Spring Boot com Angular - Ionic

Capítulo: Operações de CRUD e Casos de Uso

Objetivo geral:

Implementar operações de CRUD e de casos de uso conforme boas práticas de Engenharia de Software.

Competências:

- Implementar requisições POST, GET, PUT e DELETE para inserir, obter, atualizar e deletar entidades, respectivamente, seguindo boas práticas REST e de desenvolvimento em camadas.
- Trabalhar com DTO (Data Transfer Object)
- Trabalhar com paginação de dados
- Trabalhar com validação de dados com Bean Validation (javax.validation)
- Criar validações customizadas
- Fazer tratamento adequado de exceções (incluindo integridade referencial e validação)
- Efetuar consultas personalizadas ao banco de dados

Nota: ressaltamos novamente que CRUD's também são casos de uso, mas estamos chamando de casos de uso os usos do sistema correspondentes a processos de negócio que não se enquadram em CRUD's comuns.

Inserindo novo Cliente com POST

```
{
	"nome" : "João da Silva",
	"email" : "joao@gmail.com",
	"cpfOuCnpj" : "39044683756",
	"tipo" : 1,
	"telefone1" : "997723874",
	"telefone2" : "32547698",
	"logradouro" : "Rua das Acácias",
	"numero" : "345",
	"complemento" : "Apto 302",
	"cep" : "38746928",
	"cidadeId" : 2
}
```

Deletando uma Categoria com DELETE

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

Na classe CategoriaService:

```
repo.deleteById(id);
```

Paginação com parâmetros opcionais na requisição:

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

Na classe CategoriaService:

```
PageRequest pageRequest = PageRequest.of(page, linesPerPage, Direction.valueOf(direction),
orderBy);
```

Validação sintática com Bean Validation:

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

Na classe CategoriaDTO:

```
import javax.validation.constraints.NotEmpty;
```

PUT, DELETE e GET para Cliente:

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

Na classe ClienteDTO:

```
import javax.validation.constraints.Email;
import javax.validation.constraints.NotEmpty;
```

Inserindo um novo Cliente com POST:

CORREÇÃO

Na classe ClienteService:

- Sugestão: usar @Transactional no método insert
- Não é necessário usar cidadeRepository no método fromDTO

Validacao customizada de CPF e CNPJ na insercao de Cliente

Checklist:

- 🕐 Criar a anotação customizada
- 🕐 Criar o Valitator personalizado para esta anotação e para o nosso DTO 🕐
- Programar o Validator, fazendo testes e inserindo as mensagens de erro 🕐
- Anotar nosso DTO com a nova anotação criada

Anotação:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import  java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Constraint(validatedBy  =  NomeValidator.class)
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface Nome {

    String message() default "Erro de validação";

    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Validator:

```
import java.util.ArrayList;
import java.util.List;

import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class NomeValidator implements ConstraintValidator<Nome, Tipo> {

    @Override
```

```
    public void initialize(Nome ann) {
    }

    @Override
    public boolean isValid(Tipo objDto, ConstraintValidatorContext context) {

        List<FieldMessage> list = new ArrayList<>();

        // inclua os testes aqui, inserindo erros na lista

        for (FieldMessage e : list) {
            context.disableDefaultConstraintViolation();
            context.buildConstraintViolationWithTemplate(e.getMessage())
            .addPropertyNode(e.getFieldName()).addConstraintViolation();
        }
        return list.isEmpty();
    }
}
```

Apresentando o caso de uso

Registrar Pedido

Atores	Cliente
Interessados	Departamento de vendas
Precondições	Cliente cadastrado
Pós-condições	-
Visão geral	Este caso de uso consiste no processo de escolha de produtos e fechamento de pedido por parte do cliente.

Cenário Principal de Sucesso

1. [OUT] O sistema informa os nomes de todas categorias ordenadamente.
2. [IN] O cliente informa um trecho de nome de produto desejado, e seleciona as categorias desejadas.
3. [OUT] O sistema informa nome e preço dos produtos que se enquadram na pesquisa.
4. [IN] O cliente seleciona um produto para adicionar ao carrinho de compras (*).
5. [OUT] O sistema exibe o carrinho de compras (**).
6. [IN] O cliente informa que deseja fechar o pedido, e informa seu usuário e senha.
7. [OUT] O sistema informa logradouro, numero, complemento, bairro, cep, cidade e estado de todos endereços do cliente.
8. [IN] O cliente seleciona um endereço para entrega.
9. [OUT] O sistema exibe as formas de pagamento.
10. O cliente escolhe uma das opções:
    10.1. Variante: Pagamento com boleto
    10.2. Variante: Pagamento com cartão
11. [OUT] O sistema informa a confirmação do pedido (***).

Cenários Alternativos: Variantes

Variante 5.1: Nova busca
5.1.1 [IN] O cliente informa que deseja realizar uma nova busca.
5.1.2 Vai para 1.

Variante 10.1: Pagamento com boleto
10.1.1. [IN] O cliente informa que deseja pagar com boleto.

Variante 10.2: Pagamento com cartão
10.2.1. [IN] O cliente informa que deseja pagar com cartão e informa a quantidade de parcelas.

Cenários Alternativos: Exceções

Variante 6.1: Falha na autenticação
6.1.1 [IN] O sistema informa mensagem de usuário ou senha inválidos.
6.1.2 Vai para 6.

Informações complementares

(*) Quando um produto já existente no carrinho é selecionado, a quantidade deste produto no carrinho deve ser incrementada, caso contrário o produto é adicionado ao carrinho com quantidade 1.

(**) As informações do carrinho de compras são: nome, quantidade e preço unitário de cada produto (não será dado desconto), o subtotal de cada item do carrinho, e o valor total do carrinho.

(***) As informações da confirmação do pedido são: número, data e horário do pedido, valor total do pedido, bem como o tipo e estado do pagamento (Pendente). Caso o pagamento seja com boleto, informar a data de vencimento, e caso o pagamento seja com cartão, informar o número de parcelas.

Operações identificadas para o backend a partir do caso de uso:

Passos 2 e 3:

Classe ProdutoService:

Parâmetros:

nome: um trecho de nome de produto
ids: uma lista de códigos de categorias

Retorno:

A listagem de produtos que contém o trecho de nome dado e que pertencem a pelo menos uma das categorias dadas

```
public Page<Produto> search(String nome, List<Integer> ids)
```

Passos 10 e 11:

Classe PedidoService:

Parâmetros:

pedido: um novo pedido a ser inserido na base de dados

Retorno:

A instância monitorada do pedido inserido

```
public Pedido insert(Pedido obj)
```

Nivelamento sobre SQL e JPQL

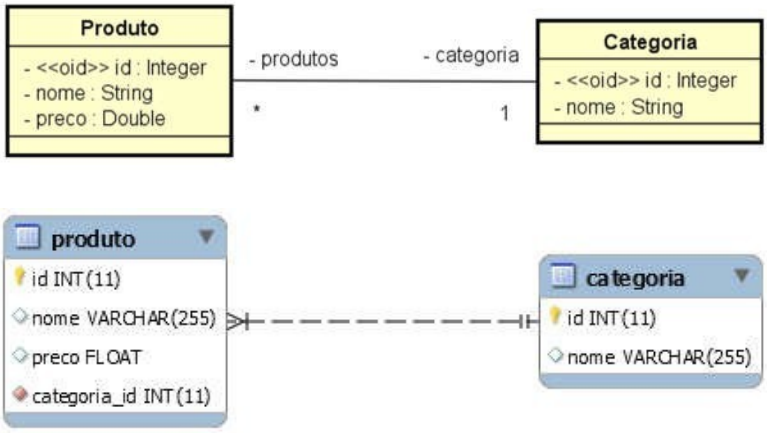
JPQL:

- Linguagem de consulta da JPA
- Similar à SQL, porém as consultas são expressas em "nível" de objetos
- É obrigatória a atribuição de um "alias" (apelido) aos objetos pretendidos na consulta:

`SELECT * FROM CLIENTE` *(retorna um resultset com os dados da tabela CLIENTE)*

`SELECT obj FROM Cliente obj` *(retorna um List<Cliente>)*

SUPONHA O SEGUINTE MODELO:



CONSULTA SIMPLES:

Recuperar todos produtos cuja categoria possui id = 1

SQL:

```
SELECT * FROM PRODUTO WHERE CATEGORIA_ID = 1
```

JPQL:

```
SELECT obj FROM Produto obj WHERE obj.categoria.id = 1
```

JUNÇÃO SIMPLES:

Recuperar todos produtos cuja categoria possui nome 'Informática'

SQL:

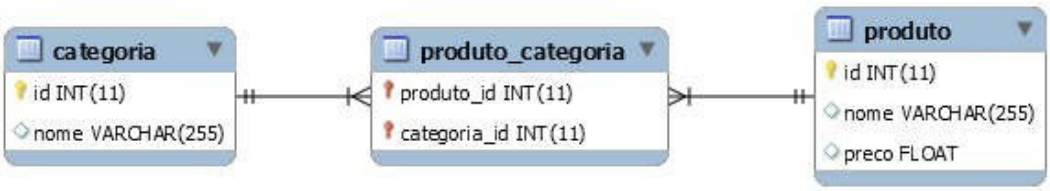
```
SELECT PRODUTO.*
FROM PRODUTO, CATEGORIA
WHERE
       PRODUTO.CATEGORIA_ID  =  CATEGORIA.ID  AND
       CATEGORIA.NOME = 'Informática'
```

```
SELECT PRODUTO.*
FROM PRODUTO
INNER JOIN CATEGORIA cat
   ON PRODUTO.CATEGORIA_ID = cat.ID
WHERE
  cat.NOME = 'Informática'
```

JPQL:

```
SELECT obj FROM Produto obj WHERE obj.categoria.nome = 'Informática'
```

EXEMPLO MAIS COMPLEXO. SUPONHA O SEGUINTE MODELO:



Recuperar todos produtos cujo nome contenha um dado string, e que pertença a pelo menos uma categoria dentre as categorias de uma data lista

SQL:

```
SELECT DISTINCT *
    FROM PRODUTO
    INNER JOIN
        PRODUTO_CATEGORIA cat1
            ON PRODUTO.ID = cat1.PRODUTO_ID
    INNER JOIN
        CATEGORIA cat2
            ON cat1.CATEGORIA_ID = cat2.ID
    WHERE
        PRODUTO.NOME LIKE ? AND
        cat2_.ID IN (?, ?)
```

Será substituído por um String

Serão substituídos por números inteiros

JPQL:

```
SELECT DISTINCT obj
    FROM Produto obj
    INNER JOIN
        obj.categorias cat
    WHERE
        obj.nome LIKE %:nome% AND
        cat IN :categorias
```

Será substituído por um String

Será substituído por um List<Categoria>

Busca de produtos por nome e categorias

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

Na classe ProdutoService:

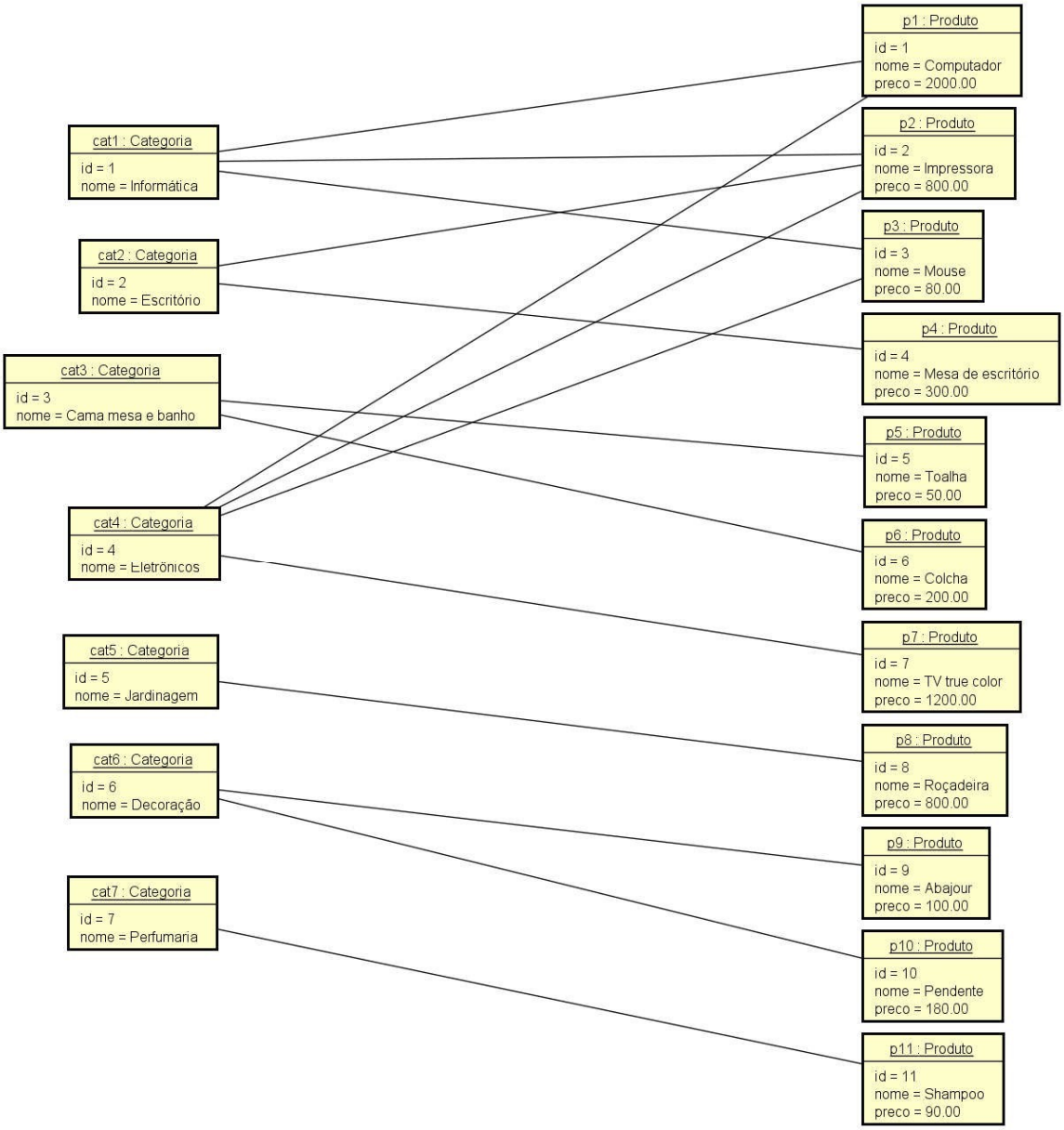
```
List<Categoria> categorias = categoriaRepository.findAllById(ids);
```

```
@Repository
@Transactional(readOnly=true)
public interface ProdutoRepository extends JpaRepository<Produto, Integer> {

	// https://docs.spring.io/spring-data/jpa/docs/current/reference/html/

	@Query("SELECT DISTINCT obj FROM Produto obj INNER JOIN obj.categorias cat WHERE
obj.nome LIKE %:nome% AND cat IN :categorias")
	Page<Produto> findDistinctByNomeContainingAndCategoriasIn(
			@Param("nome") String nome,
			@Param("categorias")  List<Categoria>  categorias,
			Pageable pageRequest);

}
```



Inserindo pedido

ATUALIZAÇÃO

Se você criou o projeto usando Spring Boot versão 2.x.x:

-

- Sugestão: usar @Transactional no método insert (esqueci de colocar no projeto)
- Na classe PedidoService, usar ProdutoService ao invés de ProdutoRepository

-

Checklist para permitir a instanciação de subclasses a partir de dados JSON:

1) Na superclasse abstrata, defina que haverá um campo adicional @type:

```
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "@type")
public abstract class Pagamento implements Serializable {
```

2) Em cada subclasse, defina qual será o valor do campo adicional para ela:

```
@JsonTypeName("pagamentoComBoleto")
public class PagamentoComBoleto extends Pagamento {

@JsonTypeName("pagamentoComCartao")
public class PagamentoComCartao extends Pagamento {
```

3) Crie uma classe de configuração com um método @Bean para registrar as subclasses:

```
@Configuration
public class JacksonConfig {

	//  https://stackoverflow.com/questions/41452598/overcome-can-not-construct-instance-of-interfaceclass-without-hinting-the-pare
	@Bean
	public Jackson2ObjectMapperBuilder objectMapperBuilder() {
		Jackson2ObjectMapperBuilder builder = new Jackson2ObjectMapperBuilder() {
			public void configure(ObjectMapper objectMapper) {
				objectMapper.registerSubtypes(PagamentoComCartao.class);
				objectMapper.registerSubtypes(PagamentoComBoleto.class);
				super.configure(objectMapper);
			};
		};
		return builder;
	}
}
```

4) Quando for enviar um objeto do tipo da superclasse (Pagamento), inclua o campo adicional para indicar qual das subclasses deve ser instanciada:

```
{
    "cliente" : {"id" : 1},
    "enderecoDeEntrega" : {"id" : 1},
```

```
    "pagamento" : {
         "numeroDeParcelas" : 10,
         "@type": "pagamentoComCartao"
    },
    "itens" : [
        {
            "quantidade" : 2,
            "produto" : {"id" : 3}
        },
        {
            "quantidade" : 1,
            "produto" : {"id" : 1}
        }
    ]
}
```