Lista 26 – *Back-End* Spring

1. Utilizando como base o sistema desenvolvido nas aulas dos dias 56 (23.02.2023) e 57 (24.02.2023), elabore do zero um sistema semelhante que realize o cadastro de funcionários, da seguinte forma:
    a. Na criação do projeto inclua os *starters* Spring DevTools, Spring Web, H2, PostgreSQL, Lombok e Spring Data JPA. Após o projeto ser criado, adicione no $pom.xml$ os webjars do EasyAutoComplete, Bootstrap e jQuery (copie as dependências relativas aos webjars do $pom.xml$ de algum projeto anterior).
    b. Crie 2 perfis adicionais: $application\text{-}dev.properties$ e $application\text{-}test.properties$. Utilize como base os perfis criados na aula de ontem.
        - $application\text{-}test.properties$: SGBD H2
        - $application\text{-}dev.properties$: SGBD PostgreSQL, banco de dados `empresa`; use o DBeaver para criar o banco de dados.
    c. A tabela se chamará `funcionario`, com os campos abaixo:
        - `id: LONG AUTO_INCREMENT`
        - `nome: VARCHAR(100)`
        - `num_dep: INTEGER` (número de dependentes, se desejar use outro nome para este campo)
        - `salario: NUMERIC(10, 2)`
        - `nascimento: DATE`
    d. A entidade deverá ser uma classe chamada `Funcionario`, com os atributos:
        - `id: Long`
        - `nome: String`
        - `num_dep: Integer`
        - `salario: BigDecimal`
        - `nascimento: Date`
    e. A entidade `Funcionario` deverá dar suporte à representação XML. Para fazer isso anote a classe com `@XmlRootElement`.
    f. Crie uma interface chamada `FuncionarioRepository` que estende de `JpaRepository`, de forma a poder implementar mais tarde a funcionalidade de consultas paginadas.
        - Declare um método personalizado em `FuncionarioRepository`, sem utilizar `@Query` e nem JPQL, que busque os funcionários por uma parte do nome sem levar em consideração as letras maiúsculas / minúsculas.
        - Crie uma consulta personalizada com `@Query` / JPQL de forma a buscar funcionários por uma faixa salarial (maior ou igual a um salário mínimo informado e menor ou igual a um salário máximo informado).
        - Crie uma consulta personalizada com `@Query` / JPQL chamada `possuiDependentes()` que retorne somente os funcionários que possuam ao menos 1 dependente.
    g. Crie uma classe de serviço que implemente os métodos da *interface* abaixo:

```
public interface FuncionarioService {
    void popularTabelaInicial();

    Funcionario incluir(Funcionario funcionario);
    Funcionario alterar(Long id, Funcionario funcionario);
    void excluir(Long id);

    List<Funcionario> listar();
    Page<FuncionarioDTO> listarPaginado(Pageable pagina);

    Funcionario buscar(Long id);      // Lança uma exceção caso o não exista o funcionario com id procurado
    Funcionario buscarPorId(Long id); // Retorna um novo objeto Funcionario caso id não seja encontrado
    Optional<Funcionario> porId(Long id);

    List<Livro> buscarPorNome(String nome);
    List<Livro> buscarPorFaixaSalarial(BigDecimal salarioMinimo, BigDecimal salarioMaximo);
    List<Funcionario> buscarPossuiDependentes(Integer numDependentes);
}
```

h. Crie uma classe do tipo REST *controller* chamada `FuncionarioController` que implemente os seguintes métodos:
    - `public Iterable<Funcionario> listar()`: Listar todos os funcionários
    - `public ResponseEntity<Page<FuncionarioDTO>> listarPaginado(Pageable pagina)`: Listar os funcionários utilizando paginação
    - `public ResponseEntity<Funcionario> porId(@PathVariable Long id)`: Listar somente um funcionário cujo código é informado na URL por *path param*
    - `public Livro porIdXML(@PathVariable Long id)`: Semelhante ao acima mas usando o *end-point* extra '/xml' e retornando a resposta em formato XML
    - `List<Livro> porNome(@RequestParam("nome") String nome)`: Lista funcionários filtrando por uma parte do nome
    - `public List<Funcionario> porFaixaSalarial`: Procure implementar este método utilizando *query parameters* de forma que os valores mínimo e máximo sejam opcionais
    - `public ResponseEntity<Void> incluir(@RequestBody Funcionario funcionario)`: Método deverá incluir um funcionário e retornar a URI do recurso criado
    - `public ResponseEntity<Funcionario> atualizar(@PathVariable("id") Long id, @RequestBody Funcionario funcionario)`: Alteração de funcionário
    - `public ResponseEntity<Void> excluir(@PathVariable("id") Long id)`: Exclusão de funcionário
i. Implemente na classe principal o método `commandLineRunner()` e faça alguns testes utilizando para isso a classe de serviço. Use esse método para popular a tabela de funcionários e fazer testes com as funcionalidades do CRUD.
j. Crie uma classe chamada `FuncionarioDTO` que contenha todos os atributos de `Funcionario`, exceto o campo `id`. Crie um construtor que receba uma instância de `Funcionario` e copie os valores de seus atributos para os atributos de `FuncionarioDTO`.
k. Crie uma funcionalidade de consulta paginada:
    - O método na classe de serviço deverá retornar um `Page<FuncionarioDTO>`.
    - O método no controlador deverá retornar um `ResponseEntity<Page<LivroDTO>>`.
l. Alterne entre os perfis para testar o sistema tanto com o SGBD H2 quanto com o SGBD PostgreSQL.
m. Integre no sistema o *front-end* feito na lista 23, adaptando-o para funcionar com esta tabela.
n. Utilize os recursos de localização do JavaScript para formatar os campos de salário e data de nascimento corretamente nas *interfaces web*. Na trilha da aula 50 há exemplos disso no projeto 'lista 22 – front-end'.

2. Utilize o Postman para testar todos os métodos da API Rest implementados na classe `FuncionarioController`. Não é necessário entregar os *prints* das telas.

3. Após finalizar o sistema desenvolvido no item 1 acima e se certificar que ele está funcionando corretamente, crie um repositório no GitHub para armazená-lo na nuvem. Utilize o GitBash para criar um repositório local e conectá-lo ao repositório no GitHub. Faça o *commit* com a versão atual do sistema e então realize o *push* para enviar o projeto para o repositório remoto.