34. Listando todas categorias com DTO

Vamos incluir um **EndPoint** que **retorna todas** as **categorias**.

Em **CategoriaResource** vamos **reaproveitar** o método **find( )** para criar o novo EndPoint e chamamos o novo método de **findAll( )**.

Para **exibir todas as categorias** não vamos precisar do ID da categoria na URL basta solicitar /categorias, logo o parâmetro (value = "/{id}") é removido do novo método O parametro ID também é removido. O método deve retornar uma lista de categorias.

TipoRetorno: responseEntity<List<Categoria>>
Corpo: list<Categoria> list = categoriaService.findAll();
Retorno: return responseEntity.ok().body(list);

Em **CategoriaService** também **implementamos** um método chamado **findAll( )** que deve retornar uma lista de categorias.

TipoRetorno: List<Categoria>
Retorno: return categoriaRepository.findAll();

Observamos problemas ao solicitar o URL /categoria:

Categorias são listadas com seus produtos.
Um produto pode estar associado a mais de uma categoria.

Para listar apenas as categorias sem seus produtos usamos o padrão **DTO** (Data Transfer Object). DTO é um **objeto** que **contém apenas** os **dados necessarios** para determinada **operação** do sistema.

DTO: http://www.mauda.com.br/?p=1788

Resolvendo o problema…

Criar um **package** chamado **dto** e dentro do package uma **classe** chamada **CategoriaDTO**. Esse objeto definirá os dados que devem trafegar ao realizar operações básicas com Categoria.

A **Categoria** possui ID, nome e uma lista de produtos e com base nisso na classe **CategoriaDTO** devem ser **criados dois atributos**, um deles **Integer id** e outro **String nome**. Além dos atributos cria-se um **método construtor padrão**, **getters e setters** para estes atributos e por fim um **construtor que receba** um **objeto** do tipo **Categoria** para que o possamos **atribuir valores** ao **objeto CategoriaDTO**.

```
public CategoriaDTO(Categoria categoria) {
      this.id = categoria.getId();
      this.nome = categoria.getNome();
   }
```

Feito isso, vamos para CategoriaResource…

O método **findAll( )** agora deve **retornar CategoriaDTO** ao **invés** de **Categoria**, logo:

TipoRetorno: ResponseEntity<List<CategoriaDTO>>
Corpo: List<CategoriaDTO> listDTO = list.stream( ).
map(obj → new CategoriaDTO(obj)).
collect(Collectors.toList());
Retorno: return responseEntity.ok().body(listDTO);

Streams API:
- oferece a possibilidade de t**rabalhar com conjuntos de elemtos de forma mais simples**.
- incorporação do p**aradigma funcional** combinado as **expressões lambda**.
- **reduzir** a **preocupação** do **programador** com a forma de **implementar** o **controle de fluxo** ao lidar com **coleções**.
- **iterar** sob estas **coleções** e, a **cada elemento**, **realizar** uma **ação** (filtro, map, transformaçao, etc).

Exemplo:

```
// lista de strings definida.
static List<String> items = new ArrayList<String>();

public static void main(String[] args) {
    // itens adicionados a lista de strings.
    items.add("um");
    items.add("dois");
    items.add("tres");

    // obtendo um stream de strings.
    // pode-se usar items.parallelStream().
    Stream<String> stream =  items.stream();

    // imprimindo a stream de strings.
    System.out.println(stream.collect(Collectors.toList()));
}
```

Exemplo:

```
// criando uma lista de pessoas.
List<Pessoa> pessoas = fillPessoas();

// obtendo stream, iterando sobre a stream e imprimindo nome das pessoas.
pessoas.stream().forEach(pessoa -> System.out.println(pessoa.getNome()));

// obtendo stream, iterando sobre a stream e aplicando filtro sobre as pessoas.
Stream<Pessoa> stream = pessoas.stream().filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"));

// obtendo strem, iterando sobre a stream e calculando a média de idade.
double media = pessoas.stream().filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))
        .flatMapToInt(pessoa -> pessoa.getIdade()).average().getAsDouble();
```

fonte: https://www.devmedia.com.br/java-streams-api-manipulando-colecoes-de-forma-eficiente/37630