Módulo Spring Data

Fontes e Links de referência

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#preface
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods
https://medium.com/huawei-developers/database-relationships-in-spring-data-jpa-8d7181f50f60
https://www.baeldung.com/jpa-embedded-embeddable
https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories.core-concepts

---

Sumário

O que é o Spring Data?

- O Spring Data tem por objetivo facilitar nosso trabalho com persistência de dados de uma forma geral. Ele possui vários outros projetos:
  - Spring Data Commons
  - Spring Data Gemfire
  - Spring Data KeyValue
  - Spring Data LDAP
  - Spring Data MongoDB
  - Spring Data REST
  - Spring Data Redis
  - Spring Data for Apache Cassandra
  - Spring Data JPA
- Para o treinamento, vamos utilizar o projeto **Spring Data JPA**.

Definição e Configuração Spring Data JPA

- O Spring Data JPA é um framework que nasceu para facilitar a criação dos nossos repositórios
- Ele faz isso nos liberando de ter que implementar as interfaces referentes aos nossos repositórios (ou DAOs)
- Já deixa pré-implementado algumas funcionalidades como, por exemplo, de ordenação das consultas e de paginação de registros

Configuração do Spring Data JPA

Passos:

1. Colocar a dependência do spring jpa e do driver do banco de dados

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>com.oracle.database.jdbc</groupId>
  <artifactId>ojdbc8</artifactId>
  <scope>runtime</scope>
</dependency>
```

2. Configurar parâmetros do banco de dados no arquivo application.properties.

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=oracle
spring.datasource.driverClassName=oracle.jdbc.driver.OracleDriver
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.properties.hibernate.default_schema=VEM_SER
```

```
# create and drop tables and sequences, loads import.sql
# spring.jpa.hibernate.ddl-auto=create-drop
# none, validate, update, create-drop

spring.jpa.show-sql=true
log4j.logger.org.hibernate.type=trace
spring.jpa.properties.hibernate.format_sql=true
```

3. Configurar entidades (tabelas do banco de dados)

```
@Entity(name = "PESSOA")

public class PessoaEntity {
```

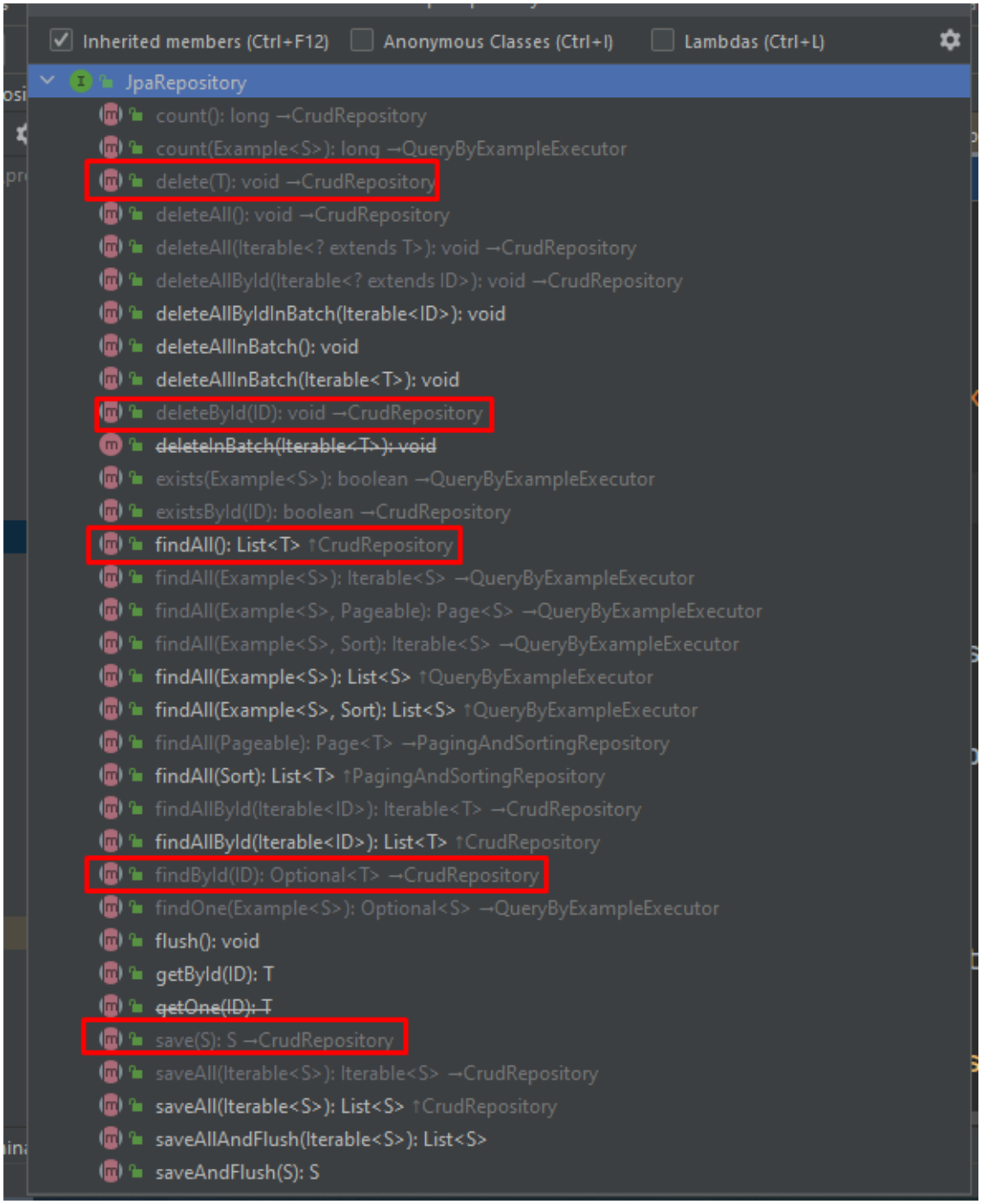
```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "PESSOA_SEQ")
@SequenceGenerator(name = "PESSOA_SEQ", sequenceName = "seq_pessoa2", allocationSize = 1)
@Column(name = "id_pessoa")
private Integer idPessoa;

@Column(name = "nome")
private String nome;
```

4. Configurar repositórios (JPARepository)

```
@Repository
public interface PessoaRepository extends JpaRepository<PessoaEntity, Integer> {
```

- Ela tem todos os métodos que a gente precisa para fazer um CRUD (criar, ler, atualizar, deletar).



Exercício #1

- Criar uma pasta no seu repositório “modulo3.2”, copiar o seu pessoa-api do módulo anterior para dentro dessa pasta
- Executar o script “script_aula1.sql” no seu banco de dados
- Configurar o Spring JPA no seu projeto pessoa-api
- Configurar o PessoaEntity com as anotações corretas
- Configurar o PessoaRepository com o JPARepository
- Ajustar o projeto para funcionar com os métodos do JPARepository

Métodos de Consulta: Query Methods

Permite criar consultas por assinatura de métodos

```
1  public interface Produtos extends JpaRepository<Produto, Long> {
2
3    Produto findByNome(String nome);
4
5  }
```

Alguns exemplos de Query Methods:

```
public interface Produtos extends JpaRepository<Produto, Long> {
    Produto findByNome(String nome);

    // Equivalente ao like, mas não precisamos nos preocupar com o sinal de percentual.
    // Podemos usar também EndingWith, Containing.
    List<Produto> findByNomeStartingWith(String nome);

    // Ordenando pelo nome.
    List<Produto> findByNomeStartingWithOrderByNome(String nome);

    // Não levar em consideração a caixa.
    List<Produto> findByNomeStartingWithIgnoreCase(String nome);

    // Pesquisando por duas propriedades: nome e ativo.
    List<Produto> findByNomeStartingWithIgnoreCaseAndAtivoEquals(String nome, boolean ativo);

    // Nesse caso, precisamos usar o sinal de percentual em nossas consultas.
    List<Produto> findByNomeLike(String nome);

    // Podemos usar também IsNotNull ou NotNull.
    List<Produto> findByDescricaoIsNull();

    // Quando você quer negar o que passa no parâmetro
    List<Produto> findByNomeNot(String nome);

    // Todos os produtos com os IDs passados no varargs. Poderia usar NotIn para negar os IDs.
    List<Produto> findByIdIn(Long... ids);

    // Todos onde a propriedade ativo é true. Poderia ser falso, usando False.
    List<Produto> findByAtivoTrue();

    // Buscar onde a data de cadastro é depois do parâmetro passado.
    // Pode ser usado Before também.
    List<Produto> findByCadastroAfter(Date data);

    // Buscar onde a data cadastro está dentro de um período.
    List<Produto> findByCadastroBetween(Date inicio, Date fim);

    // Todos com quantidade "menor que". Poderia ser usado
    // também LessThanEqual, GreaterThan, GreaterThanEqual.
    List<Produto> findByQuantidadeLessThan(int quantidade);
}
```

Exercício #2

- Criar métodos nas respectivas repositoryes para busca
  - Pessoas
    - por nome (contains ignore case)
    - por cpf
    - por data de nascimento que está entre data inicial e final que o usuário irá informar
  - Contato
    - por tipo
  - Endereço
    - por tipo
    - por cep ordenado pelo logradouro
- Criar uma controller (ConsultasController (“/consultas”)) e injetar as repositoryies (somente para testes) e para cada consulta faça um método para testar

Classes embutidas e chaves compostas

- **javax.persistence.Embeddable**

A JPA fornece a anotação @Embeddable para declarar que uma classe será incorporada por outras entidades.

```
@Embeddable
public class ContactPerson {
    private String firstName;
    private String lastName;
    private String phone;
    // standard getters, setters
}
```

- **javax.persistence.Embedded**

O @Embedded diz para o JPA que, tudo que estiver dentro do objeto que tem a tag @Embedded, vai trazer um nivel para cima, todos os seus campos.

```
@Entity
public class Company {
    @Id
    @GeneratedValue
    private Integer id;
    private String name;
    private String address;
    private String phone;
    @Embedded
    private ContactPerson contactPerson;
    // standard getters, setters
}
```

- **javax.persistence.EmbeddedId**

A anotação @EmbeddedId é usada para especificar que o identificador de entidade é um tipo incorporável e é uma chave composta.

```
@Entity
public class Book {
    @EmbeddedId
    private BookId id;
    private String genre;
    private Integer price;
    //standard getters and setters
}
```

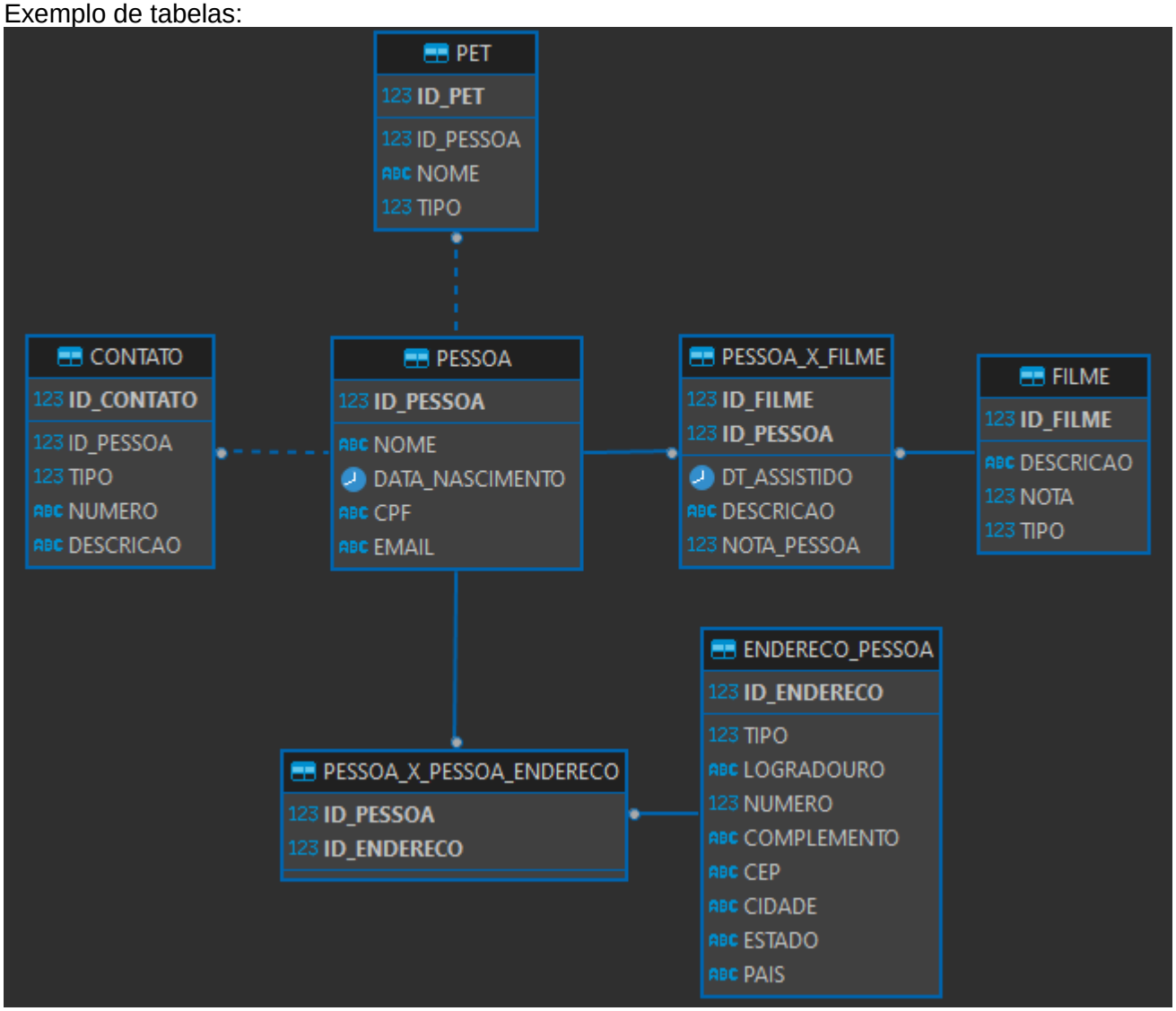
Exercício #3

- Crie no banco de dados e mapeie no JPA estrutura abaixo:

- Após a criação da estrutura e mapeamento faça:
    - Criar Controller/Service/Repository com todas as operações para Filmes
        - create, update, delete, list
        - /avaliar-filme/{idUsuario}: esse endpoint irá fazer a avaliação do filme para preencher a tabela de relacionamento (PESSOA_X_FILME)

Relacionamentos

Exemplo de tabelas:




```java
@ManyToOne
@JoinColumn(name = "id_pessoa", referencedColumnName = "id_pessoa")
private PessoaEntity pessoaEntity;
```

```java
@OneToMany(mappedBy = "pessoaEntity", cascade = CascadeType.ALL, orphanRemoval = true)
private Set<ContatoEntity> contatos;
```

```java
@OneToOne(fetch = FetchType.LAZY)
@JoinColumn(name = "id_pet", referencedColumnName = "id_pet")
private PetEntity pet;
```

```java
@ManyToMany
@JoinTable(name = "Pessoa_X_Pessoa_Endereco",
        joinColumns = @JoinColumn(name="id_pessoa"),
        inverseJoinColumns = @JoinColumn(name="id_endereco")
)
private Set<EnderecoEntity> enderecos;
```

```java
@ManyToMany(mappedBy = "enderecos")
private Set<PessoaEntity> pessoas;
```

Java Persistence Query Language (JPQL)

- JPQL (Java Persistence Query Language) é uma linguagem para desenvolver consultas em banco de dados em Java.
- Foi desenvolvido tendo como base o SQL (Structured Query Language), porém podendo trabalhar com objetos dentro da linguagem Java

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

- Para fazer Consultas JPQL basta usar a anotação @Query

Exercício #4

- Construir as seguintes consultas abaixo com @Query para consultar:
  - endereços por país
  - endereços por id da pessoa
  - contatos por tipo de contato
  - pessoas por data de nascimento entre duas datas
  - pessoas que possuem endereço
- **Apenas para facilitar, nesse exercício utilizar direto no controller ConsultasController os repositoryes...**

Queryes nativas

- Da mesma forma como podemos fazer consultas JPQL, podemos fazer consultas utilizando SQL nativo.

```
public interface UserRepository extends JpaRepository<User, Long> {

  @Query(value = "SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1", nativeQuery = true)
  User findByEmailAddress(String emailAddress);
}
```

- OBS: Só usar em casos onde o JPQL não se aplica

Exercício #5

- Construir as seguintes consultas abaixo com @Query nativa para consultar:
  - endereços por cidade ou país
  - endereços sem complemento
  - contatos por id da pessoa
  - pessoas que não possuem endereço
  - **Apenas para facilitar, nesse exercício utilizar direto no controller ConsultasController os repositoryes...**

Queryes Personalizadas

- É possível fazer uma consulta JPQL ou nativa específica e retornar um resultado personalizado (somente usar em casos de relatórios ou casos específicos)

```
@Query("     select  new com.dbc.vemser.dto.response.AlunoCompletoDTO(" +
        "                e.idAluno," +
        "                u.idUsuario, " +
        "                u.nome," +
        "                u.email," +
        "                e.stack, " +
        "                e.semestre, " +
        "                e.git, " +
        "                e.curso, " +
        "                e.sistemaOperacional, " +
        "                e.processador, " +
        "                e.memoriaRam, " +
        "                e.cidadeEstado, " +
        "                e.idEdicao," +
        "                edc.descricao) " +
        "         from AlunoEntity e " +
        "         join e.usuario u " +
        "    left join e.edicao edc" +
        "        where (:nome is null or upper(u.nome) like upper(concat('%',:nome,'%')))" +
        "          and (:stack is null or e.stack = :stack)" +
        "          and (:idEdicao is null or e.idEdicao = :idEdicao)")
Page<AlunoCompletoDTO> findAllWithUser(@Param("nome") String nome,
                                       @Param("stack") Stack stack,
                                       @Param("idEdicao") Integer idEdicao,
                                       Pageable pageable);
```

Paginação e Ordenação de Registros

- A paginação costuma ser útil quando temos um grande conjunto de dados e queremos apresentá-lo ao usuário em partes menores.

Exemplos de aplicação: página de produtos do mercado livre, pesquisa do google, linha do tempo das redes sociais (conceito de scroll infinito é uma paginação)

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ModuloRepository extends JpaRepository<ModuloEntity, Integer> {

    Page<ModuloEntity> findAllByNomeContainingIgnoreCaseAndAtivo(Pageable pageable,
                                                                 String nome,
                                                                 Ativo ativo);
    Page<ModuloEntity> findAllByIdModuloAndAtivo(Pageable pageable,
                                                 Integer id,
                                                 Ativo ativo);
    Page<ModuloEntity> findAllByAtivo(Ativo ativo,
                                      Pageable pageable);
}
```

- Podemos estender **JpaRepository**, já que ele estende **PagingAndSortingRepository** também.

- Para utilizar a paginação, temos métodos e classes padrão para isso, são elas:
  - Pageable
  - PageRequest
  - Sort
  - Page
- Além disso, podemos classificar esses dados por alguns critérios durante a paginação.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
...
@RestController
@RequestMapping("/paginacao")
@RequiredArgsConstructor
public class PaginacaoController {
    private final PessoaRepository pessoaRepository;

    @GetMapping("/lista-paginada-ordenada")
    public Page<PessoaEntity> listPaginadaOrnenadaPorCpf(@RequestParam Integer paginaSolicitada,
                                                         @RequestParam Integer tamanhoPagina) {
        Pageable solicitacaoPaginaSemOrdenacao = PageRequest.of(paginaSolicitada, tamanhoPagina);
        Pageable solicitacaoPagina = PageRequest.of(paginaSolicitada,
                                        tamanhoPagina,
                                        Sort.by("cpf").ascending()
                                        .and(Sort.by("nome")));
        Page<PessoaEntity> all = pessoaRepository.findAll(solicitacaoPagina);
        return all;
    }
}
```

- Sobre o objeto **Page**, ele possui vários dados que não são tão utilizados, por esse motivo iremos criar um DTO somente com o que é relevante no momento.

```
@AllArgsConstructor
@NoArgsConstructor
@Data
public class PageDTO<T> {
    private Long totalElementos;
    private Integer quantidadePaginas;
    private Integer pagina;
    private Integer tamanho;
    private List<T> elementos;
}
```

```java
public PageDTO<ProgramaDTO> listarPorNomeId(Integer idPrograma,
                                            String nome,
                                            Integer pagina,
                                            Integer tamanho) {
    Pageable solicitacaoPagina = PageRequest.of(pagina, tamanho);
    Page<ProgramaEntity> paginaDeProgramasPorNomeId = repository.findAllByIdPrograma(idPrograma,
                                                                                     nome,
                                                                                     solicitacaoPagina);

    List<ProgramaDTO> paginaDeProgramasDTO = paginaDeProgramasPorNomeId.getContent().stream()
            .map(x -> objectMapper.convertValue(x, ProgramaDTO.class))
            .toList();

    return new PageDTO<>(paginaDeProgramasDTO.getTotalElements(),
                         paginaDeProgramasDTO.getTotalPages(),
                         pagina,
                         tamanho,
                         paginaDeProgramasDTO);
}
```

Exercício #6

- Crie um endpoint paginado para trazer todas as pessoas filtradas por
  data de nascimento >= dataInformada, se não passar nenhuma data trazer todos os registros com os parâmetros informados. Além disso, o serviço deve ordenar pelo nome de (A-Z).
- O serviço deve receber:
  - Página solicitada (0 – N)
  - Quantidade de registros por página (1 – N)
  - Data de Nascimento (não obrigatório)
- Retornar um PageDTO<Pessoa>