Java JPA: persistência de objetos Java para tabelas de forma automática.

Tudo sobre JPA: analisar documento “***Mapeamento objeto relaciona JPA.doc***”

Springboot já tem embutido o Tomcat, que é o servidor da aplicação (pra rodar a aplicação, que é o container).

Como modificar a porta do localhost, caso a porta 8080 esteja já sendo utilizada?

Finalizar aplicação – src/main/resources – application.properties – escrever: server.port=${port:8081} salvar e rodar de novo a aplicação.

Inicialmente já tem toda configuração como dependências (pom.xml).

1 commit

Controladores REST: são os recursos (resources)

Padrão REST: colocar os verbos corretamente sobre as requests. **(estudar endpoint REST)**

hashCode e equals: para que sejam verificados se os objetos são iguais pelos conteúdos e não pelos ponteiros de memória

Classe Serializable: declaração na classe para que ela implemente a interface Serializable objetos podem ser convertidos para sequência de bytes para poderem serem gravados em arquivos ou usados na web.

Quando testamos no Postman os dados já saem em JSON, o próprio Springboot faz a conversão de dados para JSON com os mesmos nomes das Classes.]

OBS: Interfaces não podem ser instanciadas, portanto escolhemos uma classe que implementa a interface.

Ex: List<Categoria> lista = new ArrayList<>( );

2 commit

Dependências a instalar: Banco de dados H2 (relacional) ele é em memória e open source.

JPA padrão Java...

Devtools: utilizar para testar o banco de dados

Por padrão, no pom.xml temos as dependências padrão do Springboot: starter web (escolhemos no início do projeto)

3 commit



Camada de domínio: classes de domínio, nada mais são que as Entidades.

Camada de serviço: oferece consultas para os controladores REST’s. Não tem contato com nada, apenas regras de negócio...algo que não foi implementado nas classes de domínio.

Camada de acesso a dados: os Repository (ou DAO), conversar com o banco de dados (consultar SQL, tudo que for banco), e a camada de serviço utiliza a camada de acesso a dados para realizar regras de negócios.

Controladores REST: *resources*.

No projeto:

Camada de domínio -> Categoria.java

Camada de serviço -> CategoriaService.java

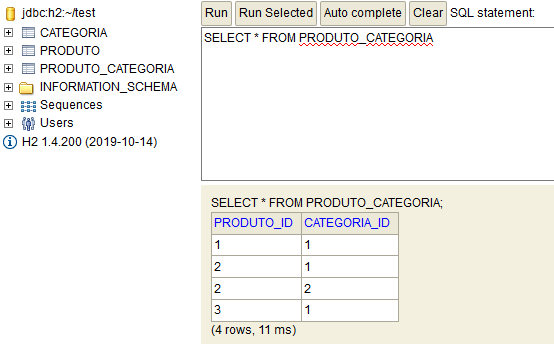
Camada de acesso a dados -> CategoriaRepository.java

Controlador REST -> CategoriaResouce.java

4 commit: criou operação de instanciação

Criando Produto: a associação está sendo feita de para muitos, no diagrama está escrito “produtos”, portanto a nível de projeto devemos manter o nome da lista como “produtos”;

Neste relacionamento entre produto e categoria, como é relação N-N, banco de dados relacional, é criado uma terceira tabela com os id’s dos dois. Criada tabelas!



@JsonManagedReference é colocado na Classe onde quer que mostre os objetos que estão associados a ela, ex: foi colocado em Categoria, portanto mostrará os produtos associados. Do outro lado (Produto), colocar @JsonBackReference

Criadas exceções para erro de categoria não existente, pacote .exception e acertadas classes produto e categoria.

Próximo commit: trabalhado Estado e Cidade, criadas entidades, e dentro da Cidade feita relação N-1 com criação da chave estrangeira. No programa principal feitas instâncias. O “save” serve para salvar os dados no banco...

Quando criamos ENUM no Springboot, o JPA automaticamente pode criar os códigos no banco de duas formas: String com o nome dos ENUMS ou ainda criar Int, começando com 0. Porém qualquer um que adicionar um novo ENUM em cima do primeiro, os códigos começarão do 0 de novo...problema!

Criamos o tipo enumerado TipoCliente: dentro dele fizemos um tipo mais sofisticado para controla-lo, com método estático (pode ser processado sem a instanciação de objetos) em que caso o código seja nulo, retorna nulo...é percorrido um for e se é passado algum código que existe, ele retorna o enum que foi passado, senão retorna um “throw” em uma exceção.

Segundo Nelio, dependendo da implementação e a linguagem, algumas classes não precisam ser criadas, caso do Telefone (uma classe “fraca”, com apenas um atributo), uma decisão que pode ser feita pelo próprio programador. Portanto é feita uma coleção de Strings associadas ao Cliente (de acordo com o diagrama).

Feito um macete para armazenar internamente um número inteiro e exposto para a aplicação o TipoCliente. No get e set colocado o método criado.

De acordo com o diagrama, apenas o Endereço conhece Cidade.

No application feitas as associações e criados clientes e endereços respectivos. Criados repositories e dado “@Autowired” na classe da aplicação. A partir dos repositories podemos salvar os dados.

Feito a cópia do CategoriaResource e dado CTRL F5 para modificar para Cliente, assim como no Service.

De acordo com o diagrama, feita proteção cíclica JSON na classe Cliente e na classe endereço colocada notação @back. Como o endereço conhece a cidade e é uma associação direcionada, não precisa ser feito nada na cidade. Porém há uma relação cíclica entre Cidade e Estado, precisa proteger também.

OBS: EM JSON O {} é LISTA!!

Criando Pedido, EstadoPagamento, Pagamento. Uma alteração no Cliente foi feito pela sua relação. Na hora de criar o Hash Equals, PagamentoComCartao e ComBoleto não precisa, pois o ‘id’ está na ‘mãe’. Para Serializable, colocar apenas o numero da versão nas subclasses (serialVersionUID = 1L)

Quando começamos a fazer o mapeamento Objeto-relacional em Pagamento e Pedido, queremos que ambos possuam o mesmo id, pois a relação é 1-1. Então, não é colocado o **GeneratedValue**. Portanto colocado *@MapsId* //para garantir que o id do pedido seja o mesmo do Pagamento

OBS: Lá em pedido colocado: *@OneToOne*(cascade=*CascadeType*.***ALL***, mappedBy="pedido")Pois é uma peculiaridade do JPA!

Para fazer o mapeamento quando se tem subclasses (mapeamento de herança) podemos fazer uma única tabela com ambas subclasses, e quando estanciar uma, colocar null na outra e vice-versa, ou uma tabela pra cada, como se fosse no diagrama de Classe. Normalmente quando tem poucos atributos em cada classe, faz uma tabela.

Quando colocamos *@Inheritance*(strategy=*InheritanceType*.***JOINED***) é só colocar o @Entity nas subclasses...assim estamos fazendo as duas tabelas. Lembrando que o ID vem da SuperClasse!

OBS: na tabela de pagamento, a chave primária é pedido\_id, pois a relação é 1-1 e dentro da Classe pedido fizemos o “join table” com a chave estrangeira do pedido.

Na hora de instanciar o pedido, observou-se que mesmo tendo a relação entre pedido e pagamento é necessário retirar do construtor do pedido o pagamento, para que possa ser instanciado um após o outro.

OBS: apenas criar uma repository para superclasse, as subclasses não precisam!!!

Criado ItemPedidoPK pois nesta classe possui Pedido pedido e Produto produto (é uma chave composta que aponta para Pedido e Produto, conforme diagrama), eles são a referência. Eles que identificam o ItemPedido. Feito HashEquals dos dois atributos pois a comparação deve ser com os dois msm, pra saber se é igual deve ser o mesmo pedido e o mesmo produto!!

Criamos o ItemPedidoPK, em JPA quando temos (um atributo composto - em Java seria um tipo primitivo) um atributo sendo uma Classe (no caso o id do ItemPedidoPK dentro da classe ItemPedido, colocar no ItemPedidoPK @embeddable

Criado PedidoService pela cópia de outro Service.

Na criação do endpoint do Pedido, de acordo com o diagrama, temos uma relação bidirecional, o endpoint do Pedido deve mostrar o Cliente, mas o Cliente não deve mostrar o Pedido (falando a respeito de Serialização)... Para isso: Pedido colocamos o @JsonManaged em cima do cliente e no Cliente colocamos o @Back em cima da lista de pedidos.

Pedido e Endereço é uma mão única, não precisa da notação, Pedido e Pagamento é a mesma coisa do Cliente. Pedido com itemPedido é uma chave composta, portanto, não será Serializado, temos que dar um @JsonIgnore (dentro de Produto). Pedido e Produto: ignonar a lista de temPedido no Produto. Assim como no Produto fizemos a ignore de id’s, temos que fazer no dos métodos get do getPedido pois está dando a referência cíclica, dentro do ItemPedido.

**OBS: os métodos get é entendido que tem que serializar!**

ATUALIZAÇÃO DE MATERIAL: ONDE ESTÁ BACKREFERENCE UTILIZAR JSONIGONRE!!!

Novo capitulo!!

Aula de POST

(Comando git no doc Git do Hybris para mudar repositório já existente para outro)

Como abrir um novo projeto (existente) no STS? File – Import – Maven – Existing Maven Projects – (procurar a pasta).

Em CategoriaResources

//método para receber um Categoria no formato JSON e inserí-lo no banco de dados

public ResponseEntity<Void> insert(Categoria obj){ //resposta HTTP sem corpo

obj = service.insert(obj); //obj é inserido no Banco de Dados e o próprio banco vai atribuir novo ID e fornecer como argumento da URI

URI, ex: localhost:8080/categorias/1

Após finalizar a Classe, abrir o Postman e dar um Post : localhost:8080/categorias no raw e escolhido o JSON, depois colocado URI com o endpoint e dado get, ok deu certo

Começo do uso do DTO: Data Transfer Object-> objeto que eu vou ter só para dados que vou fazer no sistema. Muito utilizado nos frameworks mostrando aquilo que vc quer.

No caso na hora de mostrar o endpoint Categorias, não abrirá o “leque” dos produtos

Colocando endpoint de paginação

Uso da paginação com parâmetros na requisição: busca de tanto em tanto no banco de dados, para não sobrecarregar o sistema. Ex: buscar de 20 em 20.

Quando criamos o método do page não vamos fazer assim: /categorias/page/0/20 (página 0 com 20 linhas), ou seja, não serão variáveis do próprio path, e sim como parâmetros...

Teste da paginação: http://localhost:8080/categorias/page?linesPerPage=3&page=1

***Bean validation***

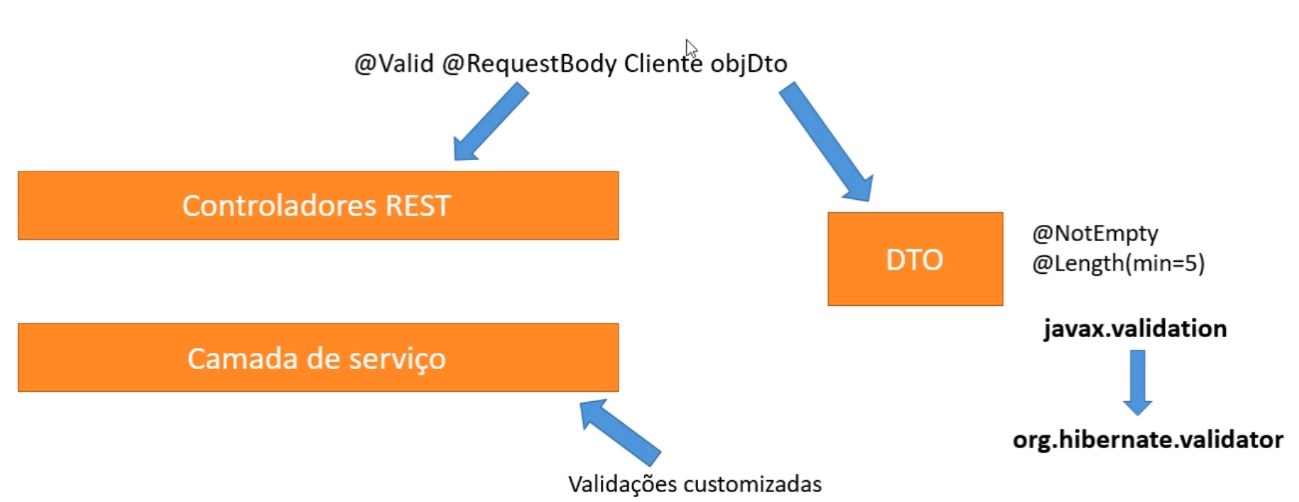
Validação de dados

Sem acesso a dados: quando não precisa acessar banco do back-end

Com acesso: ex: email não pode ser repetido ou qualquer regra que é verificado banco de dados.

Para fazer a validação, faremos pelo Framework, portanto, ele faz no Controlador REST (post ou put)....antes de passar pela camada de serviço já é validado no REST.





Vamos colocar anotações no DTO pelo Java EE... Lá no site da Oracle tem as validações

