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

Camada de serviço: oferece consultas para os controladores REST’s. Não tem contato com nada, apenas regras de negócio.

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

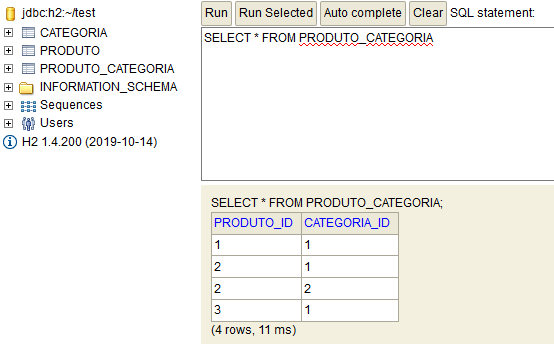
Camada de acesso a dados -> CategoriaRepository.java

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 precisa!!!