06

Listando os alunos cadastrados

Transcrição

Com o cadastro de alunos funcionando, podemos criar a página de listagem dos mesmos. Primeiro adicionaremos um novo cartão na página inicial da aplicação indicando a página de listagem, que também conterá a pesquisa de alunos. No `index.html`, teremos:

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

Ainda não temos o caminho `/aluno/listar` pronto. Vamos criá-lo! O método `listar` na classe `AlunoController` deverá buscar a listagem de alunos no MongoDB e depois deixar essa listagem disponível para o template.

```
@GetMapping("/aluno/listar")
public String listar(Model model){
  List<Aluno> alunos = null;
  model.addAttribute("alunos", alunos);
  return "aluno/listar";
}
```

Lembre-se de que precisaremos do objeto `model` para enviar objetos para a *view* e, por ora, deixaremos a listagem como `null`, por não termos um método na classe `AlunoRepository` que nos retorne esta listagem. Este método se chamará `obterTodosAlunos`.

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

  MongoClient cliente = new MongoClient("localhost:27017", options);
  MongoDatabase bancoDeDados = cliente.getDatabase("test");
  MongoCollection<Aluno> alunos = bancoDeDados.getCollection("alunos", Aluno.class);

  MongoCursor<Aluno> resultado = alunos.find().iterator();

  List<Aluno> alunosEncontrados = new ArrayList<>();
  while(resultado.hasNext()){
```

```
      Aluno aluno = resultado.next();
      alunosEncontrados.add(aluno);
    }

    return alunosEncontrados;
  }
```

O trecho responsável por conectar ao banco está idêntico ao método `salvar`. A diferença é que utilizamos o método `find` seguido do `iterator` para obtermos um cursor a ser utilizado para se percorrer o resultado retornado da coleção. Depois, criaremos um `ArrayList` e utilizaremos o cursor para obter os alunos um a um, adicionando-os na listagem retornada ao fim do método.

Esse código já é funcional, porém, ele possui um problema! Temos muito código repetido nos métodos `salvar` e `obterTodosAlunos`. Toda a parte de conexão com o banco de dados está duplicada e caso precisemos alterar algo nesse procedimento, teríamos que fazê-lo em várias partes do código. Vamos refatorá-lo.

A primeira observação é que a conexão e o objeto cliente da conexão podem ser atributos da classe. Isso evita termos que recriar esses objetos sempre que precisarmos deles. Assim, teremos:

```
  private MongoClient cliente;
  private MongoDatabase bancoDeDados;
```

Agora podemos selecionar todo o código que cria a conexão com o banco de dados e extraí-lo para um método que chamaremos de `criarConexao`. Este método criará a conexão e atribuirá ao atributo da classe, sem precisar de nenhum retorno. Vejamos:

```
  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");
  }
```

Com essa pequena extração, veja como ficou muito mais simples o método `salvar`:

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

E o método `obterTodosAlunos`:

```
public List<Aluno> obterTodosAlunos(){
  criarConexao();
  MongoCollection<Aluno> alunos = this.bancoDeDados.getCollection("alunos", Aluno.class);

  MongoCursor<Aluno> resultado = alunos.find().iterator();

  List<Aluno> alunosEncontrados = new ArrayList<>();
  while(resultado.hasNext()){
    Aluno aluno = resultado.next();
    alunosEncontrados.add(aluno);
  }

  return alunosEncontrados;
}
```

Poderemos usar este método em nosso *controller* para listar os alunos! No método `listar` da classe `AlunoController` teremos:

```
@GetMapping("/aluno/listar")
public String listar(Model model){
  List<Aluno> alunos = repositorio.obterTodosAlunos();
  model.addAttribute("alunos", alunos);
  return "aluno/listar";
}
```

O último passo para testarmos se tudo funciona corretamente é criar o template `listar.html` na pasta `aluno` , em nossos `resources` . O HTML dessa página se encontra abaixo:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
<link type="text/css" rel="stylesheet"
  href="../materialize/css/materialize.min.css" media="screen,projection" />
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
  rel="stylesheet" />
<meta charset="UTF-8" />
<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="listaDeAlunos">
    <h3 class="main-title center">Lista de Alunos</h3>
    <div class="container">
      <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;">Ações</th>
          </tr>
        </thead>
        <tr th:each="aluno : ${alunos}">
          <td style="text-align: center;"><span th:text="${aluno.nome}"></span></td>
```

```
            <td style="text-align: center;"><span th:text="${#dates.format(aluno.dataNascimento,
            <td style="text-align: center;">
              <a th:href="@{'/aluno/visualizar/' + ${aluno.id}}" title="Visualizar">
                <i class="tiny material-icons ">perm_contact_calendar</i>
              </a>
              <a th:href="@{'/habilidade/salvar/' + ${aluno.id}}" title="Cadastrar habilidade">
                <i class="tiny material-icons ">verified_user </i>
              </a>
              <a th:href="@{'/nota/salvar/' + ${aluno.id}}" title="Notas">
                <i class="tiny material-icons ">spellcheck </i>
              </a>
            </td>

          </tr>
        </table>
      </div>
    </div><!-- Fim DIV listar alunos -->
    <script type="text/javascript" src="../materialize/js/materialize.min.js"></script>

  </body>
  </html>
```

Se abrirmos a aplicação e a listagem dos alunos, teremos um **erro**! No console é possível verificar a mensagem abaixo:

```
ReadBSONType can only be called when State is TYPE, not when State is VALUE.
```

O problema pode não estar muito claro, mas isso se trata da conversão dos documentos Mongo para objetos Java. Lembra que o nosso *codec* faz o *encode* mas não o *decode* desses objetos? Ficamos de implementar essa lógica depois, e agora faz sentido ela ser criada. Voltaremos ao método `decode` da classe `AlunoCodec`.

```
@Override
public Aluno decode(BsonReader reader, DecoderContext decoder) {
  return null;
}
```

O primeiro passo para decodificar o aluno é fazer isto com o documento. Lembra do *codec* de documento que recebemos via construtor? Utilizaremos ele para isso. O segundo passo é popular um objeto aluno com os dados do documento decodificado. Confuso? Vejamos o código!

```
@Override
public Aluno decode(BsonReader reader, DecoderContext decoder) {
  Document document = codec.decode(reader, decoder);
  Aluno aluno = new Aluno();

  aluno.setId(document.getObjectId("_id"));
  aluno.setNome(document.getString("nome"));
  aluno.setDataNascimento(document.getDate("data_nascimento"));
  Document curso = (Document) document.get("curso");

  if(curso != null){
    String nomeCurso = curso.getString("nome");
    aluno.setCurso(new Curso(nomeCurso));
```

```
    }

    return aluno;
  }
```

É um processo simples no qual só precisaremos estar atentos ao curso. Neste caso precisaremos de um *casting*, pois o método `get` retorna um `Object`, e não um `Document`. Diante disso verificaremos se o mesmo não é nulo, e depois capturamos o nome do curso e o atribuímos ao aluno como um novo objeto `Curso`. Outro ponto é que a classe `Curso` não tem um construtor em que é possível passar o nome do curso como argumento. Precisaremos criá-lo também.

```
public class Curso {
  private String nome;

  public Curso(String nome) {
    this.nome = nome;
  }

  // código omitido
}
```

Com o decodificador pronto, podemos testar nossa aplicação! A listagem deverá estar semelhante a imagem abaixo: