou a implementação de pesquisa na aplicação *Escolalura*! Porém, não se trata de apenas uma pesquisa, e sim duas, sendo que a primeira será por nome, ou seja, ao digitarmos o nome do aluno teremos uma lista correspondente. E a segunda é por nota - neste caso uma nota de corte é informada, e a listagem deve trazer todos os alunos aprovados com base nela.

Iniciaremos com a pesquisa por nome e, para isso, criaremos primeiro o cartão dessa nova página em nosso `index.html` :

```
<div class="col s12 m4 l4 waves-effect waves-light">
    <a href="/aluno/pesquisarnome">
        <div class="card-panel hoverable z-depth-1 center grey lighten-4">
            <i class="large material-icons">search</i>
            <div class="truncate">Pesquisar por nome</div>
        </div>
    </a>
</div>
```

Perceba que o endereço é `/aluno/pesquisarnome` , e para que a aplicação responda a essa nova rota, precisaremos defini-la no `AlunoController` .

```
@GetMapping("/aluno/pesquisarnome")
public String pesquisarNome() {
  return "aluno/pesquisarnome";
}
```

Como último passo para a página com o formulário de pesquisa, precisaremos criar o *template* correspondente. Lembrando que este deve estar em `resources/templates/aluno` e se chamar `pesquisarnome.html` . Neste arquivo teremos o seguinte código:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<meta charset="UTF-8" />
<link type="text/css" rel="stylesheet" href="../../materialize/css/materialize.min.css" media="
<link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet" />
<title>EscolAlura</title>
<script type="text/javascript" src="https://code.jquery.com/jquery-2.1.1.min.js"></script>
</head>
<body class="grey lighten-3">
  <div id="pesquisaAluno" class="container">
    <h3 class="main-title center">Pesquisar Aluno</h3>
    <form class="col s12" action="/aluno/pesquisar" method="get">
      <div class="row">
        <div class="row">
```

```
            <div class="input-field col s12">
              <input id="nome" name="nome" type="text" /> <label
                for="first_name">Nome</label>
            </div>
          </div>

          <div class="row">
            <div class="input-field col s12 center">
              <button class="btn waves-effect waves-light" type="submit">Pesquisar</button>
            </div>
          </div>
        </div>
      </form>
      <div th:if="${alunos} != null">
        <div class="row">
          <div class="input-field col s12">
            <table class="striped responsive-table">
              <thead>
                <tr>
                  <th style="text-align: center;">Nome</th>
                  <th style="text-align: center;">Dt. Nascimento</th>
                  <th style="text-align: center;">Curso</th>
                </tr>
              </thead>
              <tr th:each="aluno : ${alunos}">
                <td style="text-align: center;" th:text="${aluno.nome}"></td>
                <td style="text-align: center;"
                  th:text="${#dates.format(aluno.dataNascimento, 'dd/MM/yyyy')}"></td>
                <td style="text-align: center;" th:text="${aluno.curso.nome}"></td>
              </tr>
            </table>
          </div>
        </div>
      </div>
    </div>
    <!-- Fim do formulario de pesquisa -->

    <script type="text/javascript" src="../../materialize/js/materialize.min.js"></script>

  </body>
  </html>
```

Note que para fazer a requisição com o formulário, não estamos utilizando nenhuma *tag* especial do *Thymeleaf*, pois não enviamos nenhum objeto Java, somente texto comum. Criaremos o método que utilizará o repositório de alunos para buscar aqueles com determinado nome, respondendo a rota `/aluno/pesquisar`. O código inicial é algo como:

```
@GetMapping("/aluno/pesquisar")
public String pesquisar(@RequestParam("nome") String nome, Model model) {
  List<Aluno> alunos = repository.pesquisarPor(nome);
  model.addAttribute("alunos", alunos);
  return "aluno/pesquisarnome";
}
```

Como a busca pode retornar vários alunos, criamos uma lista. Algo que não vimos até aqui foi a anotação `@RequestParam`, uma forma diferente de conseguir capturar parâmetros. Assim, capturamos o parâmetro que não faz parte do caminho da *URL*.

O método `pesquisaPor` recebe um nome ainda não existente em nosso repositório. Vamos criá-lo!

```
public List<Aluno> pesquisaPor(String nome) {
  criarConexao();
  MongoCollection<Aluno> alunoCollection = this.bancoDeDados.getCollection("alunos", Aluno.class
  MongoCursor<Aluno> resultados = alunoCollection.find(Filters.eq("nome", nome), Aluno.class).it
  List<Aluno> alunos = new ArrayList<>();

  while(resultados.hasNext()) {
    alunos.add(resultados.next());
  }

  this.cliente.close();

  return alunos;
}
```

Lembrando que é necessário usar o método `criarConexao` para realizarmos as operações no banco de dados. Depois selecionaremos a coleção de alunos indicando a conversão dos documentos para objetos do tipo `Aluno`. Estamos filtrando a coleção com base no nome recebido como argumento, a partir do qual recuperamos um *iterador*. Ao fim, criaremos uma nova lista de alunos e a populamos com os resultados do cursor, fechando a conexão com o banco de dados. Ótimo, já podemos testar!

Antes de testarmos efetivamente, vamos fazer pequenas refatorações que ajudarão a simplificar o código: criar um método que popula uma lista de alunos dado um cursor de resultados - temos este código duplicado em algumas partes do repositório. O método será `popularAlunos`, com o código abaixo:

```
private List<Aluno> popularAlunos(MongoCursor<Aluno> resultados) {
  List<Aluno> alunos = new ArrayList<>();

  while(resultados.hasNext()) {
    alunos.add(resultados.next());
  }
  return alunos;
}
```

Feita esta pequena mudança, o método `pesquisaPor` fica um pouco mais curto e claro. Veja:

```
public List<Aluno> pesquisaPor(String nome) {
  criarConexao();
  MongoCollection<Aluno> alunosCollection = this.bancoDeDados.getCollection("alunos", Aluno.clas
  MongoCursor<Aluno> resultados = alunosCollection.find(Filters.eq("nome", nome), Aluno.class).i
  List<Aluno> alunos = popularAlunos(resultados);

  this.cliente.close();
```

```
    return alunos;
  }
```

Outra refatoração interessante é fecharmos a conexão com o banco de dados. Repare que, para abri-la, utilizamos o método `criarConexao`, chamando-o quando necessário, porém, para fecharmos a conexão teremos que usar `this.cliente.close`. Para deixar o código mais claro, pode-se usar apenas `fecharConexao`. Extraindo-se o fechar da conexão para um método separado, teremos:

```
  private void fecharConexao() {
    this.cliente.close();
  }
```

Veja novamente o método `pesquisarPor`; a alteração foi mínima, mas há mais clareza do que está acontecendo no método:

```
  public List<Aluno> pesquisarPor(String nome) {
    criarConexao();
    MongoCollection<Aluno> alunosCollection = this.bancoDeDados.getCollection("alunos", Aluno.cla
    MongoCursor<Aluno> resultados = alunosCollection.find(Filters.eq("nome", nome), Aluno.class).
    List<Aluno> alunos = popularAlunos(resultados);

    fecharConexao();
    return alunos;
  }
```

Com esses dois métodos nos outros pontos da classe também, deixamos a classe completa com a refatoração, assim:

```
  @Repository
  public class AlunoRepository {

    private MongoClient cliente;
    private MongoDatabase bancoDeDados;

    public void salvar(Aluno aluno){
      criarConexao();
      MongoCollection<Aluno> alunos = this.bancoDeDados.getCollection("alunos", Aluno.class);

      if(aluno.getId() == null){
        alunos.insertOne(aluno);
      }else{
        alunos.updateOne(Filters.eq("_id", aluno.getId()), new Document("$set", aluno));
      }
      fecharConexao();

    }

    private void criarConexao() {
      Codec<Document> codec = MongoClient.getDefaultCodecRegistry().get(Document.class);
      AlunoCodec alunoCodec = new AlunoCodec(codec);

      CodecRegistry registro = CodecRegistries.fromRegistries(
```

```
            MongoClient.getDefaultCodecRegistry(),
            CodecRegistries.fromCodecs(alunoCodec));

        MongoClientOptions options = MongoClientOptions.builder().codecRegistry(registro).build();

        cliente = new MongoClient("localhost:27017", options);
        bancoDeDados = cliente.getDatabase("test");
    }


    public List<Aluno> obterTodosAlunos(){
        criarConexao();

        MongoCollection<Aluno> alunos = this.bancoDeDados.getCollection("alunos", Aluno.class);
        MongoCursor<Aluno> resultados = alunos.find().iterator();

        List<Aluno> alunosEncontrados = popularAlunos(resultados);

        fecharConexao();
        return alunosEncontrados;
    }

    public Aluno obterAlunoPor(String id){
        criarConexao();

        MongoCollection<Aluno> alunos = this.bancoDeDados.getCollection("alunos", Aluno.class);
        Aluno aluno = alunos.find(Filters.eq("_id", new ObjectId(id))).first();

        fecharConexao();
        return aluno;
    }

    public List<Aluno> pesquisarPor(String nome) {
        criarConexao();

        MongoCollection<Aluno> alunosCollection = this.bancoDeDados.getCollection("alunos", Aluno.c
        MongoCursor<Aluno> resultados = alunosCollection.find(Filters.eq("nome", nome), Aluno.class
        List<Aluno> alunos = popularAlunos(resultados);

        fecharConexao();
        return alunos;
    }

    private void fecharConexao() {
        this.cliente.close();
    }

    private List<Aluno> popularAlunos(MongoCursor<Aluno> resultados) {
        List<Aluno> alunos = new ArrayList<>();

        while(resultados.hasNext()) {
            alunos.add(resultados.next());
        }
        return alunos;
    }

}
```

Ao pesquisarmos pela Julia ou por qualquer outro aluno teremos uma listagem logo abaixo ao formulário com os resultados, como ilustra a imagem abaixo: