Resolução

1) (Fácil) O que é Spring Data JPA? Explique sua relação com JPA
Resultado esperado:
O Spring Data JPA é um módulo do ecossistema Spring responsável pela construção e manutenção da camada de persistência da aplicação, ou seja, comunicação com o Banco de Dados. O JPA é apenas a especificação Java para acesso a banco de dados. Podemos dizer que o JPA é a parte teórica e o Spring Data JPA é uma das implementações dessa teoria.

2) (Fácil) Para o correto funcionamento da comunicação entre o micro serviço e o Banco de Dados, foi preciso ajustar o arquivo "application.yml" conforme destacado no código abaixo. O que é a configuração "**ddl-auto: update**"?

```yaml
spring:
  profiles:
    active: local
  mvc:
    pathmatch:
      matching-strategy: ant-path-matcher
  datasource:
    url: jdbc:postgresql://localhost:5432/demo-api
    username: postgres
    password: senhaDoDatabase
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

Resultado esperado:
Esta configuração permite que a camada de persistência crie as informações (tabelas e colunas), sempre que o projeto for executado. Todas as vezes que a API executar, será verificado pelo Spring Data JPA (Hibernate) se existe alguma entidade mapeada que precisa ser criada no banco de dados.

3) (Fácil) Para manipular o banco de dados e os registros de suas tabelas, o Spring Data JPA utiliza o conceito de ORM, Mapeamento Objeto Relacional em português. Dito isto, como uma classe de modelo deve ser configurada de modo que seja visualizada pelo ORM?
Resultado esperado:
Todas as classes de modelo que precisam ser persistidas no banco de dados, devem ter minimamente a anotação @Entity, indicando que se trata de uma entidade no banco de dados, conforme exemplo:

```java
@Data
@Entity(name = "tb_produto")
public class Produto {
  //Codigo removido
}
```

4) (Fácil) Quais são as diferenças entre uma abordagem "Code First" e uma abordagem "Database First"?
Resultado esperado:
As duas abordagens possuem o mesmo objetivo, sincronizar as entidades do banco de dados com as classes de entidade da aplicação. A diferença que quando estamos trabalhando com "Code First", as classes de modelos são usadas para criar as tabelas do

banco de dados. Quando estamos trabalhando com “Database First”, as classes de modelo da aplicação precisam ser mapeadas de acordo com as especificações das tabelas.

5) (Fácil) Por que quando utilizamos o recurso JpaRepository, não é necessário codificar os comandos SQL?
Resultado esperado:
Quando criamos uma interface que estende de JpaRepository, já existe no Spring Data JPA uma implementação padrão com as principais operações de acesso e manipulação de uma tabela/Entidade (Leitura, pesquisa, inclusão, alteração, exclusão). Caso o desenvolvedor ache necessário, pode implementar as operações de forma customizada.

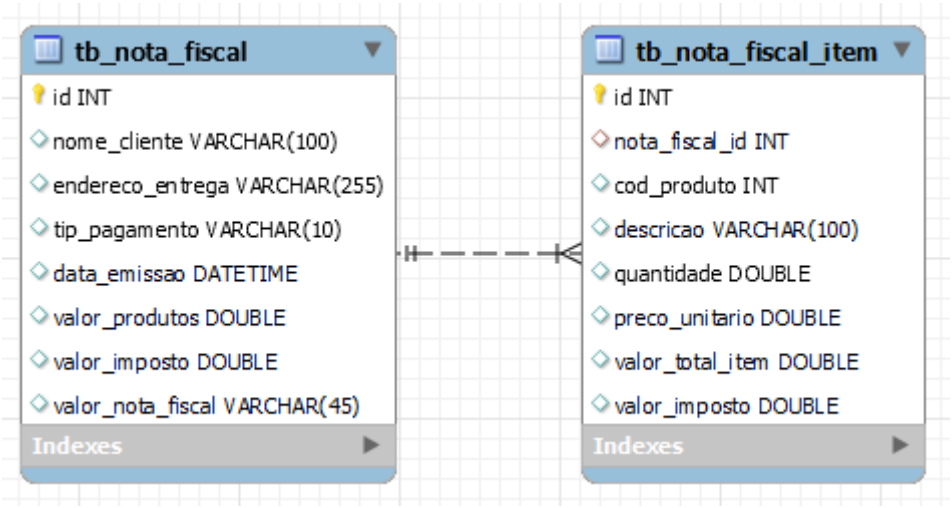
6) (Médio) Dado a classe “Cliente” e o repositório “ClienteRepository” apresentados abaixo, implemente um método capaz de filtrar a tabela por número de documento e por nome do Responsável, considerando apenas parte do nome. Este método pode retornar mais de um cliente.

```java
@Data
@Entity(name = "tb_cliente")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "codigo")
    private Long id;

    @Column(length = 150)
    private String razaoSocial;

    @Column(name = "nom_fantasia", length = 200)
    private String nomeFantasia;

    @Column(length = 20)
    private String numeroDocumento;

    @Column(name = "responsavel", length = 100)
    private String nomeResponsavel;
}
```

```java
public interface ClienteRepository extends JpaRepository<Cliente, Long> {
    // Implementação aqui
}
```

Resultado esperado:
Public List<Cliente> findByNumeroDocumentoAndNomeResposanvelContaingIgnoreCase(String documento, String resposavel);

7) (Médio) Dado as tabelas “tb_nota_fiscal” e “tb_nota_fiscal_item” apresentadas abaixo. Construa as classes de modelo que as representam, considerando o relacionamento entre elas.



Resultado esperado:

```java
@Data
@Entity(name = "tb_nota_fiscal")
public class NotaFiscal {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nomeCliente;
    private String enderecoEntrega;
    private String tipPagamento;
    private LocalDateTime dataEmissao;
    private Double valorProdutos;
    private Double valorImposto;
    private Double valorNotaFiscal;
}
```

```java
@Data
@Entity(name = "tb_nota_fiscal_item")
public class NotaFiscalItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    private NotaFiscal notaFiscal;

    private Integer codProduto;
    private String descricao;
    private Double quantidade;
    private Double precoUnitario;
    private Double valorTotalItem;
    private Double valorImposto;
}
```

8) (Médio) Dado a classe Cliente representada abaixo, implemente um método “filtro” no controller ClienteController, de modo que seja permitido filtrar os clientes por: RazaoSocial, NomeFantasia, NumeroDocumento e NomeResponsavel. A URL do endpoint deve permitir que o usuário informe apenas os campos desejados, conforme exemplo.

```java
@Data
@Entity(name = "tb_cliente")
public class Cliente {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "codigo")
    private Long id;

    @Column(length = 150)
    private String razaoSocial;

    @Column(name = "nom_fantasia", length = 200)
    private String nomeFantasia;

    @Column(length = 20)
    private String numeroDocumento;

    @Column(name = "responsavel", length = 100)
    private String nomeResponsavel;
}
```

Exemplo de Requisição
http://localhost:8080/produtos/filtro?dataValidade=2022-12-04&precoVenda=12
Resultado esperado:

```java
@RestController
public class ClienteController {
    @GetMapping(value = "/filtro")
    public ResponseEntity<List<Cliente>> filtro(FiltroClienteParam params) {
        //Implementação do Método retirada
        return null;
    }
}
```

9) (Médio) Dado o exemplo da questão anterior, apresente a implementação completa do CriteriaBuilder para filtrar os clientes de acordo com o formulário de parâmetros.

Resultado esperado:

```java
public interface ClienteRepositoryCustom {
    List<Cliente> getFiltro(FiltroClienteParam params);
}
```

```java
public class ClienteRepositoryCustomImpl implements ClienteRepositoryCustom {
    private EntityManager entityManager;
    @Override
    public List<Cliente> getFiltro(FiltroClienteParam params) {
        CriteriaBuilder criteriaBuilder =
this.entityManager.getCriteriaBuilder();
        CriteriaQuery<Cliente> query = criteriaBuil-
der.createQuery(Cliente.class);

        Root<Cliente> clienteRoot = query.from(Cliente.class);
        List<Predicate> predicates = new ArrayList<>();

        if ( params.getNomeFantasia() != null ) {
            predicates.add(criteriaBuilder.like(clienteRoot.get("nomeFantasia"),
"%" + params.getNomeFantasia() + "%"));
        }
        if ( params.getNomeResponsavel() != null ) {
            predica-
tes.add(criteriaBuilder.like(clienteRoot.get("nomeResponsavel"), "%" + pa-
rams.getNomeResponsavel() + "%"));
        }
```

```
        if ( params.getNumeroDocumento() != null ) {
            predica-
tes.add(criteriaBuilder.like(clienteRoot.get("numeroDocumento"), "%" + pa-
rams.getNumeroDocumento() + "%"));
        }
        if ( params.getRazaoSocial() != null ) {
            predicates.add(criteriaBuilder.like(clienteRoot.get("razaoSocial"),
"%" + params.getRazaoSocial() + "%"));
        }
        if (!predicates.isEmpty()) {
            query.where(predicates.stream().toArray(Predicate[]::new));
        }
        TypedQuery<Cliente> queryResult = this.entityManager.createQuery(query);
        return queryResult.getResultList();
    }
}
```

10) (Médio) Baseado na solução do exercício anterior, quais são as etapas para construção do CriteriaBuilder?
Resultado esperado:
As etapas para criação são: Criação do Criteria, criação da Query, definição do Root, inclusão das condições, execução da Criteria e recuperação dos resultados.

11) (Difícil) Considerando as diferenças entre abordagem "**Code First**" e "**Database First**". Explique qual é cenário mais adequado para cada abordagem. Justifique sua resposta.
Resultado esperado:
Sabemos que ambas as abordagens têm o mesmo objetivo final, que é mapear o banco de dados com as classes de modelo da aplicação, porém, cada cenário exige uma análise. Por questões organizacionais, pode não ser permitido a configuração do Code First, nos obrigando a modelar as tabelas do banco de Dados manualmente antes da construção do projeto JPA.

12) (Difícil) Dados a implementação das classes abaixo que filtram os clientes por documento, analise o código e aponte onde esta o erro de implementação:

```
@RestController
public class ClienteController {
    @Autowired
    private ClienteService service;
    @GetMapping(value = "/cliente/{numeroDocumento}")
    public ResponseEntity<list<Cliente>> getClienteByDocumento(String numeroDo-
cumento) {
        return ResponseEntity.ok(service.buscarPorDocumento(numeroDocumento));
    }
}
```

```
public class ClienteService {
    @Autowired
    private ClienteRepository repository;
    public List<Cliente> buscarPorDocumento(String documento) {
        return repository.findByNumeroDocumento(documento);
    }
}
```

```
public interface ClienteRepository extends JpaRepository<Cliente, Long> {
    public Cliente findByNumeroDocumento(String numeroDocumento);
}
```

Resultado esperado:

Na classe ClienteController, está previsto a possibilidade de retornar mais de um registro, representado por uma lista de Cliente, contudo, nas camadas inferiores (service e Repository), os métodos foram desenvolvidos considerando apenas um único cliente por vez.

13) (Difícil) Considerando a implementação de pesquisa por Criteria da entidade “Produto” representada abaixo, refatore o código de forma que o filtro aplicado considere os valores maiores ou iguais ao informado no parâmetro.

```
@Override
public List<Produto> getFiltro(ProdutoFilterParam params) {
   CriteriaBuilder criteriaBuilder = this.entityManager.getCriteriaBuilder();
   CriteriaQuery<Produto> query = criteriaBuilder.createQuery(Produto.class);

   Root<Produto> produto = query.from(Produto.class);
   List<Predicate> predicates = new ArrayList<>();

   if ( params.getDescricao() != null ) {
      predicates.add(criteriaBuilder.like(produto.get("descricao"), "%" + pa-
rams.getDescricao() + "%"));
   }
   if ( params.getDataValidade() != null ) {
      predicates.add(criteriaBuilder.equal(produto.get("dataValidade"), pa-
rams.getDataValidade()));
   }
   if ( params.getPrecoVenda() != null ) {
      predicates.add(criteriaBuilder.equal(produto.get("precoVenda"), pa-
rams.getPrecoVenda()));
   }
   if (!predicates.isEmpty()) {
      query.where(predicates.stream().toArray(Predicate[]::new));
   }
   TypedQuery<Produto> queryResult = this.entityManager.createQuery(query);
   return queryResult.getResultList();
}
```

Resultado esperado:

```
if (params.getPrecoVenda() != null ) {
   predica-
tes.add(criteriaBuilder.greaterThanOrEqualTo(produto.get("precoVenda"), pa-
rams.getPrecoVenda()));
}
```

14) (Difícil) A interface CriteriaBuilder, que utilizamos como apoio para construção de Filtros, possui diversos tipos de predicados, como por exemplo “criteriaBuilder.equal()”. Explique qual é a principal diferença entre os predicatos “criteriaBuilder.greaterThan” e “criteriaBuilder.gt()”

Resultado esperado:

Tecnicamente não existe diferença entre elas, os dois métodos executam a mesma atividade e retornam o mesmo resultado.

15) (Difícil) Dado a interface “ProdutoRepository”, considerando que seja necessário implementar um novo filtro na entidade “Produto”, buscando por descrição e dataValidade, porem é preciso retornar mais de um registro (Lista de Produtos). Podemos criar um novo método com o mesmo nome?

```
public interface ProdutoRepository extends JpaRepository<Produto, Integer> {

    public List<Produto> findByDescricaoContainingIgnoreCase(String descricao);
    public Produto findByDescricaoAndDataValidade(String descricao, LocalDate
```

```
dataValidade);

}
```

Resultado esperado:

A criação de dois métodos com a mesma assinatura (nome e parâmetros) não é permitida pelo Java. Essa não é uma questão exclusiva do Spring JPA. Contudo, pensando no requisito proposto, o mais indicado é modificar o retorno do método findByDescricaoAndDataValidade, retornando vários registros e, dentro da camada de Serviço tratar os casos que precisam devolver apenas um registro.