|  |
| --- |
| **Projeto CRUD com Spring Boot** |
| **Objetivo Geral: Tutorial de Criação de Projetos CRUD com Spring Boot** |
| *Conteúdo:* |
| 1. Instalação das Ferramentas necessárias 2. Criação de Componentes Spring Boot 3. Anexando o CRUD ao projeto |
| *Metodologia e Estratégia:* |
| 1. Aula expositiva dialogada com apoio de tutorial; 2. Exercícios de aplicação. |

**O que é Spring Boot?**

Para quem ainda não ouviu falar, o Spring Boot é uma ferramenta que nasceu a partir do Spring, um framework desenvolvido para a plataforma Java baseado nos padrões de projetos, inversão de controle e injeção de dependência.

Embora o Spring framework tenha sido criado justamente com o intuito de simplificar as configurações para aplicações web, ele não atendeu 100% as expectativas do mercado ao ser lançado, já que as configurações seguiam grandes e complexas demais.

Sendo assim, um novo projeto foi acrescentado ao framework para mudar esse jogo e abstrair toda a complexidade que uma configuração completa pode trazer: o Spring Boot.

Apresentando um modelo de desenvolvimento mais simples e direto, esse framework foi determinante para que o uso do ecossistema Spring decolasse.

No geral, ele fornece a maioria dos componentes necessários em aplicações em geral de maneira pré-configurada, possibilitando uma aplicação rodando em produção rapidamente, com o esforço mínimo de configuração e implantação.

Em outras palavras, podemos entender o Spring Boot como um template pré-configurado para desenvolvimento e execução de aplicações baseadas no Spring.

**Como startar o Spring Boot?**

Agora que você já sabe mais sobre o Java Spring Boot, chegamos à pergunta que não quer calar: como começar a utilizar este framework?

Bom, para iniciar a criação do projeto, recomendamos o uso de um facilitador disponibilizado pelo próprio Spring: o Spring Initializr. Com ele, é possível habilitar os módulos desejados em seu projeto em poucos cliques.

No final, a página irá gerar um projeto Maven ou Gradle pré-configurado e com todos os componentes solicitados especificados, bastando ao desenvolvedor começar a trabalhar com a codificação.

Para usar o Spring Initializr, basta acessar https://start.spring.io/ e inserir as informações necessárias sobre projeto, como:

* Tipo de projeto que será utilizado (Maven ou Gradle);
* Linguagem que será usada no desenvolvimento back-end;
* Qual a versão do Spring Boot você pretende utilizar;
* Grupo da aplicação;
* Lista das dependências que o projeto irá usar.

Após finalizar o preenchimento, basta clicar no botão “Generate Project” para o Spring Initializr realizar a configuração e download de projeto em formato zip.

Na sequência, será preciso descompactar o arquivo e importá-lo para o IDE ou Editor de Código Fonte da sua preferência, para que você possa enfim iniciar o desenvolvimento.

Essa sequencia também pode ser feita no VS Code com a instalação da extenção Spring Boot Extension Pack, poderá sem aberta diretamente pelo Editor de Códigos.

Criando um Projeto em Spring Boot

Classe Model – e a classe eu será a base do nosso projeto, nela deve conter todas os atributos e métodos dos objetos que serão administrados pelo banco de dados.

A classe **Funcionario** com os três atributos do diagrama de classes:

**public class** Funcionario {

**private long** id;

**private** String nome;

**private** String email;

Para o acesso dos frameworks às classes que estão dentro de Models e Controllers, devem ser implementados os conceitos de encapsulamento, que é a aplicação dos métodos Getters and Setters para cada atributo de uma classe.

Para que o Model funcione precisamos incluir no POM.xml as dependecias do banco de dados e da JPA

</dependency>

        <dependency>

            <groupId>org.postgresql</groupId>

            <artifactId>postgresql</artifactId>

            <scope>runtime</scope>

        </dependency>

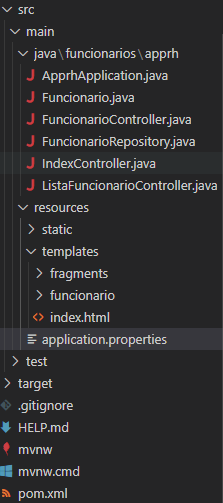
        <dependency>

            <groupId>org.springframework.boot</groupId>

            <artifactId>spring-boot-starter-data-jpa</artifactId>

        </dependency>

Após editar o arquivo pom.xml, o IDE automaticamente fará o download das dependências, sendo necessário alterar o aplication.properties conforme a figura a seguir:



spring.jpa.database=POSTGRESQL

spring.sql.init.platform=postgres

spring.jpa.show-sql=true

spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=jdbc:postgresql://localhost:5432/postgres

spring.datasource.username=postgres

spring.datasource.password=postgres

server.port=8080

O que e JPA?

O Spring Data JPA é uma framework que faz parte do conjunto de projetos do Spring Data que tem como finalidade tornar a integração de aplicações Spring com a JPA (Java Persistence API), uma de suas principais vantagens é a capacidade que o mesmo possui para criar a camada de acesso aos dados sem a necessidade de termos que implementar manualmente as famosas classes de DAO (Data Access Object).

Ajustando o Model – Criando as Entidades.

Após a implantação das dependências JPA e do BD, é necessário inserir, na classe Model, anotações para que as dependências tenham efeito na classe escolhida e esta, consequentemente, seja gravada no banco de dados. O primeiro item a ser inserido é a anotação @Entity e insere-se o pacote javax.persistence da dependência JPA.

Assim, a Classe Funcionario.java deverá ficar da seguinte forma

import java.io.Serializable;

import javax.persistence.\*;

@Entity

public class Funcionario implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id

    @GeneratedValue(strategy = GenerationType.AUTO)

    private long id;

    private String nome;

    private String email;

    public long getId() {

        return id;

    }

    public void setId(long id) {

        this.id = id;

    }

    public String getNome() {

        return nome;

    }

    public void setNome(String nome) {

        this.nome = nome;

    }

    public String getEmail() {

        return email;

    }

    public void setEmail(String email) {

        this.email = email;

    }

}

A linha @Entity é uma notação que diz ao JPA que a classe Funcionario é uma entidade, assim, o framework JPA tratará a classe Funcionario como uma classe que poderá ser persistida (armazenada em banco de dados).

Na linha “public class Funcionario implements Serializable{“, a classe Funcionario implementa a interface Serializable, que prepara a classe para os tratamentos do JPA. O JPA precisa que a classe seja transformada em binária para poder fazer a mudança de objeto para entidade e persistir o objeto como entidade no banco de dados. Por padrão, o JPA relaciona a classe Funcionario em uma tabela, também chamada Funcionario, com as colunas id, nome e email.

A anotação @Id para transformar o atributo long id em chave primária na tabela Funcionario dentro do banco de dados. A anotação @GeneratedValue(strategy = GenerationType.AUTO) tem como objetivo gerar uma chave primária automática e sequencial para cada funcionário persistido na tabela; caso não tenha essa anotação, o controle da chave primária será responsabilidade do programador.

A variável serialVersionUID do tipo long é uma propriedade dentro da classe Funcionario que fará com que o objeto Funcionario tenha uma numeração interna e única para o JPA, por isso os comandos static e final.

**Criando a Interface Repository**

O que é interface ? - A interface é um recurso muito utilizado em Java, bem como na maioria das linguagens orientadas a objeto, para “obrigar” a um determinado grupo de classes a ter métodos ou propriedades em comum para existir em um determinado contexto, contudo os métodos podem ser implementados em cada classe de uma maneira diferente. Pode-se dizer, a grosso modo, que uma interface é um contrato que quando assumido por uma classe deve ser implementado.

Para implementar a interface de integração JPA e o Model devemos criar um Aruivo Chamado FuncionarioRepository.java com os seguintes atributos:

import java.util.List;

import org.springframework.data.jpa.repository.Query;

import org.springframework.data.repository.CrudRepository;

public interface FuncionarioRepository extends CrudRepository<Funcionario, Long> {

    // criado para a busca Funcionario por id ou chave primária

    Funcionario findById(long id);

    // criado para a busca Funcionario por nome

    Funcionario findByNome(String nome);

    // Busca para vários nomes Funcionários

    @Query(value = "select u from Funcionario u where u.nome like %?1%")

    List<Funcionario> findByNomes(String nome);

}

Por ser uma interface, não implementa código ou lógica, apenas padrões de nomes para os métodos. A lógica e o funcionamento dos métodos são criados pelo JPA, que interage com o banco de dados.

Observe que, na linha 12, ela recebe a classe Funcionario e trabalha essa mesma classe nos 3 métodos descritos na sequência. Por se tratar de uma classe em polimorfismo, que é “filha” da classe CrudRepository, o JPA implementa os métodos já criados dentro do JPA que trabalha com a seguinte regra:

OBJETO\_DE\_RETORNO nome\_do\_método(PARÂMETRO\_DE\_PESQUISA);

Na linha 15, o método findById(long id) retornará um objeto funcionário do banco de dados desde que localize o funcionário com o id passado via parâmetro no banco de dados.

Na linha 18, o método findByNome(String nome) retornará um objeto funcionário do banco de dados desde que nele exista este nome de funcionário.

Note que na linha 22 está sendo usada a anotação @Query, que permite inserir comandos em SQL para condições mais específicas. Na sintaxe do SQL, há a letra “u”, que representa os registros que serão recebidos como resultado da select e, na cláusula where, tem-se a veriricação de “u.nome Like”.

Nas linhas 22 e 23, por fim, tem-se um método que localiza partes do nome, passando por parâmetro o dado em like %?1%. Nesse formato, captam-se letras do nome do funcionário que são recebidas pelo parâmetro String nome. Caso exista mais de um funcionário com partes do nome com as letras informadas, é retornada uma List (lista de funcionários).

**Gravando os Valores no Banco de Dados**

Para gravar os dados da classe funcionario que está dentro do Model em um banco de dados, devem ser construídos uma view e um controller para trabalhar esses dados. A view do funcionário será uma página web, em HTML, com o nome funcionario.html e localizada no endereço “src/main/resources/templates/funcionario”.

A página funcionario.html contém um formulário básico em HTML; no entanto, na tag em realce <form method=”post”>, sem a propriedade “action” definida, o formulário enviará os dados dos campo para o próprio endereço, ou seja, para ela mesmo, e o atributo method=”post” é a forma de envio dos dados. Este formato será mapeado pelo Controller FuncionarioController desta view. O botão submit, por se tratar do front-end, terá apenas a função de envio de dados.

O código da página funcionario.html é:

<!doctype html>

<html lang=”pt-br” xmlns:th="http://thymeleaf.org">

<head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Funcionário</title>

</head>

<body>

    <div class="container">

        <h4>Cadastro de Funcionário</h4>

        <form method="post">

            <div class="form-group">

                <label>Nome</label>

                <br>

                <input type="text" name="nome" id="nome">

                <br>

                <label>Email </label>

                <br>

                <input type="text" name="email" id="email">

                <br><br>

            </div> <button type="submit" class="btn btn-primary">Enviar</button>

        </form>

    </div>

</body>

</html>

Já o Código para o Classe FuncionarioController.java deve ser:

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

@Controller

public class FuncionarioController {

    @Autowired

    private FuncionarioRepository fr;

    @RequestMapping(value = "/funcionario", method = RequestMethod.GET)

    public String abrirfuncionario() {

        return "funcionario/funcionario";

    }

    @RequestMapping(value = "/funcionario", method = RequestMethod.POST)

    public String gravarfuncionario(Funcionario funcionario) {

        fr.save(funcionario);

        return "redirect:/funcionario";

    }

}

Na aplicação em execução para a requisição do usuário no endereço http://localhost:8080/funcionario, aparecerá a view funcionario.html, atendendo à linha 19 @RequestMapping(value = "/funcionario", method = RequestMethod.GET) e acionando o método abrirFuncionario(), que retornará a página a seguir, construída em HTML e que está dentro da pasta “src/main/resources/funcionario”.

O funcionamento desta página se dá ao ser apertado o botão “Enviar”. Com isso, a função submit do botão envia os dados do formulário para o próprio endereço da página (pois o atributo action=’ ’ não foi definido) utilizando o method=”post”. No controlador FuncionarioController, a rota @RequestMapping(value = "/funcionario", method = RequestMethod.POST) será captada e acionará o método gravarFuncionario(Funcionario funcionario), recebendo como parâmetro os dados do formulário.

No controller, a anotação @Autowired instancia a Interface “fr”, que é o repository do funcionario. Este repositório é responsável pela interação do JPA com o banco de dados. O repository foi criado quando se originou o model, como pudemos estudar anteriormente no material sobre Model da classe Funcionario.java.

A interface fr, por ser uma interface da CrudRepository, implementa diversos métodos de interações com o banco de dados, como: delete(), para deletar um registro específico; deleteAll(), para deletar todos os registros; count(), que retorna a quantidade de registros; findAll(), que busca todos os registros; findById(), para buscar o dado pela chave primária; e save(), para salvar ou editar. Essas funções já estão programadas e podem ser utilizadas nas classes do projeto. Ao aplicar a save(), que salva os dados por meio do JPA no banco de dados; a página é redirecionada para a própria página funcionario.html em requisição GET pelo comando return “redirect:/funcionario”.

**Buscando Valores**

Para consultar valores do banco de dados, considere a seguinte mudança na View index.html em “src/main/resources/templates/”. Considere também as expressões em Thymeleaf para impressão dinâmica de dados recebidos pela página via método POST.

<!doctype html>

<html lang=”pt-br” xmlns:th="http://thymeleaf.org">

<head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Buscar</title>

</head>

<body>

    <h1>Buscar</h1>

    <span th:text="${msg}"></span>

    <br>

    <br>

    <form method="post">

        Buscar : <input type="text" name="buscar" id="buscar">

        <button type="submit">Enviar</button>

    </form>

    <div th:if="${#objects.nullSafe(funcionarios, default)}">

        <table class="table table-hover table-responsive w-auto table-striped">

            <thead>

                <tr>

                    <th scope="col">Nome:</th>

                    <th scope="col">E-mail:</th>

                </tr>

            </thead>

            <tbody>

                <tr th:each="funcionario : ${funcionarios}">

                    <td><span th:text="${funcionario.nome}"></span></td>

                    <td><span th:text="${funcionario.email}"></span></td>

                </tr>

            </tbody>

        </table>

    </div>

</body>

</html>

A div com o comando <div th:if="${#objects.nullSafe(funcionarios, default)}"> verifica se, na expressão boolena th:if=”${}”, o objeto funcionários, recebido pelo controller, é válido ou inexistente; caso não tenha valores, ele não imprime a tag div e seu conteúdo.

Dentro da div, a tag table possui a expressão em thymeleaf th:each=”funcionario : ${funcionarios}”. O objeto funcionario da página receberá do controller o objeto ${funcionários} com os resultados da busca. A impressão dos atributos é feita por meio das expressões th:text=”${funcionario.nome}” e th:text=”${funcionario.email}”

Caso o objeto ${funcionarios} não exista, não serão impressos os valores dentro da tag TR; caso tenha como resultado uma lista de funcionários, o objeto repetirá a impressão das linhas em HTML.

Veja a página http://localhost:8080/home a seguir na opção GET, quando somente se chama o endereço pelo browser antes de clicar no botão “Enviar”.

**Considere a seguinte mudança no IndexController e os requestmapping para GET e POST.**

mport org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.RequestParam;

import org.springframework.web.servlet.ModelAndView;

@Controller

public class IndexController {

    @RequestMapping(value = "", method = RequestMethod.GET)

    public ModelAndView abrirIndex() {

        ModelAndView mv = new ModelAndView("index");

        String mensagem = "Olá Seja Bem Vinda(o) !";

        mv.addObject("msg", mensagem);

        return mv;

    }

    @Autowired

    FuncionarioRepository fr;

    @RequestMapping(value = "", method = RequestMethod.POST)

    public ModelAndView buscarIndex(@RequestParam("buscar") String buscar) {

        ModelAndView mv = new ModelAndView("index");

        String mensagem = "Resultado da Busca !";

        mv.addObject("msg", mensagem);

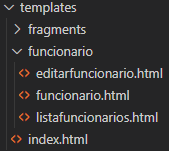
        mv.addObject("funcionarios", fr.findByNomes(buscar));

        return mv;

    }

}

**Listar, excluir e editar registros do banco de dados**



Para continuação do projeto devemos agora criar os seguintes arquivos html

editarfuncionario.html

<!doctype html>

<html lang=”pt-br” xmlns:th="http://thymeleaf.org">

<head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Funcionario</title>

</head>

<body>

    <div class="container">

        <h4>Editar Funcionário</h4>

        <form method="post">

            <div class="form-group" th:each="funcionario : ${funcionario}">

                <label>ID</label> <br> <input type="text" name="nome" id="id" th:value="${funcionario.id}"

                    disabled="true"> <br> <label>Nome</label>

                <br> <input type="text" name="nome" id="nome" th:value="${funcionario.nome}"> <br> <label>Email

                </label> <br> <input type="text" name="email" id="email" th:value="${funcionario.email}"> <br>

                <br>

            </div>

            <button type="submit" class="btn btn-primary">Enviar</button>

        </form>

    </div>

</body>

</html>

E listarfuncionarios.html

<!doctype html>

<html lang=”pt-br” xmlns:th="http://thymeleaf.org">

<head>

    <!-- Required meta tags -->

    <meta charset="utf-8">

    <meta name="viewport" content="width=device-width, initial-scale=1">

    <!-- Bootstrap CSS -->

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet">

    <title>Listar Funcionário</title>

</head>

<body>

    <div class="container">

        <h4>Listar Funcionário</h4>

        <table class="table table-hover table-responsive w-auto table-striped">

            <thead>

                <tr>

                    <th scope="col">Nome:</th>

                    <th scope="col">E-mail:</th>

                </tr>

            </thead>

            <tbody>

                <tr th:each="funcionario : ${funcionarios}">

                    <td><span th:text="${funcionario.nome}"></span></td>

                    <td><span th:text="${funcionario.email}"></span></td>

                    <td><a th:href="@{'/deletarfuncionario/' + ${funcionario.id} }"

                            class="waves-effect waves-light btn-small">

                            <button type="button" class="btn btn-danger">Excluir</button>

                        </a></td>

                    <td> <a th:href="@{'/editarfuncionario/' + ${funcionario.id} }"

                            class="waves-effect waves-light btn-small">

                            <button type="button" class="btn btn-primary">Editar</button>

                        </a>

                    </td>

                </tr>

            </tbody>

        </table>

    </div>

</body>

</html>

Para que as páginas tenham suas funcionalidades atribuídas, devemos criar o controller. O ListaFuncionarioController.java fara a atribuição da funções listar, editar e deletar.

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.servlet.ModelAndView;

@Controller

public class ListaFuncionarioController {

    @Autowired

    private FuncionarioRepository fr;

    @RequestMapping(value = "/lista", method = RequestMethod.GET)

    public ModelAndView listarfuncionario() {

        ModelAndView mv = new ModelAndView("funcionario/listafuncionarios");

        mv.addObject("funcionarios", fr.findAll());

        return mv;

    }

    @RequestMapping(value = "/deletarfuncionario/{id}", method = RequestMethod.GET)

    public String deletarFuncionario(@PathVariable("id") long id) {

        fr.delete(fr.findById(id));

        return "redirect:/lista";

    }

    @RequestMapping(value = "/editarfuncionario/{id}", method = RequestMethod.GET)

    public ModelAndView abrireditarfuncionario(@PathVariable("id") long id) {

        ModelAndView mv = new ModelAndView("funcionario/editarfuncionario");

        mv.addObject("funcionario", fr.findById(id));

        return mv;

    }

    @RequestMapping(value = "/editarfuncionario/{id}", method = RequestMethod.POST)

    public String updateFuncionario(Funcionario funcionario) {

        fr.save(funcionario);

        return "redirect:/lista";

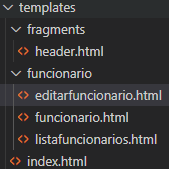
    }

}

Ajuste Finais e Criação de um Cabeçalho.

Criando um Cabeçalho – Para o projeto vamos utilizar o Thymeleaf para desenvolvimento de um cabeçalho único para o projeto.

Para isso vamos criar uma pasta chamada Fragments dentro do Template e vamos criar o arquivo header.html.



<header th:fragment="header">

    <nav class="navbar navbar-expand-lg navbar-light bg-light">

        <div class="collapse navbar-collapse" id="navbarSupportedContent">

            <ul class="navbar-nav mr-auto">

                <li class="nav-item active">

                    <a class="nav-link" href="/">Home</a>

                </li>

                <li class="nav-item">

                    <a class="nav-link" href="/funcionario">Funcionario</a>

                </li>

                <li class="nav-item">

                    <a class="nav-link" href="/lista">Lista</a>

                </li>

            </ul>

        </div>

    </nav>

</header>

Para que o NavBar Funcione em todas as telas do projeto devemos inserir a seguinte DIV nas páginas do projeto

<body>

    <div th:insert="~{fragments/header :: header}"><!--  barra de menu -->...</div>