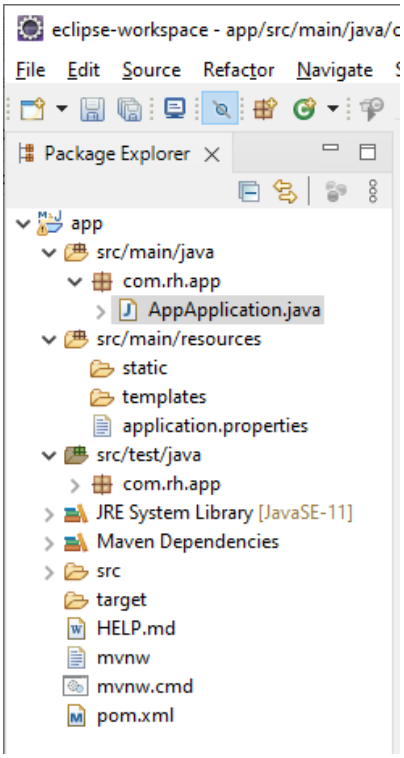
MVC – Modelo, View e Controller

Estrutura de pastas

Ao utilizar o Spring Boot, aplica-se os conceitos de programação em três camadas, o que é denominado de aplicação MVC (Model, View e Controller). As três camadas dentro da aplicação MVC são pastas ou pacotes que conterão arquivos **.java**, classes ou qualquer outro arquivo, o qual irá agrupar responsabilidades distintas dentro do projeto. Inicialmente, a estrutura do projeto, ao utilizar o start initilizr do Spring Boot, será como a figura a seguir ilustra.



Os pacotes dentro do Spring para as três camadas são compostos pelas convenções: views, controller e models.

A camada **view** está dentro do **src/main/resources/**. Nessa pasta, estão localizadas as páginas que serão executadas no browser do cliente e os frameworks responsáveis pelas tecnologias de front-end, como bootstrap, angular etc. Por exemplo: páginas html ou jsp, como **index.html** ou **index.jsp**.

A camada **model** contém todas as classes que serão gravadas dentro de um banco de dados ou comporta classes que serão preenchidas com dados do usuário para execução de uma tarefa. Ela também contém as chamadas dos frameworks de persistência, como Hibernate ou drivers para banco de dados MYSQL, Postgre ou Oracle. O pacote Repository também faz parte da camada model, trata-se de uma camada intermediária entre o model e os frameworks de persistências.

As classes dos Repository são muito utilizadas dentro das classes Controller para acessar os dados do banco de dados. Por exemplo: classes Funcionário; Dependentes; Vagas ou Candidatos que serão gravadas em banco de dados.

A camada **controller** contém todas as classes que farão parte das regras de negócios do projeto ou da aplicação, comportando a lógica do programa e a implementação dos requisitos.

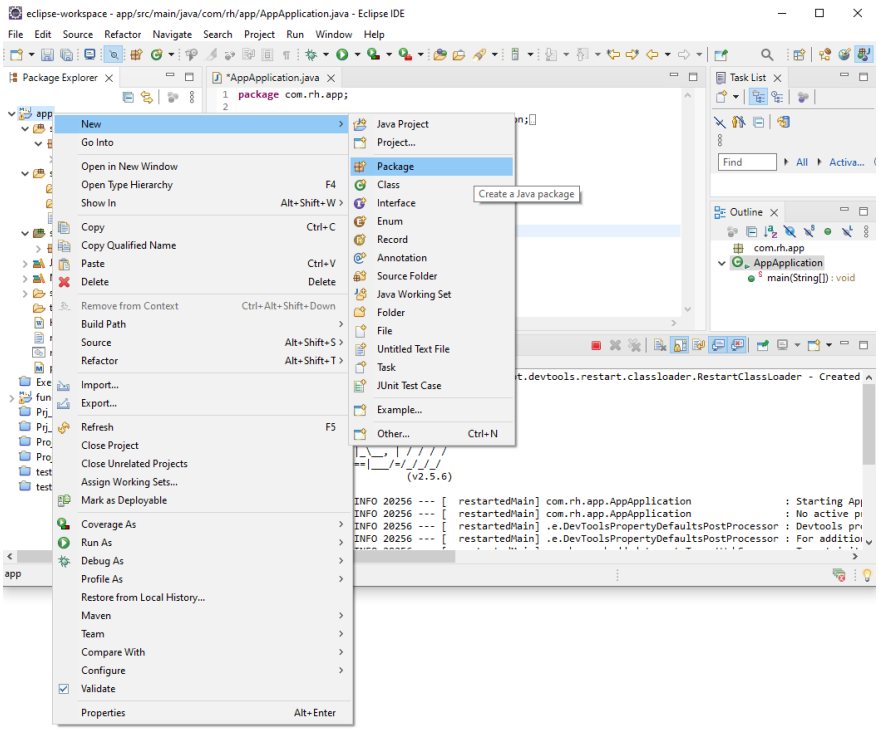
As camadas **controller** e **model** geralmente estão dentro da **src/main/java** e **com.rh.app** com esses nomes, no entanto podem ser criadas de maneiras distintas, acompanhe nas próximas páginas um exemplo da criação de pacotes para essas camadas.

Criando as estruturas de pacotes (pastas) para o MVC

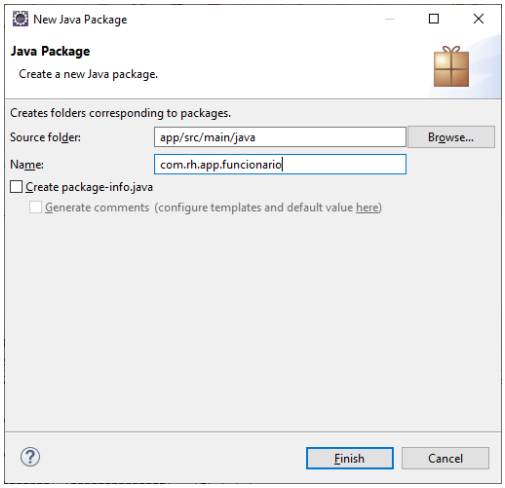
Dentro dos frameworks, geralmente, separa-se por pacotes ou pastas o assunto principal da aplicação. No exemplo a seguir será desenvolvida uma aplicação MVC do tipo CRUD (Create, Read, Update e Delete) para cadastro de funcionário.

Neste sentido, para seguir as boas práticas, devem ser criados os pacotes MVC para o assunto funcionário; um pacote para model e controller.

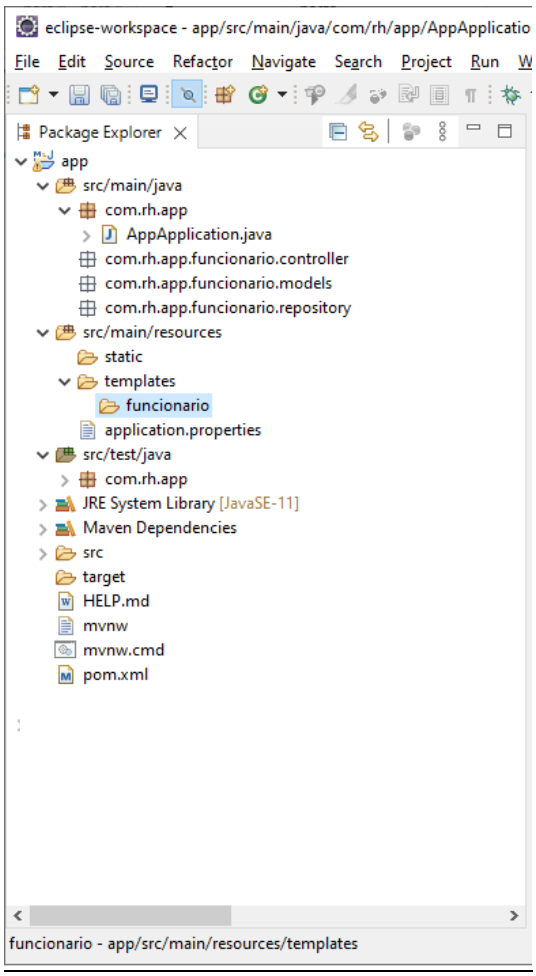
Na figura a seguir, mostra-se o caminho para criação dos pacotes. Clicar no botão **direito do mouse** sobre o pacote **app** ou no **nome do projeto**, em seguida, selecionar as opções **New** e depois **Package**.



Observe, ao abrir o **Package**, em Name, insere-se o nome do assunto principal da aplicação: funcionario.



Após a criação do pacote **funcionario**, deverão ser criado os pacotes MVC para esse assunto com o mesmo formato, criando-se, assim, a estrutura do pacote funcionario – controller, models e repositor, como ilustrada a seguir.



Dependências – pom.xml

Nas estruturas de pastas, um arquivo muito importante é o **pom.xml**, esse arquivo **contém as dependências necessárias para o projeto funcionar**, pode-se adicionar ou remover pacotes de dependência do pom.xml escrevendo dentro do arquivo. No exemplo seguinte, dentro da meta tag <dependencies>, coloca-se cada dependência pelo conjunto <dependency></dependency>.

```
<dependencies>
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-security</artifactId>
		</dependency>
		<dependency>
			<groupId>OUTROPACOTE</groupId>
			<artifactId>NOMEDOARQUIVO</artifactId>
		</dependency>

</dependencies>
```

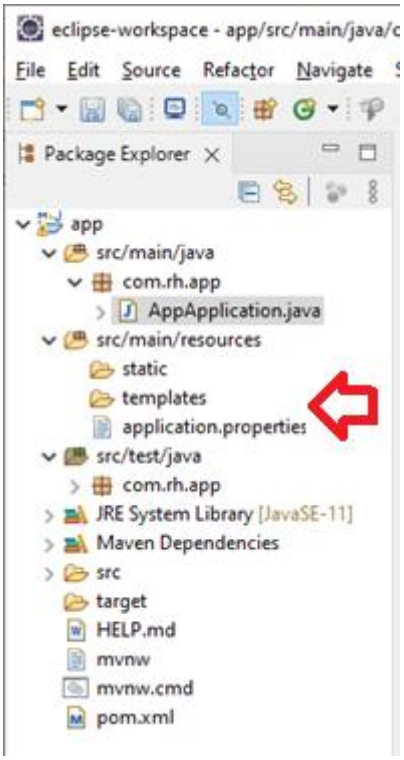
O Spring Boot, quando inicia o projeto, escaneia esse arquivo e faz o download das dependências, caso elas não existam, e aplica as dependências dentro dos comandos. Esse processo é feito por um plug-in da IDE Eclipse chamado **maven**.

O **maven** é um repositório Java em que contém diversos frameworks, aplicações, pacotes e bibliotecas para execução de diversas tarefas para a linguagem Java. Após baixar as dependências, o Spring Boot busca, dentro da programação, as classes que estão sendo programadas e aplica a injeção de dependência.

A **injeção de dependência** é a característica de padrões de projeto e de boas práticas, em que uma classe possa ser testável e reutilizável sem causar mudanças bruscas na estrutura do projeto.

View

A **view** é a divisão de projetos, responsável pela camada de apresentação no **front-end**, ou seja, a apresentação das páginas web para o cliente. Um dos caminhos para colocar a camada de apresentação em um projeto Spring é a seguinte pasta: **src/main/resources/templates**.



A **static** é o diretório onde são colocadas as pastas de frameworks de front-end.

Confira, a seguir, frameworks para front-end e seus respectivos links.

Bootstrap: https://getbootstrap.com/

Pure: https://purecss.io/

Material-UI: https://mui.com/pt/#/

Semantic UI: https://semantic-ui.com

Materialize: https://materializecss.com/

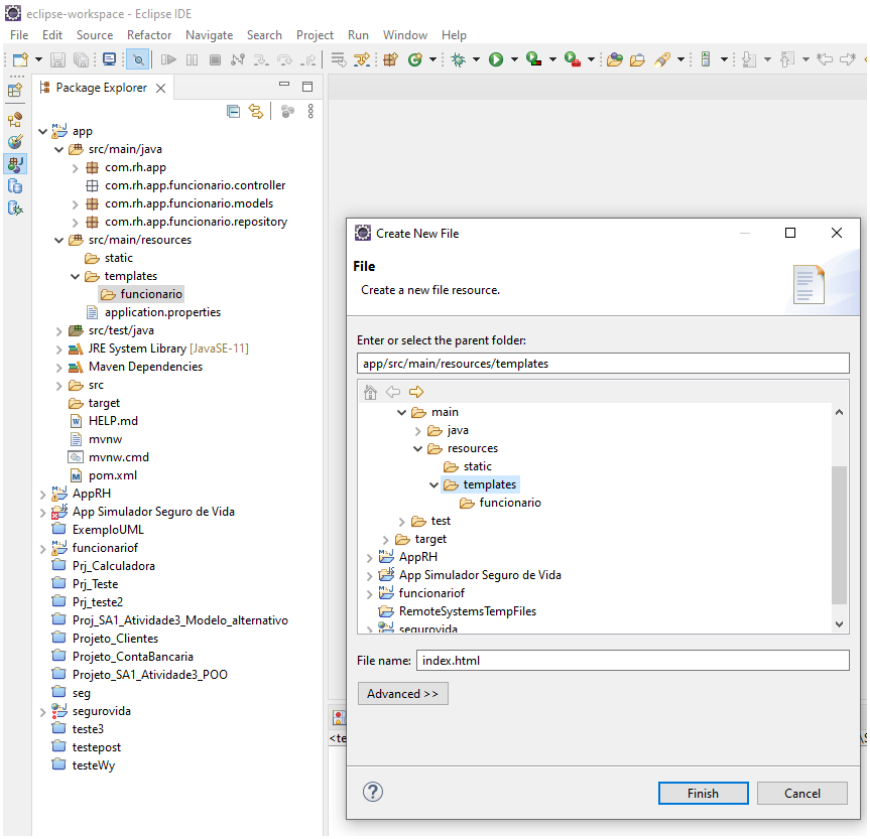
Foundation: https://get.foundation/

No link: https://codigosimples.net/frontend-frameworks/, há uma lista com mais de 45 frameworks para front-end.

Também é possível utilizar o framework sem incluí-lo no diretório **static**, mas é possível referenciá-lo com a tag <HEAD>.

A pasta **templates** é onde ficam as páginas web que serão apresentadas na página do cliente.

Para criar uma página web, você deverá clicar com o botão direito do mouse no caminho escolhido e no campo **File name**, digitar o nome da página com a extensão **.html**. No caso do exemplo a seguir, está sendo criada uma página **index.html** dentro da pasta **templates**.



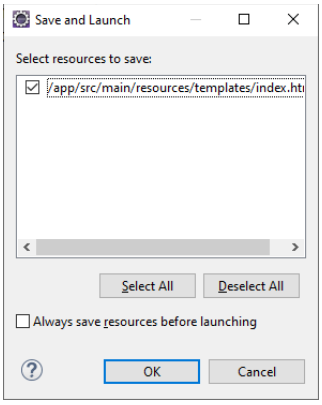
Nessa camada, pode conter as páginas com as extensões **.html**, **.css** e **.js** ou **.jsp**. Acompanhe, a seguir, um exemplo de uma página HTML, com o framework bootstrap na linha em realce, com o nome de arquivo **index.html**.

```
<!doctype html>
<html lang=”pt-br” >
  <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>Hello, world!</title>
```

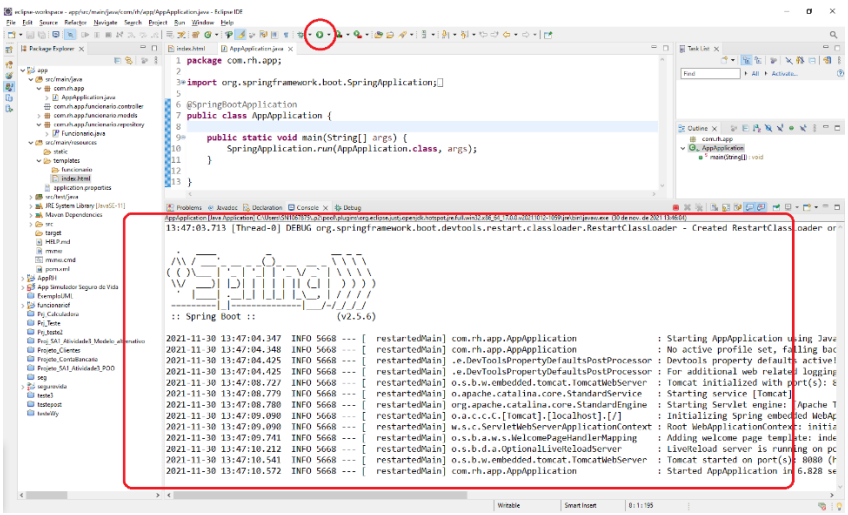
```
  </head>
  <body>
    <h1>Hello, world!</h1>
  </body>
</html>
```

No caso do boostrap desse projeto, ele é chamado pela URL do CDN, outra maneira poderia ser colando o framework **boostrap.css** na pasta static e referenciá-la na página.

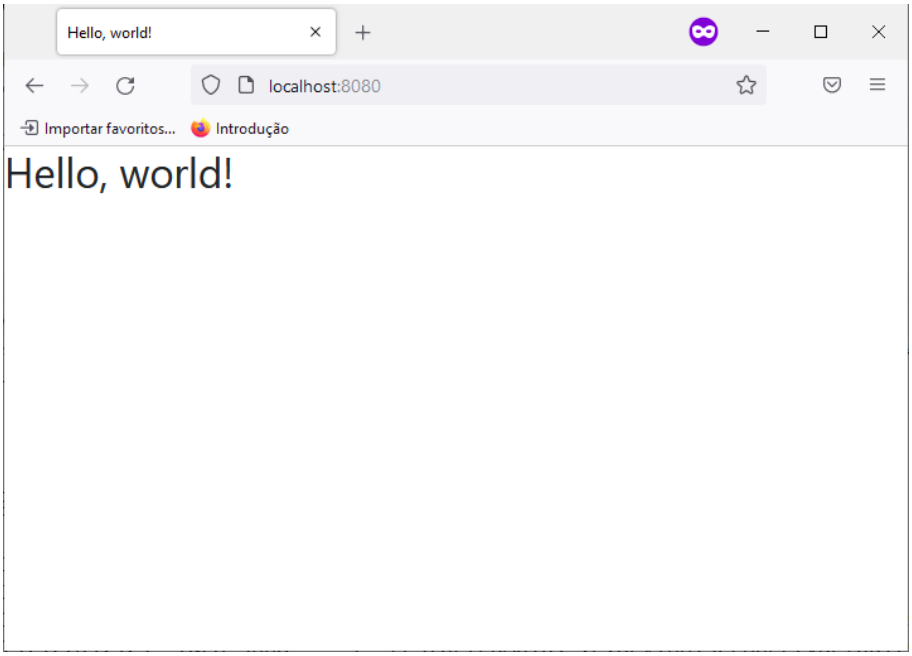
Ao executar o projeto, faça as seguintes seleções no menu superior: **Run** -> **Run As** -> **Java Aplication** e clicar em **OK**.



Após o **Run As**, a IDE Eclipse irá subir o Spring, e o status da execução do Spring aparecerá no **Console**. A informação mais importante desse console é a última linha: **Started AppApplication in 6.828 s**. No caso do start, significa que a aplicação está sendo executada no servidor, caso haja algum erro, não irá aparecer na última linha o **Started AppApplication in**.



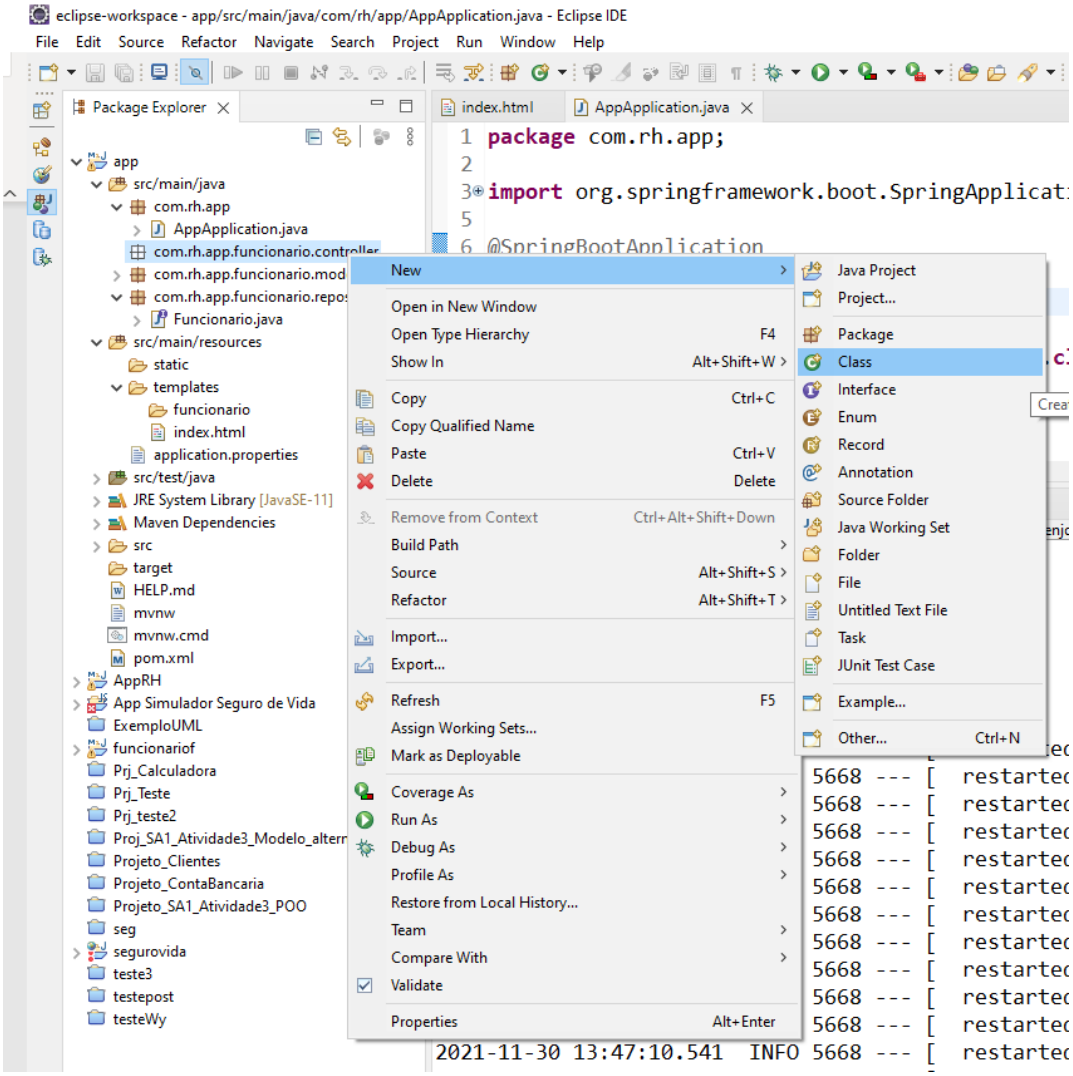
Para visualizar o projeto, você deverá entrar no endereço **localhost:8080**.



Controller

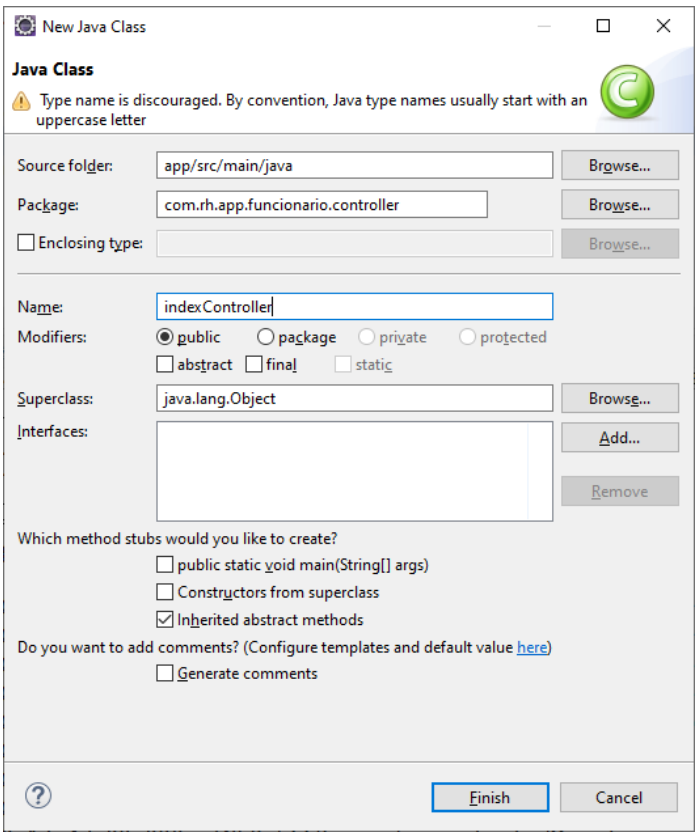
A camada **controller** é responsável pela lógica do sistema e implementação dos requisitos e das regras de negócio. A camada **view** (pacote **src/main/resources**) é utilizada para mostrar ao usuário as informações do sistema.

Na sequência, é exemplificada a criação da classe controller. Clicar no pacote **com.rh.app.funcionario.controller** com o botão direito do mouse, em seguida, selecionar as opções **New** e **Class**.



Após inserir, no campo **Name**, o teor **indexController**, clicar em **Finish**. A IDE Eclipse irá gerar a classe ilustrada a seguir, chamada **indexController**.

Uma das regras de nomes para controllers é a repetição do nome da view, sempre com o final controller, por exemplo: **index.html** e **indexController**; ou **lista.html** e **listaController**, e assim por diante.



O código fonte, a seguir, é gerado pela IDE Eclipse para implementação.

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

public class indexController {

}
```

Mapeando páginas e controle de rotas

Os **controllers** são as classes Java que irão controlar a aplicação. As classes controllers devem possuir métodos e dentro dos métodos podemos inserir as instruções como: declaração de variáveis, instanciação de objetos, inserir instruções de loops, condicionais, operações aritméticas e booleanas, casting e recursão, entre outras instruções que executam no lado servidor.

Um dos conceitos mais utilizados dentro das classes de controllers são as anotações ou **annotations**. As anotações são palavras reservadas após o símbolo **@** que servem para informar o comportamento de uma classe ou dar uma instrução ao framework. A depender da informação que a anotação fornece, pode-se ter comportamentos diferentes no servidor.

Para que uma classe seja um controller dentro do Spring Boot, deve-se usar a anotação **@Controller**. Confira no código-fonte, a seguir, a implementação da classe **indexController** com 34 linhas.

Na linha 10 desse código, após inserir a anotação dentro da classe, a própria IDE Eclipse solicitará o pacote que disponibiliza a anotação @Controller. A IDE Eclipse, utilizando o atalho **CTRL+SHIFT+O**, fornece de forma automática os pacotes que podem conter a anotação ou classes. No caso da anotação controller, o pacote disponível é o **org.springframework.stereotype.Controller**, como está na linha 3.

Segue, na sequência, o código da indexController e mais abaixo seu respectivo código em HTML.

```
1 package com.rh.app.funcionario.controller;
2
3 import org.springframework.stereotype.Controller;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RequestMethod;
6 import org.springframework.web.bind.annotation.RequestParam;
7 import org.springframework.web.servlet.ModelAndView;
8
9 /** @author Rolfi Luz - Senai  *  */
10 @Controller
11 public class indexController {
12
13     @RequestMapping(value = "/home", method = RequestMethod.GET)
14     public ModelAndView abrirIndex() {
15             ModelAndView mv = new ModelAndView("index");
16
17             String mensagem = "Olá, seja bem-vinda(o)!";
18             mv.addObject("msg", mensagem);
19
20             return mv;
21     }
22
23     @RequestMapping(value = "/home", method = RequestMethod.POST)
24     public ModelAndView buscarIndex(@RequestParam("buscar") String buscar) {
25             ModelAndView mv = new ModelAndView("index");
26
27             String mensagem = "Resultado da Busca !";
28             mv.addObject("msg", mensagem);
29             mv.addObject("buscar", buscar);
30
31             return mv;
32     }
33
34 }
```

No código index.html do pacote **view**, foi inserida, na tag HTML, a referência ao framework **thymeleaf**, para que se possa receber valores da indexController e mostrá-los em tela.

EL - Expression Language – Thymeleaf

O **thymeleaf** é um template engine que interage com o controller e está na view fazendo com que ela possa imprimir valores da controller; e da view enviar valores para o controller. O thymeleaf é interpretado pelo Tomcat do lado servidor.

O thymeleaf é inserido dentro das páginas HTML, pela meta tag HTML `xmlns:th=http://thymeleaf.org`. Assim, dentro do html, pode-se usar as EL (Expression Language), que são expressões que podem ser usadas para acessar os dados

advindos de um controller ou variável, e imprimi-los em HTML para o usuário de forma dinâmica, como na tag span, a expressão é EL **${ ... }** `<span th:text="${msg}">`.

As EL no front-end são muito utilizadas, pois é possível fazer alguns tratamentos de dados, loops básicos ou simples condicionais, melhorando o suporte para a impressão das variáveis recebidas pelo controller.

Saiba mais: dentro do Java, existem outras formas de se trabalhar com o front-end utilizando componentes dos frameworks Prime Faces e Ice Faces, que aplicam EL através do Java ServerFaces.

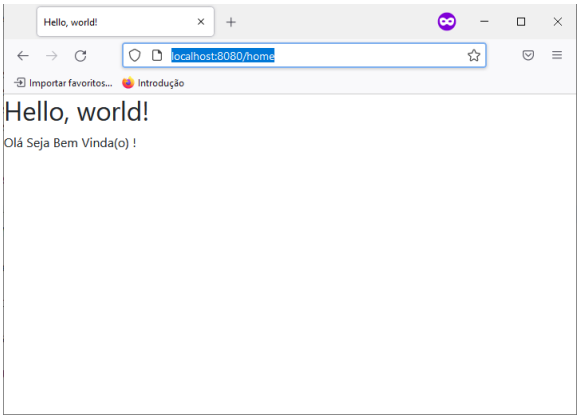
Confira os links:

- https://www.primefaces.org/
- http://www.icesoft.org/java/projects/ICEfaces/overview.jsf

Na sequência, consta o código-fonte da **index.html**, o qual possui a EL em thymeleaf **th:text=**.

```
<!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>Hello, world!</title>
  </head>
  <body>
    <h1>Hello, world!</h1>

    <span th:text="${msg}"></span>

  </body>
</html>
```

Ao executar o projeto com as mudanças, é possível observar o seguinte resultado:



Controlando rotas – método GET

O método GET dentro do Back-end e do Spring Boot tem a finalidade de receber uma solicitação HTTP Request ou endereço de uma página web. Ao receber essa solicitação, ele tem o retorno de uma página da view ou a execução de instruções.

No código do indexController, tem-se a anotação **@RequestMapping**. Essa anotação mapeia o endereço solicitado pelo usuário: **http://localhost:8080/home**. Quando o usuário digitar esse endereço, a função **public ModelAndView abrirIndex()** será executada.

```
13      @RequestMapping(value = "/home", method = RequestMethod.GET)
14      public ModelAndView abrirIndex() {
15              ModelAndView mv = new ModelAndView("index");
16
17              String mensagem = "Olá, seja bem-vinda(o)!";
18              mv.addObject("msg", mensagem);
19
20              return mv;
21      }
```

O método **abrirIndex()** tem como comportamento imprimir **Olá, seja bem-vinda(o)!** na página **/home**. O endereço do site que o usuário irá solicitar é **/home**, e o **RequestMapping** irá mapear e interceptar a URL **/home**. E, através do objeto **ModelAndView("index")**, é possível retornar à página **index** que está na camada **view src/main/resources/templates**. Desse modo, observa-se que o usuário poderá digitar na URL endereços que podem ser mapeados, além de retornar páginas que não possuem o mesmo nome.

O controller é responsável por executar os requisitos funcionais, não funcionais e regras de negócios, porém para enviar os dados para as views, deve-se utilizar o método **addObject** do objeto **ModelAndView mv**.

O objeto **mv** possui o método **addObject ("msg", mensagem)**.

Esse método recebe dois parâmetros: o primeiro é o **"msg"** e a segundo a variável **mensagem**. O primeiro parâmetro é o nome que será enviado para a página **index.html** e tem como valor o conteúdo da variável **mensagem**.

Na página **index.html**, que está em view para receber a variável e imprimi-la em tempo de execução, deve-se utilizar a scriptlet **${msg}**. Na página **index.html**, a tag HTML possui a inserção do seguinte thymeleaf:

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

E na tag Body, temos o scriptlet em thymeleaf: `<span th:text="${msg}"></span>`.

Assim como se utiliza a tag Span, qualquer tag HTML poderia ser usada para aplicar o thymeleaf. Nesse caso, a th receberá o valor da variável **${msg}**, que foi setada na classe **indexController**. Desse modo, é possível imprimir através do comando: < span **th:text=**“${msg}”>.

Caso a variável **${msg}** não tenha valor para imprimir, a parte **HTML** da página **index.html** irá exibir normalmente.

Controlando rotas – método POST

O método POST dentro do back-end e do Spring Boot tem a finalidade de receber dados de um formulário da web ou uma resposta (HTTP Response) de um endereço ou sistema. A anotação **@RequestMapping** está configurada para receber dados enviados para **/home**, através da escolha do **method = RequestMethod.POST**.

```
23      @RequestMapping(value = "/home", method = RequestMethod.POST)
24      public ModelAndView buscarIndex(@RequestParam("buscar") String buscar) {
25              ModelAndView mv = new ModelAndView("index");
26
27              String mensagem = "Resultado da Busca !";
28              mv.addObject("msg", mensagem);
29              mv.addObject("buscar", buscar);
30
31              return mv;
32      }
```

Ao observar o método **buscarIndex()**, tem-se duas variáveis: **mensagem** e **buscar**. A variável **mensagem** é uma variável criada dentro do método, enquanto a variável **buscar** está sendo recebida pela requisição HTTP Response, ou seja, de um formulário ou sistema, através da anotação **@RequestParam(“buscar”)**. A RequestParam recebe o parâmetro **buscar** do sistema e coloca os valores desse parâmetro na variável **String buscar**, assim, todo o método buscarIndex() consegue visualizar a variável **buscar**.

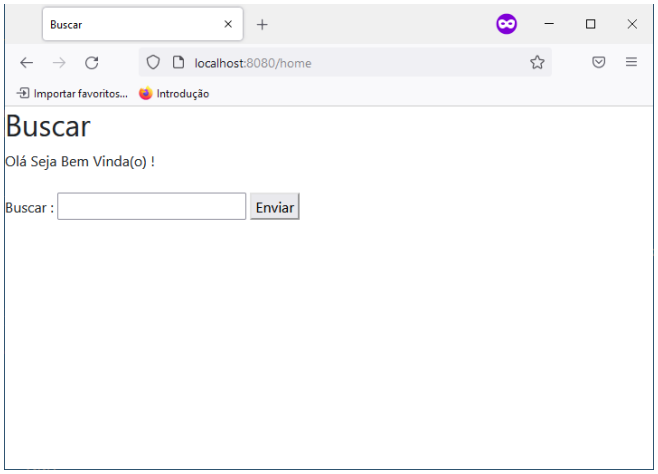
E, por fim, essas duas variáveis são enviadas para o **ModelAndView(“index”)** no intuito de mostrar os dados na página **index.html**.

IMPORTANTE: nesta etapa, é preciso modificar os códigos da **index.html**, conforme ilustrado a seguir, para poder receber esses valores pelo método POST. Considere o seguinte código, com as mudanças destacadas.

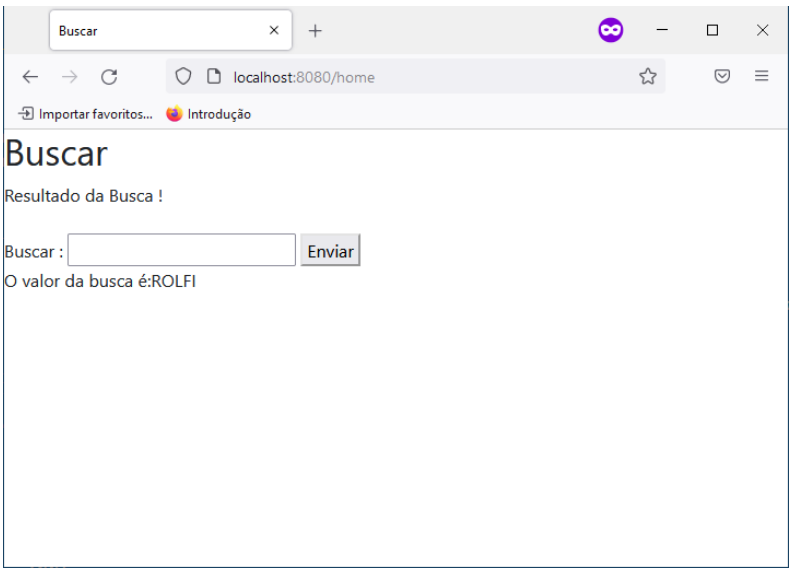
```
<!doctype html>
<html lang=”pt-br” xmlns:th="http://thymeleaf.org">
<head>
```

```
<!-- 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>

        <span th:if="${buscar}">
                O valor da busca é:<span th:text="${buscar}"></span>
        </span>

</body>
</html>
```

Esse novo código gerará a seguinte view ao usuário:



Após as mudanças da index.html, pode-se digitar valores na página e, ao clicar em **Enviar**, aparecerá o valor pesquisado.



Observe que as variáveis **msg** e **buscar** expressaram os valores **Resultado da Busca!** e **ROLFI** no método **buscarIndex()**.

A tag form, no código apresentado em HTML, é a tag que cria um formulário para entrada de dados e transporta esses dados via HTTP Response para outra página, serviço, aplicativo ou URL. Observe que ele possui a caixa de texto que irá receber com as propriedades name e id "buscar".

Dentro do thymeleaf, observe o código **th:if=**, dentro da tag span. O thymeleaf, embora seja uma tecnologia para view ou front-end, possui condicionais, loops, criação de variáveis, porém com escopo bem reduzido. Nesse exemplo, o thymeleaf verifica se a variável **${buscar}** possui valores válidos, caso sim, ele imprime em tela "O valor da busca é: ${busca}"; nesse caso, a variável buscar sendo válida, imprimirá o valor indicado.

```
<span th:if="${buscar}">
        O valor da busca é:<span th:text="${buscar}"></span>
</span>
```