**SpringBoot com thymeleaf**

<https://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html>

O Thymeleaf é um moderno mecanismo de modelo Java do lado do servidor para ambientes da Web e autônomos, capaz de processar HTML, XML, JavaScript, CSS e até texto simples.

O principal objetivo do Thymeleaf é fornecer uma maneira elegante e altamente sustentável de criar modelos. Para conseguir isso, ele se baseia no conceito de Modelos Naturais para injetar sua lógica em arquivos de modelo de uma forma que não afete o modelo de ser usado como um protótipo de design. Isso melhora a comunicação do design e preenche a lacuna entre as equipes de design e desenvolvimento.

O Thymeleaf também foi projetado desde o início com os Padrões da Web em mente - especialmente **HTML5** - permitindo que você crie modelos totalmente validados, se isso for necessário para você.

<https://www.thymeleaf.org/doc/tutorials/3.0/thymeleafspring.html>

O Thymeleaf oferece um conjunto de integrações Spring que permitem que você o use como um substituto completo para JSP em aplicativos Spring MVC.

Essas integrações permitirão que você:

* Faça com que os métodos mapeados em seus **@Controller**objetos Spring MVC sejam encaminhados para modelos gerenciados pelo Thymeleaf, exatamente como você faz com JSPs.
* Use **Spring Expression Language** (Spring EL) em vez de OGNL em seus modelos.
* Crie formulários em seus modelos totalmente integrados com seus beans de suporte de formulário e associações de resultados, incluindo o uso de editores de propriedades, serviços de conversão e tratamento de erros de validação.
* Exibir mensagens de internacionalização de arquivos de mensagens gerenciados pelo Spring (através dos **MessageSource**objetos usuais).
* Resolva seus templates usando os mecanismos de resolução de recursos próprios do Spring.

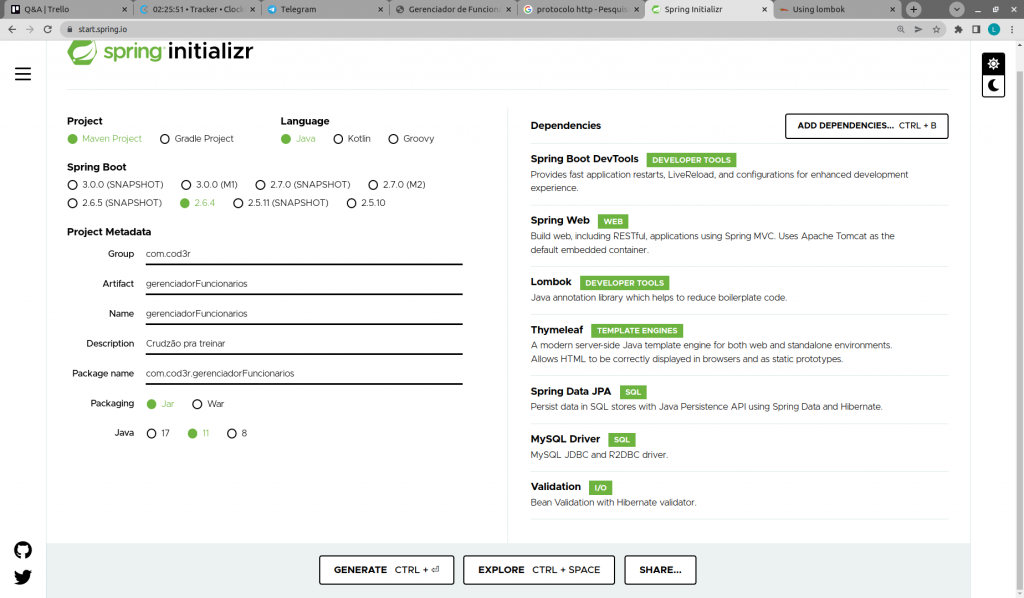
# CRUD com Spring Boot e Thymeleaf

# <https://blog.cod3r.com.br/java-spring-boot/>

Nós vamos construir um gerenciador de funcionários utlizando Spring Boot e PostGresql no back end e Thymeleaf e Bootstrap no front end. A ideia é dar o próximo passo em desenvolvimento web com Java, portanto, é importante que você já conheça Orientação a Objetos, Java, protocolo HTTP, HTML, CSS e ter ou instalar o MySQL na sua máquina. Vamos adiante.

## Iniciando o projeto com Spring Boot

Primeiramente, vamos acessar o página <https://start.spring.io/>, ela vai nos auxilia na criação do projeto.



Siga os seguintes passos: escolha a versão do Java, dê o nome que você quiser e copie as dependências que vamos usar. Em seguida, clique em Generate e um zip será baixado, extraia seus arquivos na pasta de sua escolha. Depois abra seu projeto na sua IDE, eu estou utlizando o IntelliJ Community Edition.

### Configurando o application.properties

Aqui, vamos definir as propriedades da nossa aplicação. Portanto, para isso, vamos abrir a pasta resources, renomear o arquivo **application.properties**  para application.yml e colocar o seguinte código:

**spring:**

**datasource:**

**url:** jdbc:postgresql://localhost:5432/GerenciadorFuncionarios

**username:** postgres

**password:** treinamento

**jpa:**

**show-sql:** true

**hibernate:**

**ddl-auto:** update

**properties:**

**hibernate.format\_sql:** true

**logging:**

**level:**

**org:**

**hibernate:**

**type:** trace

Crie o banco GerenciadorFuncionarios

Agora rode o arquivo **GerenciadorFuncionariosApplication**, e no navegador acesse **localhost:8080**, nossa aplicação já deve estar no ar.

## Spring Boot: Entidade Funcionario

Agora, nós iremos criar uma entidade do tipo **Funcionario**. Para isso crie um pacote chamado **model** dentro do pacote **gerenciadorDeFuncionarios** e dentro do pacote modelo crie uma nova classe chamada Funcionario. Depois coloque esse código dentro da classe:

**package** funtec.gerenciador\_funcionario.model;

**import** java.math.BigDecimal;

**import** java.time.LocalDateTime;

**import** javax.persistence.Entity;

**import** javax.persistence.EnumType;

**import** javax.persistence.Enumerated;

**import** javax.persistence.GeneratedValue;

**import** javax.persistence.GenerationType;

**import** javax.persistence.Id;

**import** lombok.AllArgsConstructor;

**import** lombok.Getter;

**import** lombok.NoArgsConstructor;

**import** lombok.Setter;

@Entity

@Getter @Setter @NoArgsConstructor @AllArgsConstructor

**public** **class** Funcionario {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**private** **int** id;

**private** String nome;

**private** String email;

**private** String cargo;

**private** BigDecimal salario;

**private** LocalDateTime dataContratacao = LocalDateTime.*now*();

@Enumerated(EnumType.***STRING***)

**private** FuncionarioSetor setor;

**public** **void** setSetor(FuncionarioSetor setor) {

**this**.setor = setor;

}

}

Contudo, aqui temos um problema: utilizamos o Lombok para gerar os getters, setters e construtores, e se você quiser fazer igual, precisa configurar a ferramenta na sua IDE: <https://projectlombok.org/setup/overview>. Se não quiser, gere normalmente da forma padrão, mas não esqueça do construtor padrão!

Nosso Funcionário possui um Id gerado e incrementado automaticamente, com detalhes como nome, email, cargo, salário e um setor que é um ENUM. Esses atributos serão colunas na nossa tabela, mas antes precisamos criar o ENUM do setor. Ainda no pacote modelo crie um enum chamado **FuncionarioSetor:**

**package** funtec.gerenciador\_funcionario.model;

**public** **enum** FuncionarioSetor {

***TECNOLOGIA***("tecnologia"),

***RH***("rh"),

***DIRETORIA***("diretoria");

**private** String value;

**private** FuncionarioSetor(String value) {

**this**.value = value;

}

**public** String getSetor() {

**return** value;

}

}

Rode a aplicação e uma tabela chamada funcionario será criada no DB.

## Spring Boot: o repositório de Funcionario

Vamos criar um pacote chamado repositorio dentro do pacote modelo, e dentro dele criar uma interface chamada **FuncionarioRepository**.

Um repositório é uma interface que tem como função fazer a ponte entre nosso banco de dados e a aplicação, através da classe JpaRepository que possui diversos métodos de CRUD prontos para uso, além disso podemos criar nossos próprios métodos específicos como findBySetor, que o Hibernate vai interpretar da maneira que queremos se seguirmos esse padrão de nomenclatura.

**package** funtec.gerenciador\_funcionario.model.repository;

**import** java.util.List;

**import** org.springframework.data.jpa.repository.JpaRepository;

**import** funtec.gerenciador\_funcionario.model.Funcionario;

**import** funtec.gerenciador\_funcionario.model.FunconarioSetor;

**public** **interface** FuncionarioRepository **extends** JpaRepository<Funcionario, Integer> {

List<Funcionario> findBySetor(FunconarioSetor funcionarioSetor);

}

Nosso modelo está pronto, agora vamos para camada de controller.

## Spring Boot: Controllers

Em seguida, ao rodar o projeto, vemos a tela de Whitelabel Error Page no localhost:8080. Isso acontece porque não estamos mostrando nada.

Por isso, vamos criar o pacote controller em funtec.gerenciador\_funcionario.

Portanto, dentro do pacote crie a classe HomeController, que é onde vamos controlar as requisições e respostas da home do nosso projeto. Essa classe é anotada com @Controller, que indica pro Spring Boot o que fazer com ela.

Depois vamos criar um método GET para a url \*\*\*[http://localhost:8080/home\*\*\*](http://localhost:8080/home***), que irá retornar um página html customizada por nós:

**package** funtec.gerenciador\_funcionario.controller;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.GetMapping;

@Controller

**public** **class** HomeController {

@GetMapping("/home")

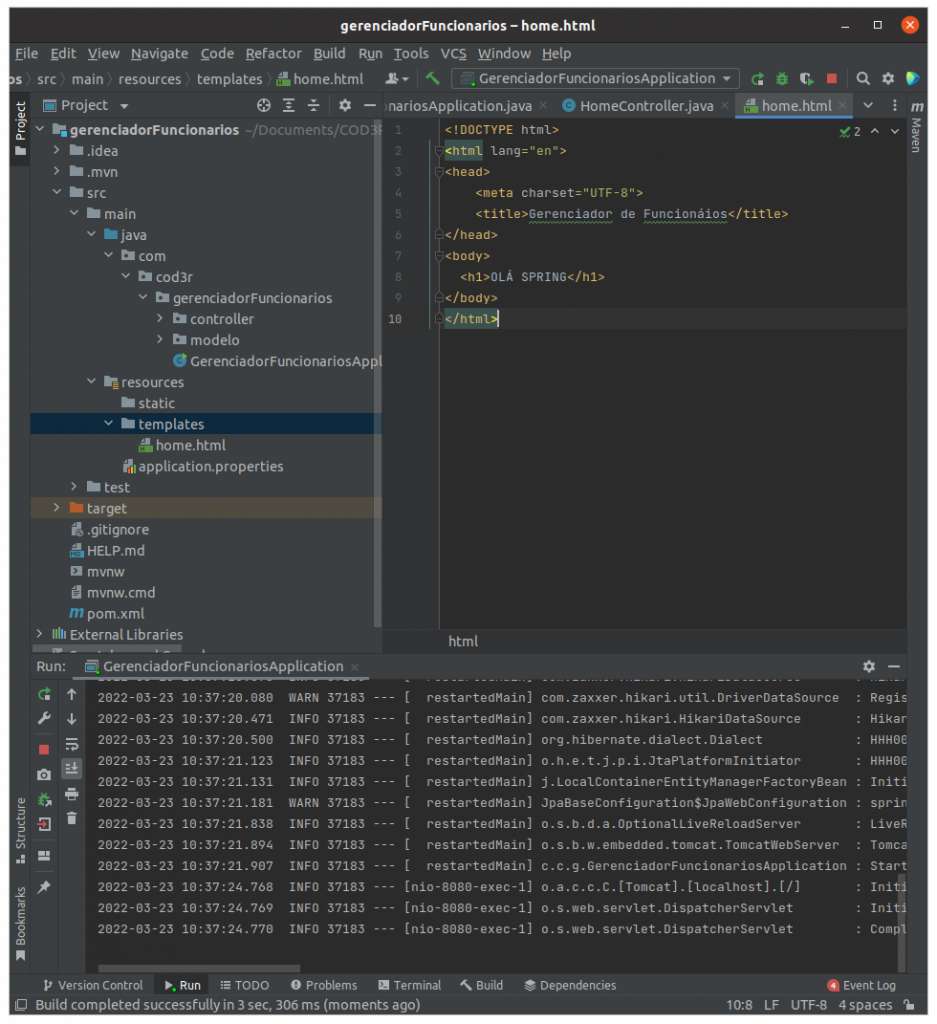
**public** String home() {

**return** "home";

}

}

Percebam que retorno uma String chamada “home”, e o que é esse home? Ele é a página html que iremos retornar, só que para que o Spring encontre essa página, precisamos criar no local correto, dentro de template em resources:



<html lang "en">

<head>

<meta charset="UTF-8">

<title>Gerenciador de Funcionários</title>

<body>

<h1>Ola Spring</h1>

</body>

</head>

</html>

Agora reinicie a aplicação e o browser deve mostrar **OLÁ SPRING** no nosso endpoint **/home.**

E se agora quisermos mostrar os funcionários nessa página? Para isso, criamos outro controller, o FuncionarioController, para cuidar dos endpoints que envolvem o CRUD de funcionários em si.

### FuncionarioController

Após criar o FuncionarioController, coloque esse código dentro da classe:

**package** funtec.gerenciador\_funcionario.controller;

**import** javax.validation.Valid;

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

**import** org.springframework.stereotype.Controller;

**import** org.springframework.validation.BindingResult;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PostMapping;

**import** funtec.gerenciador\_funcionario.model.Funcionario;

**import** funtec.gerenciador\_funcionario.model.repository.FuncionarioRepository;

@Controller

**public** **class** FuncionarioController {

@Autowired

FuncionarioRepository funcionarioRepository;

// Acessa o formulario

@GetMapping("/form")

**public** String funcionariosForm(Funcionario funcionario) {

**return** "addFuncionariosForm";

}

// Adiciona novo funcionario

@PostMapping("/add")

**public** String novo(@Valid Funcionario funcionario, BindingResult result) {

**if** (result.hasFieldErrors()) {

**return** "redirect:/form";

}

funcionarioRepository.save(funcionario);

**return** "redirect:/home";

}}

A anotação Autowired irá injetar as dependências do nosso repositório na classe.

Temos um GET para “/form” e um POST que vai bater no endpoint definido no nosso forms(/add).

Posteriormente, seguindo a mesma linha, vamos criar os métodos para atualizar um funcionário:

// Acessa o formulario de edição

@GetMapping("form/{id}")

public String updateForm(Model model, @PathVariable(name = "id") int id) {

Funcionario funcionario = funcionarioRepository.findById(id)

.orElseThrow(() -> new IllegalArgumentException("Invalid user Id:" + id));

model.addAttribute("funcionario", funcionario);

return "atualizaForm";

}

// Atualiza funcionario

@PostMapping("update/{id}")

public String alterarProduto(@Valid Funcionario funcionario, BindingResult result, @PathVariable int id) {

if (result.hasErrors()) {

return "redirect:/form";

}

funcionarioRepository.save(funcionario);

return "redirect:/home";

}

Para isso, no primeiro método do tipo GET acessamos um form, mas com um id pra identificar que funcionário que vamos editar. Esse id é uma @PathVariable(variável que está na URL).

Depois vamos utilizar o repositório para buscar um funcionário pelo id dele, passando o id da **PathVariable**. Percebam que o método **findById**  está naquele **JpaRepository**.

Beleza, encontramos o funcionário com tal id, e agora? Aqui nós usamos aquele Model passado nos parâmetros. Em seguida precisamos passar esse funcionário em questão lá pro nosso html para o usuário editar, e para isso fazemos um **model.addAttribute**, onde o primeiro parâmetro é o nome que vou usar lá no html e o segundo parâmetro o objeto que estou mandando. No formulário estarão as informações e em breve vamos aprofundar nessa parte, onde conheceremos o tal do **Thymeleaf**.

Tudo certo até agora, mas um dos funcionários foi pego roubando e queremos excluir ele do sistema. E adivinhem, já existe um método no nosso repositório pronto para isso:

@GetMapping("delete/{id}")

**public** String delete(@PathVariable(name = "id") **int** id, Model model) {

Funcionario funcionario = funcionarioRepository.findById(id)

.orElseThrow(() -> **new** IllegalArgumentException("Invalid user Id:" + id));

funcionarioRepository.delete(funcionario);

**return** "redirect:/home";

}

Portanto, só precisamos passar o id na url, buscar o funcionário e excluí-lo.

Nosso FuncionarioRepository está pronto, e agora que já temos o que controla os funcionários, é hora de mexer no HomeController,

### HomeController

Lembram-se do método home que retorna “Olá Spring”? Vamos alterá-lo para trazer os funcionários. Veja como é simples:

@Autowired

private FuncionarioRepository funcionarioRepository;

@GetMapping("/home")

public String home(Model model) {

List<Funcionario> funcionarios = funcionarioRepository.findAll();

model.addAttribute("funcionarios", funcionarios);

return "home";

}

* Primeiro o método do tipo GET que recebe um Model;
* Depois criamos uma lista de funcionários, que terá o retorno do método **findAll** do repositório;
* E por fim, adicionamos essa lista no modelo, para enviar ao html **home** com o nome de “funcionarios”.

Esse primeiro controller traz todos os funcionários. No entanto, queremos trazer também por setor:

@GetMapping

public String setor(@RequestParam String setor, Model model) {

FuncionarioSetor funcionarioSetor = FuncionarioSetor.valueOf(setor.toUpperCase());

List<Funcionario> funcionarios = funcionarioRepository.findBySetor(funcionarioSetor);

model.addAttribute("funcionarios", funcionarios);

return "home";

}

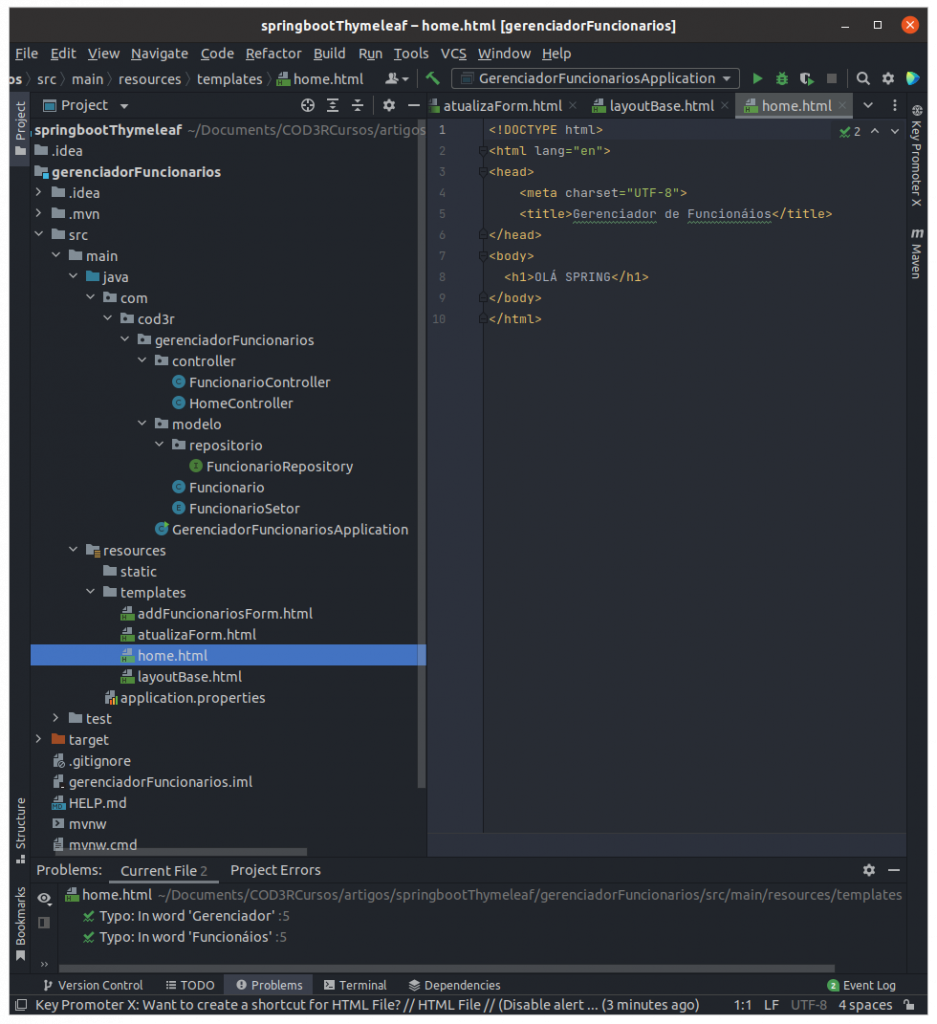
* O @RequestParam é uma anotação que indica que vamos passar o valor dessa variável na seguinte forma: \*\*\*[http://localhost:8080/?setor=tecnologia\*\*\*](http://localhost:8080/?setor=tecnologia***)
* Aqui, passamos também um Model;
* Em seguida pegamos o setor passado na URL e atribuímos a um setor do tipo FuncionarioSetor;
* Depois declaramos uma lista de funcionarios que receberá os funcionarios encontrados no findBySetor;
* E por fim, adicionamos a lista ao model.

E agora vamos partir para o front end da aplicação.

## Front End

No front, como dito anteriormente, vamos utilizar uma ferramenta chamada Thymeleaf, um renderizador de páginas que roda no lado do servidor.

Antes de vermos onde e como utilizar essa ferramenta, crie os seguintes arquivos em template:



### Layout Base

Vamos começar pelo arquivo **layoutBase.html**, que irá guardar código repetido, como o header, por exemplo.

<!--suppress ALL -->

<head th:fragment="head">

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>Gerenciador de Funcionários</title>

<link

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

rel="stylesheet"

integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"

crossorigin="anonymous">

<link rel="stylesheet"

href="<https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css>">

<link

href="<https://fonts.googleapis.com/css2?family=Roboto:wght@500&display=swap>"

rel="stylesheet">

<style>

.nav-link {

font-family: 'Roboto', sans-serif;

color: black;

font-size: 1.5em;

justify-content: flex-end;

margin-right: 15px;

}

.buttons {

width: 8em;

height: auto;

}

body {

background: linear-gradient(90deg, #efd5ff 0%, #515ada 100%);

}

label {

font-family: 'Roboto', sans-serif;

font-size: 1.5em;

}

.logo {

margin-left: 15px;

}

</style>

</head>

<div th:fragment="navbar">

<nav

class="navbar navbar-expand-lg mb-3 d-flex justify-content-between">

<a class="logo" href="/home"><img

src="<https://import.cdn.thinkific.com/220759%2Fcustom\_site\_themes%2Fid%2FhbIbe45tSJSfztUcZ1Qm\_LOGOTIPO-CODER-FUNDOTRANSPARENTE-PRETA.png?width=384&dpr=1>"

alt="" width="300" height="60" class="d-inline-row align-text-top"></a>

<ul class="nav">

<li class="nav-item"><a class="nav-link active"

aria-current="page" href="/home">Home</a></li>

<li class="nav-item"><a class="nav-link"

href="/?setor=tecnologia">Tecnologia</a></li>

<li class="nav-item"><a class="nav-link"

href="/?setor=diretoria">Diretoria</a></li>

<li class="nav-item"><a class="nav-link" href="/?setor=rh">Recursos

Humanos</a></li>

</ul>

</nav>

</div>

<div th:fragment="form">

<div class="form-group">

<label for="nome" class="form-label">Nome</label>

<input class="form-control" type="text" th:field="\*{nome}" id="nome" placeholder="Nome Completo" >

</div>

<div class="form-group">

<label for="inputEmail" class="form-label">Email</label> <input

th:field="\*{email}" type="email" class="form-control" id="inputEmail" required>

</div>

<div class="form-group">

<label for="inputCargo" class="form-label">Cargo</label> <input

th:field="\*{cargo}" type="text" class="form-control" id="inputCargo" required>

</div>

<div class="form-group">

<label for="inputSalario" class="form-label">Salario</label> <input

th:field="\*{salario}" type="number" step="any" class="form-control"

id="inputSalario" placeholder="R$1000.00" required>

</div>

<div class="col-4">

<label for="inputState" class="form-label">Setor</label> <select

th:field="\*{setor}" id="inputState" class="form-select">

<option value="TECNOLOGIA">Tecnologia</option>

<option value="RH">RH</option>

<option value="DIRETORIA">Diretoria</option>

</select>

</div>

</div>

Para começar, declaramos uma div com a propriedade th:fragment do thymeleaf. Passando o nome desse fragmento e tudo que estiver dentro desse fragmento, poderemos usar em outras páginas. Temos o fragmento head, navbar e form. No form usamos o **th:field**, que recebe o nome de um atributo do modelo, exemplo: **th:field=”\*{nome}”**.

### Home

Depois copie esse código no arquivo home.html:

<head th:replace="~{layoutBase :: head}">

</head>

<body>

<div th:replace="~{layoutBase :: navbar}"></div>

<div class="container mb-2">

<h1>Funcionários</h1>

<table class="table table-light">

<tr class="table-dark">

<th></th>

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

<th scope="col">Cargo</th>

<th scope="col">Email</th>

<th scope="col">Salario</th>

<th scope="col">Setor</th>

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

</tr>

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

<th scope="row"></th>

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

<td><span th:text="${func.cargo}"></span></td>

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

<td>R$<span th:text="${func.salario}"></span></td>

<td><span th:text="${func.setor}"></span></td>

<td class="buttons "><a th:href="@{form/{id}(id=${func.id})}"

th:method="get" class="btn btn-warning"><i class="fa fa-pencil"></i></a>

<a th:href="@{delete/{id}(id=${func.id})}" th:method="delete"

class="btn btn-danger"><i class="fa fa-trash"></i></a></td>

</tr>

</table>

<a href="/form"><button class="btn btn-dark">Adicionar

funcionario</button></a>

</div>

</body>

</html>

Em seguida, para pegar os trechos(fragmentos) do layout base, usamos o **th:replace**, passando o nome do fragmento que queremos por ali.

Criamos uma tabela pros funcionários, com os atributos definidos anteriormente. Para popular essa tabela, fazemos um **th:each** que itera sobre a lista(funcionarios) que passamos no modelo, lembram? Portanto, para cada funcionário, extraímos os nomes dos atributos com **th:text**, e para acessá-los é só fazer **func**(como chamei cada funcionario no th:each) **.*atributo.***

Abaixo temos **th:href** e **th:method** para configurar os endpoints que os botões de delete e update vão bater.

### addFuncionariosForm

<!DOCTYPE html>

<html>

<head th:replace="~{layoutBase :: head}">

</head>

<body>

<div th:replace="~{layoutBase :: navbar}"></div>

<div class="container">

<h1 class="mb-3">Cadastrar novo funcionario</h1>

<form action="#" th:object="${funcionario}" th:action="@{/add}" method="post">

<div th:replace="~{layoutBase :: form}"></div>

<div class="col-12 mt-2">

<button type="submit" class="btn btn-dark">Cadastrar</button>

</div>

</form>

</div>

</body>

</html>

No form temos o **th:object** para definir o objeto que vamos postar com o method **POST** no endpoint **/add** que definimos com **th:action.**

### atualizaForm

<!DOCTYPE html>

<html lang="en">

<head th:replace="~{layoutBase :: head}">

</head>

<body>

<div th:replace="~{layoutBase :: navbar}"></div>

<div class="container">

<h1 class="mb-3">Atualizar informacoes</h1>

<form action="/home" th:object="${funcionario}" th:action="@{/update/{id}(id=${funcionario.id})}" method="post">

<div th:replace="~{layoutBase :: form}"></div>

<div class="col-12 mt-2">

<button type="submit" class="btn btn-dark">Atualizar</button>

</div>

</form>

</div>

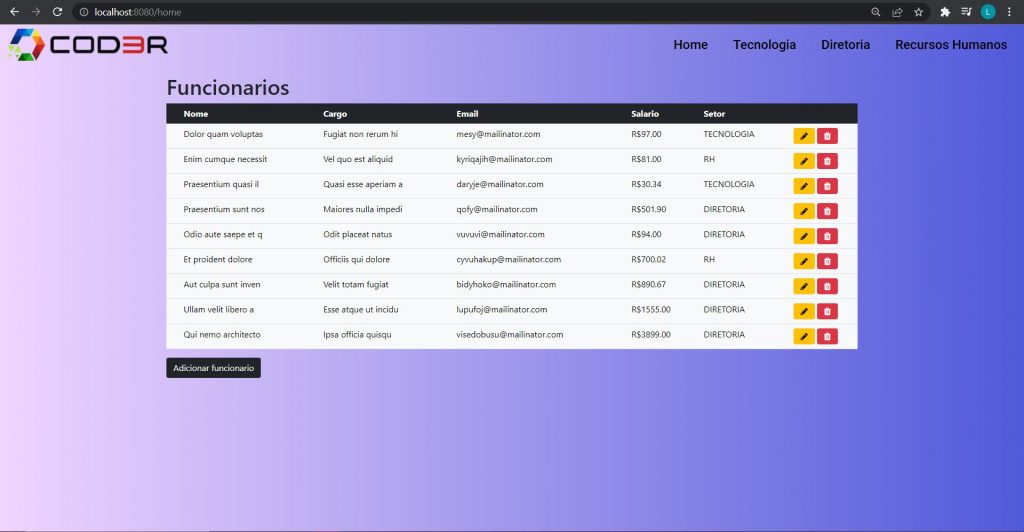
</body>

</html>

Mesma ideia do addFuncionario, porém vamos usar um id já existente no th:action.

## Conclusão

Por fim, rode a aplicação. Tudo já deve estar funcionando como esperado. Mas se não estiver, não hesite em deixar um comentário. E no caso de dúvidas também.



Você pode encontrar esse projeto no seguinte repositório do GitHub: <https://github.com/lorenz075/employee-management>.