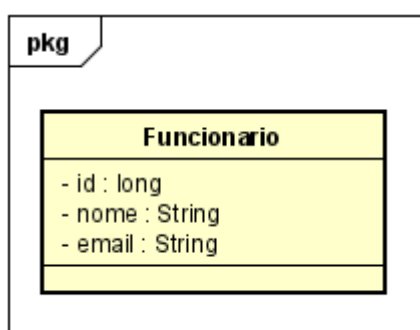


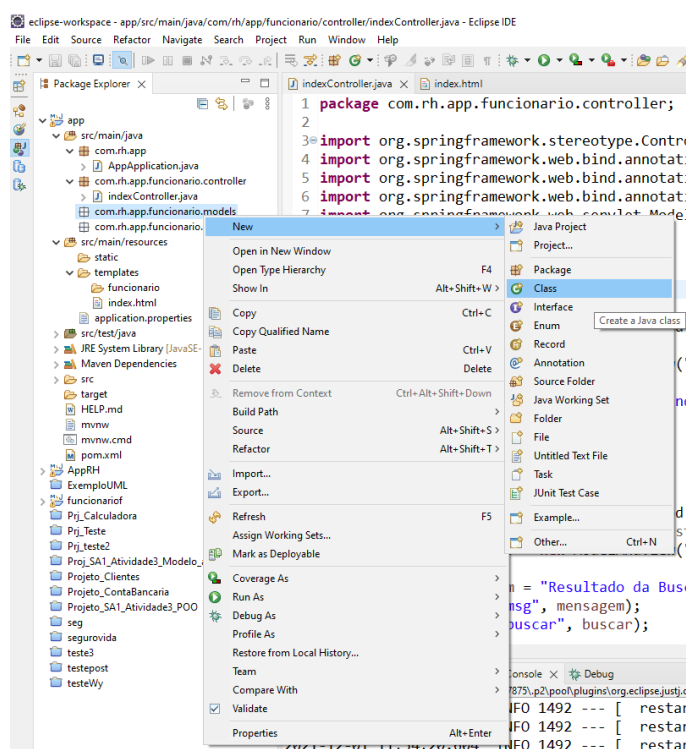
Model

O **model** é a camada do MVC responsável pelos dados e pela classe dos dados. Nesta etapa, a aplicação trabalhará o cadastro de funcionários e, para tanto, serão desenvolvidos o Model Funcionario, num primeiro momento desse cadastro, a View e o Controller, na sequência.

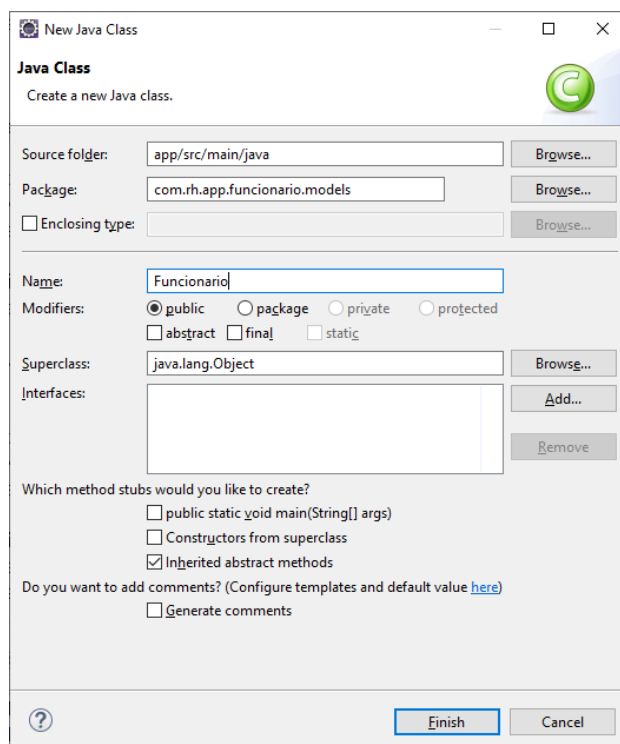
O model será criado de acordo com o **diagrama de classes** a seguir, no qual o nome da classe será **Funcionario** e terá os **atributos** id, nome e email.



Na IDE Eclipse, para inserir a classe **Funcionario** no pacote model, é preciso clicar com o botão direito do mouse no pacote "**com.rh.app.funcionario.models**", depois, em "**New**" e, por fim, em "**Class**".



O nome da classe será **Funcionario**. Clique em “**Finish**”.



Após a criação da classe, será gerado o código a seguir.

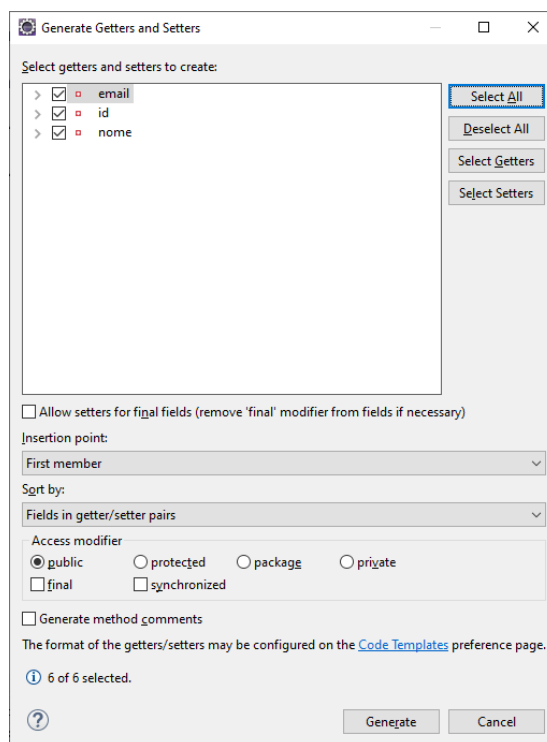
```
package com.rh.app.funcionario.models;  
  
public class Funcionario {  
  
}
```

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

```
package com.rh.app.funcionario.models;  
  
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.

A IDE Eclipse consegue gerar os Getter e Setter para cada atributos de modo automático; lembre-se de que cada atributo que estiver em modo private não pode ser visualizado por outras classes. No Menu **Source**, acesse **Generate Getters and Setters** e, na janela, Generate Getters and Setters, clique em “Select All” e, depois, em “Generate”, para selecionar todos os atributos e gerar métodos públicos.



Nesse processo, a classe Funcionario possuirá 36 linhas com os métodos getters e setters para cada atributo.

```
private long id;

public long getId() {
    return id;
}

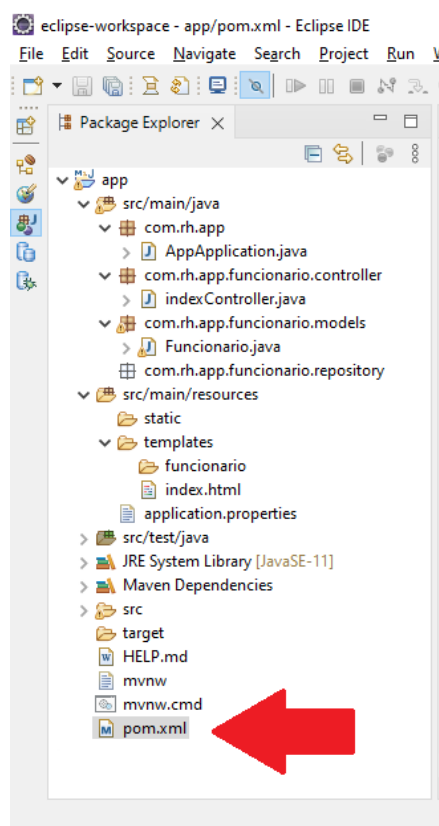
public void setId(long id) {
    this.id = id;
}
```

No método **setId()**, o parâmetro id é recebido via argumento e enviado para a variável id. No método **getId()**, é retornado o valor do atributo id. Nesse formato, caso seja passado algum valor incorreto, este poderá ser tratado no método, não permitindo o envio dos dados.

Conexão com Banco de Dados

É preciso fazer algumas configurações no **pom.xml** e no **application.properties** para que o model funcione corretamente.

Observe, na estrutura de pasta, os diretórios **com.rh.app.funcionario.models** e **com.rh.app.funcionario.repository**. Este último é um pacote que contém as interfaces JPA que serão estudadas posteriormente.



Será necessário editar o arquivo **pom.xml** para a conexão com o banco de dados em MySQL e a utilização do modo JPA com Hibernate. O arquivo **pom.xml** ficará da seguinte maneira, considerando os realces para inserir as dependências:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.5.6</version>
    <relativePath /> <!-- lookup parent from repository -->
  </parent>
```

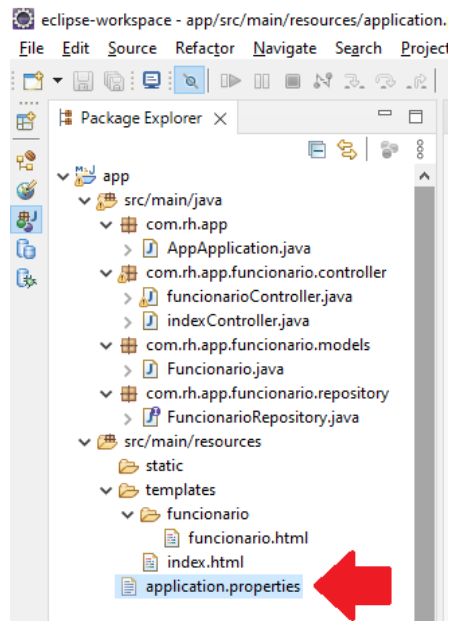


```
<groupId>com.rh</groupId>
<artifactId>app</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>app</name>
<description>Aplicação Spring para Sistema de RH</description>
<properties>
  <java.version>11</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>
```

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



O arquivo ficará com os seguintes dados:

```
spring.datasource.url=jdbc:mysql://localhost:3306/appfunc
spring.datasource.username=root
spring.datasource.password=
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.database-platform = org.hibernate.dialect.MariaDBDialect
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto = update
```

Nesse exemplo, considere que as credenciais do banco de dados são **root** (login) e senha **em branco**; está sendo utilizado o servidor MySQL do XAMPP, mas poderia ser do MySQL Server Community. A database que será criada dentro do servidor de banco de dados MySQL será a **appfunc**.

Entidades

Após a implantação das dependências JPA e do MySQL, é necessário inserir, na classe Model, **anotações** para que as dependências tenham efeito na classe escolhida e esta, conseqüentemente, seja gravada no banco de dados. O primeiro item a ser inserido é a anotação **@Entity** (linha 10) e insere-se o pacote **javax.persistence** (linha 4) da dependência JPA.

```
1 package com.rh.app.funcionario.models;
2
3 import java.io.Serializable;
4 import javax.persistence.Entity;
5 import javax.persistence.GeneratedValue;
6 import javax.persistence.GenerationType;
7 import javax.persistence.Id;
8
```




```
9 /** @author Rolfi Luz - Senai * */
10 @Entity
11 public class Funcionario implements Serializable{
12
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private long id;
18
19     private String nome;
20
21     private String email;
22
23
24     public long getId() {
25         return id;
26     }
27
28     public void setId(long id) {
29         this.id = id;
30     }
31
32     public String getNome() {
33         return nome;
34     }
35
36     public void setNome(String nome) {
37         this.nome = nome;
38     }
39
40     public String getEmail() {
41         return email;
42     }
43
44     public void setEmail(String email) {
45         this.email = email;
46     }
47
48
49 }
```

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 11 `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`.

Nas linhas de 13 a 17, mostra-se 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.IDENTITY)** 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**.

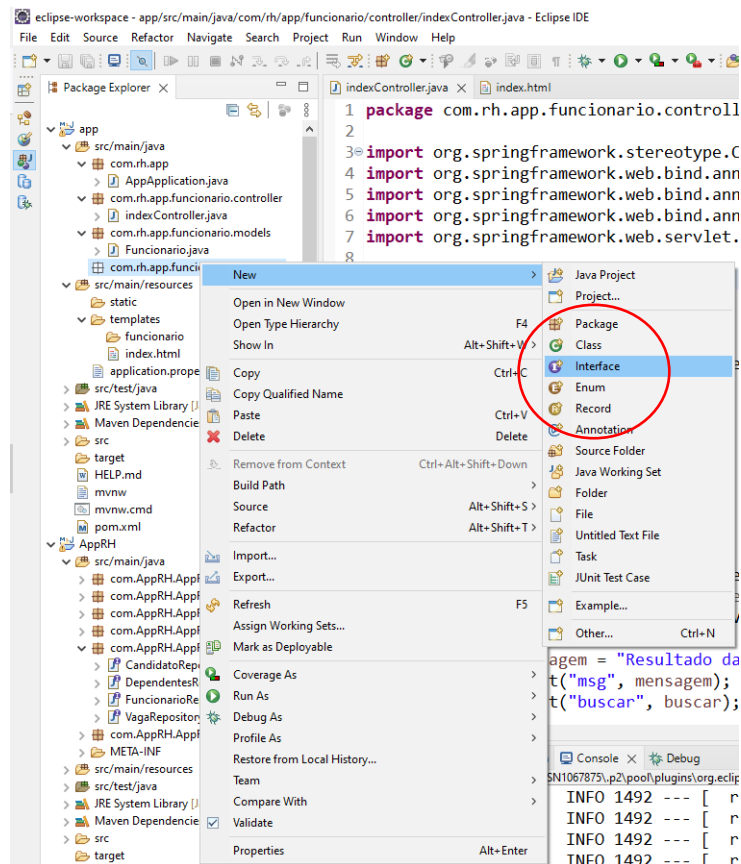
```
13     private static final long serialVersionUID = 1L;
14
15     @Id
16     @GeneratedValue(strategy = GenerationType.IDENTITY)
17     private long id;
```

Os atributos **nome** e **email**, das linhas 19 e 21, por padrão serão as colunas da tabela **Funcionario** que o JPA criará, e o acesso a esses atributos se darão via getters e setters.

```
19     private String nome;
20
21     private String email;
```

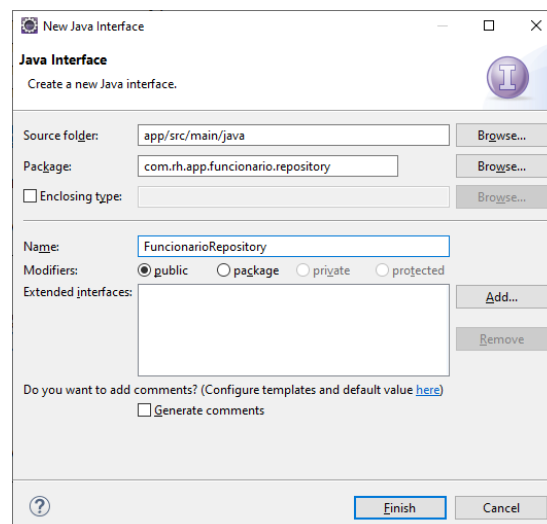
JPA Repository

O repositório é uma forma de trabalhar com os dados que mapeiam os atributos da classe que contém a anotação **@Entity** e que constroem um repositório ou acesso aos dados por meio da interface **CrudRepository**. Essa interface é utilizada pelo JPA para fazer a tradução entre a classe em Java e o Banco de Dados em SQL no servidor MySQL. No pacote **com.rh.app.repository**, clique com o botão direito do mouse em **"New"** e em **"Interface"**, conforme imagem abaixo. Um erro muito comum ao criar um repository é criar uma classe ao invés de interface para o padrão do framework.



Na imagem abaixo na janela “new Java Interface” o campo name da janela “**FuncionarioRepository**” e **Finish**. As interfaces que são repository por padrão seguem o nome do model ao qual fazem o intermédio entre a classe o JPA.

Como o nome do model é **funcionario**, seu repositório será **FuncionarioRepository**; caso fosse **dependente**, seu repositório seria **dependenteRepository**, como nome, esse padrão faz parte das boas práticas para os pacotes Repository.





Ao criar a **Interface** FuncionarioRepository, o código a seguir é gerado:

```
package com.rh.app.funcionario.repository;

public interface FuncionarioRepository {

}
```

A seguir, temos o código com a implementação da Interface FuncionarioRepository:

```
1 package com.rh.app.funcionario.repository;
2
3 import java.util.List;
4
5 import org.springframework.data.jpa.repository.Query;
6 import org.springframework.data.repository.CrudRepository;
7
8 import com.rh.app.funcionario.models.Funcionario;
9
10 /** @author Rolfi Luz - Senai * */
11
12 public interface FuncionarioRepository extends CrudRepository<Funcionario, Long> {
13
14     // criado para a busca Funcionario por id ou chave primária
15     Funcionario findById(long id);
16
17     // criado para a busca Funcionario por nome
18     Funcionario findByName(String nome);
19
20     // Busca para vários nomes Funcionários
21
22     @Query(value = "select u from Funcionario u where u.nome like %?1%")
23     List<Funcionario> findByName(String nome);
24
25 }
```

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 **findByName(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 verificação de “u.nome Like”.

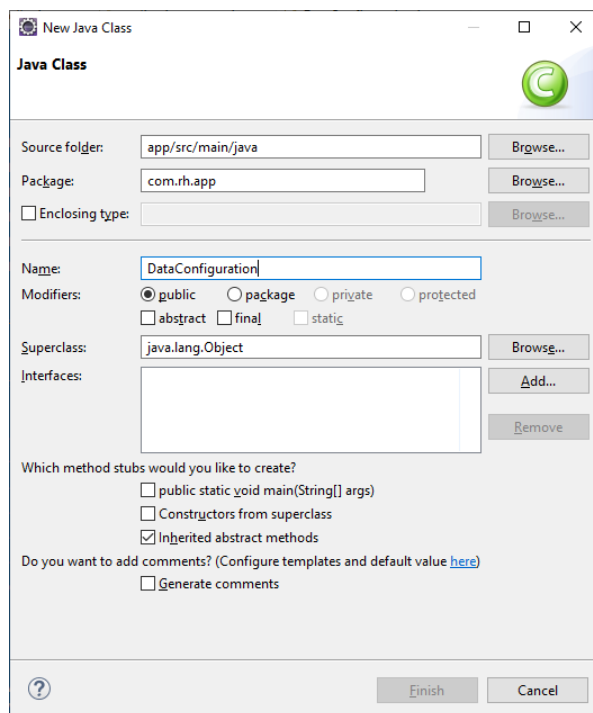
Nas linhas 22 e 23, por fim, tem-se um método que localiza partes do nome, passando por parâmetro o dado em like `%?1%`. Nesse formato, captam-se letras do nome do funcionário que são recebidas pelo parâmetro String **nome**. Caso exista mais de um funcionário com partes do nome com as letras informadas,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

```
22 @Query(value = "select u from Funcionario u where u.nome like %?1%")
23 List<Funcionario> findByNomes(String nome);
```

Assim, nesse código de interface são aplicados os métodos que serão usados nos controllers para buscar os dados e retornar as views. Dentro dos controllers serão instanciadas as interfaces do pacote repository.

Saiba mais

Em alguns casos, para que o Banco de Dados no Spring Boot (dependendo do computador e da IDE Eclipse instalada) funcione, é preciso criar uma classe de Configuração. Para tanto, é preciso criar uma classe no pacote **com.rh.app**, com nome **DataConfiguration**, e clicar em “OK”.



A **DataConfiguration** deve ter o seguinte código:



```
1 package com.rh.app;
2
3 import javax.sql.DataSource;
4
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.jdbc.datasource.DriverManagerDataSource;
8 import org.springframework.orm.jpa.JpaVendorAdapter;
9 import org.springframework.orm.jpa.vendor.Database;
10 import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
11
12 /** @author Rolfi Luz - Senai * */
13
14 @Configuration
15 public class DataConfiguration {
16     @Bean
17     public DataSource dataSource() {
18         DriverManagerDataSource dataSource = new DriverManagerDataSource();
19         dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
20         dataSource.setUrl("jdbc:mysql://localhost:3306/appfunc?useTimezone=true&serverTimezone=UTC");
21         dataSource.setUsername("root");
22         dataSource.setPassword("");
23         return dataSource;
24     }
25
26     @Bean
27     public JpaVendorAdapter jpaVendorAdapter() {
28         HibernateJpaVendorAdapter adapter = new HibernateJpaVendorAdapter();
29         adapter.setDatabase(Database.MYSQL);
30         adapter.setShowSql(true);
31         adapter.setGenerateDdl(true);
32         adapter.setDatabasePlatform("org.hibernate.dialect.MariaDBDialect");
33         adapter.setPrepareConnection(true);
34         return adapter;
35     }
36
37 }
```

Neste formato, as anotações `@Bean` fazem com que os métodos `jpaVendorAdapter()` e `dataSource()` retornem esses objetos para controle do Spring, criando conexões distintas para cada computador que acessa o sistema. Observe que os elementos que estão inseridos no **application.properties** são os mesmos que estão configurados de forma manual nesta classe. Na linha 14, é informado ao Spring que essa classe trata de uma configuração personalizada do banco de dados.



Gravando 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>
```

O código-fonte do controller **funcionarioController.java**, que será inserido no endereço “com.rh.app.funcionario.controller”, é:



```
1 package com.rh.app.funcionario.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.servlet.ModelAndView;
8
9 import com.rh.app.funcionario.models.Funcionario;
10 import com.rh.app.funcionario.repository.FuncionarioRepository;
11
12 /** @author Rolfi Luz - Senai * */
13 @Controller
14 public class funcionarioController {
15
16     @Autowired
17     private FuncionarioRepository fr;
18
19     @RequestMapping(value = "/funcionario", method = RequestMethod.GET)
20     public String abrirfuncionario() {
21         return "funcionario/funcionario";
22     }
23
24     @RequestMapping(value = "/funcionario", method = RequestMethod.POST)
25     public String gravarfuncionario(Funcionario funcionario) {
26         fr.save(funcionario);
27         return "redirect:/funcionario";
28     }
29 }
30 }
```

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”.

Funcionário

localhost:8080/funcionario

Importar favoritos... Introdução

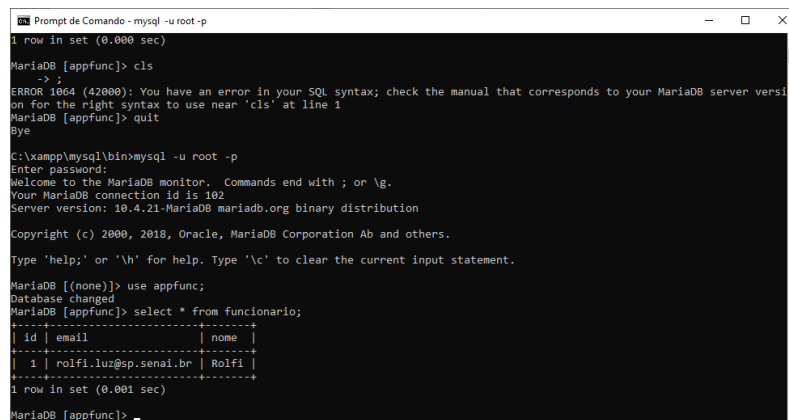
Cadastro de Funcionário

Nome
Rolfi

Email
rolfi.luz@sp.senai.br

Enviar

Ao digitar os dados nos campos e clicar no botão Enviar, os dados serão automaticamente persistidos (gravados) no banco de dados, podendo a sua gravação, inclusive, fazer a verificação no próprio banco.



```
Prompt de Comando - mysql -u root -p
1 row in set (0.000 sec)

MariaDB [appfunc]> cls
>
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'cls' at line 1
MariaDB [appfunc]> quit
Bye

C:\xampp\mysql\bin>mysql -u root -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 102
Server version: 10.4.21-MariaDB mariadb.org binary distribution

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]> use appfunc;
Database changed
MariaDB [appfunc]> select * from funcionario;
+----+-----+-----+
| id | email | nome |
+----+-----+-----+
| 1  | rolf_luz@sp.senai.br | Rolfi |
+----+-----+-----+
1 row in set (0.001 sec)

MariaDB [appfunc]>
```

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.

```
24 @RequestMapping(value = "/funcionario", method = RequestMethod.POST)
25 public String gravarfuncionario(Funcionario funcionario) {
```

Na linha 16 do 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**.

```
16 @Autowired
17 private FuncionarioRepository fr;
```

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. A linha 26 aplica a **save()**, que salva os dados por meio do JPA no banco de dados; após a execução da linha 26, a página é redirecionada para a própria página **funcionario.html** em requisição **GET** pelo comando `return "redirect:/funcionario"`.

```
26 fr.save(funcionario);
27 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">
        <u>Buscar</u> : <input type="text" name="buscar" id="buscar">
        <button type="submit"><u>Enviar</u></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"><u>Nome</u></th>
                <th scope="col"><u>E-mail</u></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="${#objects.nullSafe(funcionarios, default)}">` verifica se, na expressão booleana `th:if="${}"`, o objeto **funcionários**, recebido pelo controller, é válido ou inexistente; caso não tenha valores, ele não imprime a tag `div` e seu conteúdo.

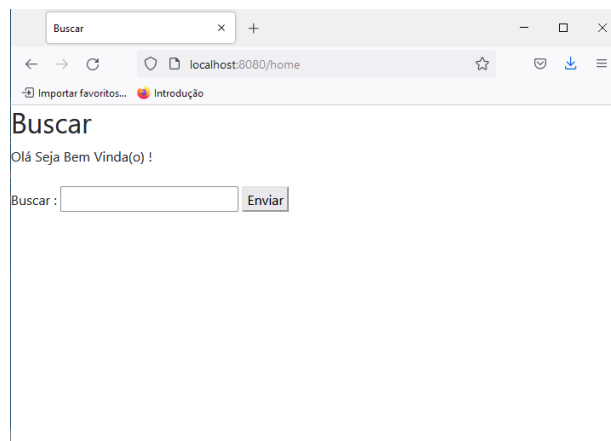


Dentro da div, a tag table possui a expressão em thymeleaf **th:each="funcionario : \${funcionarios}"**. O objeto **funcionario** da página receberá do controller o objeto **\${funcionarios}** com os resultados da busca. A impressão dos atributos é feita por meio das expressões **th:text="\${funcionario.nome}"**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

```
<tr th:each="funcionario : ${funcionarios}">
    <td><span th:text="${funcionario.nome}"></span></td>
    <td><span th:text="${funcionario.email}"></span></td>
</tr>
```

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.

```
1 package com.rh.app.funcionario.controller;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4 import org.springframework.stereotype.Controller;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RequestMethod;
7 import org.springframework.web.bind.annotation.RequestParam;
8 import org.springframework.web.servlet.ModelAndView;
9
10 import com.rh.app.funcionario.repository.FuncionarioRepository;
11
12 /** @author Rolfi Luz - Senai * */
13 @Controller
14 public class indexController {
15
16     @RequestMapping(value = "/home", method = RequestMethod.GET)
17     public ModelAndView abrirIndex() {
18         ModelAndView mv = new ModelAndView("index");
19
20         String mensagem = "Olá Seja Bem Vinda(o) !";
```



```
21         mv.addObject("msg", mensagem);
22
23         return mv;
24     }
25
26     /*
27     * @RequestMapping(value = "/home", method = RequestMethod.POST) public
28     * ModelAndView buscarIndex(@RequestParam("buscar") String buscar) {
29     * ModelAndView mv = new ModelAndView("index");
30     *
31     * String mensagem = "Resultado da Busca !"; mv.addObject("msg", mensagem);
32     * mv.addObject("buscar", buscar);
33     *
34     * return mv; }
35     */
36
37     @Autowired
38     FuncionarioRepository fr;
39
40     @RequestMapping(value = "/home", method = RequestMethod.POST)
41     public ModelAndView buscarIndex(@RequestParam("buscar") String buscar) {
42         ModelAndView mv = new ModelAndView("index");
43
44         String mensagem = "Resultado da Busca !";
45
46         mv.addObject("msg", mensagem);
47         mv.addObject("funcionarios", fr.findByNomes(buscar) );
48
49         return mv;
50     }
51 }
```

O método **abrirIndex()** será acionado quando o usuário solicitar o endereço `http://localhost:8080/home`; após o usuário preencher o campo busca e clicar em enviar, a requisição será para a própria página. Na chamada pelo método POST é que será acionado o método **buscarIndex()**, recebendo como parâmetro a variável “**buscar**”, que será recebida da view e tratada neste controller.

A variável **buscar** contém o valor que o usuário digitou para ser localizado no banco de dados, e o valor desta variável será passado como parâmetro para o método **findByNomes()**, programado dentro do repository **FuncionarioRepository** fr. No repository fr, criado com o model, temos as linhas que procurarão o valor que o usuário digitou na view (retornando uma lista).

```
21     @Query(value = "select u from Funcionario u where u.nome like %?1%")
22     List<Funcionario> findByNomes(String nome);
```

Na linha 46, dentro do controller **indexController**, é passado o resultado da busca para a variável **funcionarios**, para ser passado para a view por meio do método **mv.addObject()**.

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



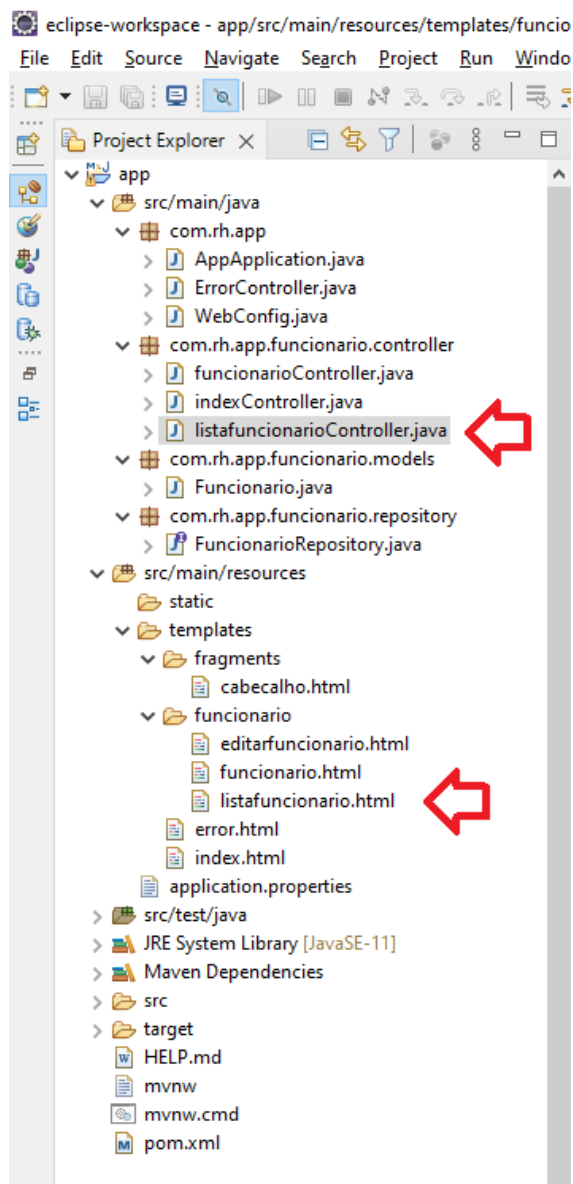
Assim, retornam para a busca os valores que são gravados no banco de dados. A seguir, temos um exemplo de resultado para um registro já inserido no banco de dados.

The screenshot shows a web browser window with the address bar at 'localhost:8080/home'. The page title is 'Buscar'. Below the title, it says 'Resultado da Busca !'. There is a search input field with the text 'Buscar :' and a button labeled 'Enviar'. Below this, there is a table with two columns: 'Nome:' and 'E-mail:'. The table contains one row with the values 'Rolfi Luz' and 'rolfi.luz@sp.senai.br'.

Nome:	E-mail:
Rolfi Luz	rolfi.luz@sp.senai.br

Listar, excluir e editar registros do banco de dados

Em muitas aplicações, às vezes é necessário listar, excluir ou editar registros do banco de dados. Observe que, de acordo com a imagem da estrutura das pastas do projeto, os arquivos criados no controller **listafuncionarioController.java** estão no local “**com.rh.app.controller**”, e os criados na view **listafuncionarios.html** estão na pasta “**src/main/resources/funcionario**”.



A página listafuncionario.html possui o seguinte código-fonte:

```
<!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>
```

Observe que esta página é muito semelhante à página index.html, porém sem o formulário de busca, mas contendo os comandos th:each do thymeleaf para poder imprimir a lista dos registros que estão gravados no banco de dados. Diferentemente da index.html, que precisava de um valor via POST para exibir na listafuncionario.html, esta página imprime todos os valores via GET diretamente.

Por consequência, temos a página a seguir ao entrar no endereço <http://localhost:8080/lista>.

Nome:	E-mail:		
Ana Paula C Luz	ana@ana.com.br	Excluir	Editar
Rolfi Luz	es.rolfi2@gmail.com	Excluir	Editar
Rolfi	es.rolfi2@gmail.com	Excluir	Editar



Considere os seguintes códigos da view para os botões excluir e editar:

Botão excluir:

```
<a th:href="@{'/deletarFuncionario/' + ${funcionario.id} }" class="waves-effect waves-light btn-small">  
    <button type="button" class="btn btn-danger">Excluir</button>  
</a>
```

Botão editar:

```
<a th:href="@{'/editarFuncionario/' + ${funcionario.id} }" class="waves-effect waves-light btn-small">  
    <button type="button" class="btn btn-primary">Editar</button>  
</a>
```

A lista da view possui botões excluir e editar para cada registro da tabela Funcionários. Esses botões são links que possuem a expressão para uma URL dinâmica em thymeleaf **th:href="@{...}"**.

Essa expressão faz com que a tag <a>, que é um link em HTML, seja um link dinâmico inserindo a requisição **'/deletarFuncionario/'** ou **'/editarFuncionario/'**, com o valor do ID do funcionário impresso conforme consta no registro com o restante da expressão **+ \${funcionario.id}**.

Assim, quando o usuário clica em excluir ou editar, será enviada uma requisição de exclusão ou edição para o registro do funcionário escolhido. Ao chamar essas requisições, o controller `listafuncionarioController` intercepta esses endereços e faz a edição ou exclusão dos dados.

Considere o código-fonte do controller `listafuncionarioController.java` como sendo o seguinte:

```
1 package com.rh.app.funcionario.controller;  
2  
3 import org.springframework.beans.factory.annotation.Autowired;  
4 import org.springframework.stereotype.Controller;  
5 import org.springframework.web.bind.annotation.PathVariable;  
6 import org.springframework.web.bind.annotation.RequestMapping;  
7 import org.springframework.web.bind.annotation.RequestMethod;  
8 import org.springframework.web.servlet.ModelAndView;  
9  
10 import com.rh.app.funcionario.models.Funcionario;  
11 import com.rh.app.funcionario.repository.FuncionarioRepository;  
12  
13 /** @author Rolfi Luz - Senai * */  
14 @Controller  
15 public class listafuncionarioController {  
16  
17     @Autowired  
18     private FuncionarioRepository fr;  
19  
20     @RequestMapping(value = "/lista", method = RequestMethod.GET)  
21     public ModelAndView listarfuncionario() {  
22         ModelAndView mv = new ModelAndView("funcionario/listafuncionario");  
23         mv.addObject("funcionarios", fr.findAll());  
24         return mv;  
25     }  
}
```



```
26
27 @RequestMapping(value = "/deletarFuncionario/{id}", method = RequestMethod.GET)
28 public String deletarFuncionario(@PathVariable("id") long id) {
29     fr.delete(fr.findById(id));
30     return "redirect:/lista";
31 }
32
33 @RequestMapping(value = "/editarFuncionario/{id}", method = RequestMethod.GET)
34 public ModelAndView abrireditarfuncionario(@PathVariable("id") long id) {
35     ModelAndView mv = new ModelAndView("funcionario/editarfuncionario");
36     mv.addObject("funcionario", fr.findById(id));
37     return mv;
38 }
39
40 @RequestMapping(value = "/editarFuncionario/{id}", method = RequestMethod.POST)
41 public String updateFuncionario(Funcionario funcionario) {
42     fr.save(funcionario);
43     return "redirect:/lista";
44 }
45 }
```

Este controller possui as programações da view para excluir, editar e listar os dados. Para requisições GET do endereço /lista, é executado o método **listarfuncionario()**, que envia à view (página listafuncionario.html) o retorno do **repository fr.findAll()**, enviando todos os registros de funcionários do banco de dados por meio da linha 23.

Deletando valores

Para deletar valores do banco de dados, a view passa o funcionário que será deletado através do link dinâmico em thymeleaf para o endereço **/deletarFuncionario/{funcionario.id}**. Diferentemente das formas de gravação, nas quais os valores são recebidos via método POST (a partir da submissão de um formulário), este formato trabalha com o método GET, enviando os dados via URL ou via endereço pela barra de endereços do navegador; neste caso, para deletar o funcionário do ID 5, pode-se digitar diretamente: <http://localhost:8080/deletarFuncionario/5>.

Nesse sentido, o **@RequestMapping** da linha 27 recebe o endereço e o valor do id e faz a separação entre o endereço e o id a ser deletado. Após o controller interceptar esse endereço, ele recupera o id pela expressão **{id}** que está dentro do **@RequestMapping** e envia para a linha 28 a função **deletarFuncionario** e o parâmetro **@PathVariable("id") long id**.

Na linha 29, a **fr.delete()** deleta o funcionário que é procurado pela função **fr.findById(id)**; caso não o localize, retorna para a própria view que é a /lista (e não faz a exclusão).

```
27 @RequestMapping(value = "/deletarFuncionario/{id}", method = RequestMethod.GET)
28 public String deletarFuncionario(@PathVariable("id") long id) {
29     fr.delete(fr.findById(id));
30     return "redirect:/lista";
31 }
```

Editando valores

Para editar valores de uma view, pode-se usar várias formas, uma delas é criar uma view para update ou para edição, semelhante à view de gravação. Neste tutorial, faremos de forma separada. A página que será a view da edição será

a **editarfuncionario.html**, uma página HTML com expressões em **thymeleaf** que aplicam o valor do **funcionario**, buscando-o nos campos do formulário em HTML.

Considere o código-fonte da view editarfuncionario:

```
<!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>

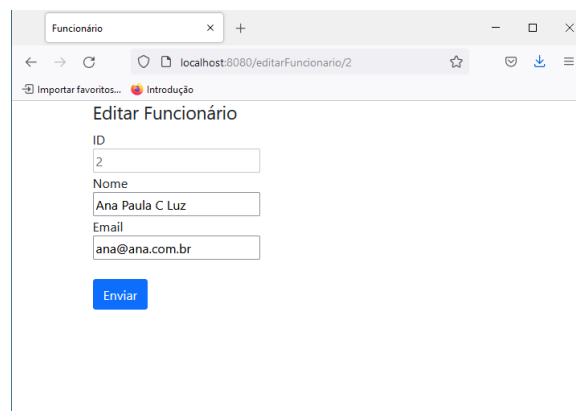
    <div class="container">

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

    </div>

</body>
</html>
```

Após o usuário entrar no endereço /lista e clicar em “Editar”, aparecerá uma página HTML com os valores preenchidos (do funcionário escolhido da lista), como mostra a tela a seguir.



Funcionário

← → ↻ 📄 localhost:8080/editarFuncionario/2

🔖 Importar favoritos... 📖 Introdução

Editar Funcionário

ID
2

Nome
Ana Paula C. Luz

Email
ana@ana.com.br

Enviar



Observe que o endereço **http://localhost:8080/editarFuncionario/2** é o que consta na barra de endereços. Na linha 33, temos o **@RequestMapping**, que intercepta o endereço por meio do **método GET** da rota **/editarFuncionario/2** e aciona o método **abrieditarfuncionario()** na linha 34, passando o id como parâmetro semelhante ao **@RequestMapping** do excluir. Após a separação do endereço e do id, é localizado o funcionário que possui o id solicitado e os dados são retornados por meio da linha 36, em que se chama o funcionário pelo método **findById()** e repassa-o para o método **mv.addObject()**, retornando, assim, os dados do funcionário no formulário HTML para edição.

```
33  @RequestMapping(value = "/editarFuncionario/{id}", method = RequestMethod.GET)
34  public ModelAndView abrieditarfuncionario(@PathVariable("id") long id) {
35      ModelAndView mv = new ModelAndView("funcionario/editarfuncionario");
36      mv.addObject("funcionario", fr.findById(id));
37      return mv;
38  }
```

Temos, ainda, a requisição por **POST**. A linha 40 receberá os dados do funcionário que foram digitados no formulário da view **editarfuncionario.html** e passará via parâmetro o objeto **funcionario** para o método **updateFuncionario()**, contido na linha 41. O comando de gravação do repositório **fr.save()**, que envia o objeto recebido para gravação no banco de dados, será executado na linha 42. Após a gravação do banco de dados, os dados são redirecionados para a rota /lista mostrada na linha 43.

```
40  @RequestMapping(value = "/editarFuncionario/{id}", method = RequestMethod.POST)
41  public String updateFuncionario(Funcionario funcionario) {
42      fr.save(funcionario);
43      return "redirect:/lista";
44  }
```

Acesso ao código completo: https://github.com/senaidev-ead/Codigos-UC_12/tree/main/SA2/codigo_apostila_material_web/app