



Desenvolvimento de Sistemas

Ambiente de desenvolvimento (Spring MVC)

Introdução

A seguir, será abordado como instalar e configurar um ambiente de desenvolvimento para programar projetos Spring MVC. Nesse momento, não será dada atenção à persistência dos dados em um banco de dados. Assim que o ambiente estiver pronto, será iniciada a criação dos projetos, com muitas dicas úteis para ajudá-lo a ter sucesso nesse processo.

Spring e Spring Boot

Como você pôde conferir no conteúdo “Arquitetura *Web*” desta unidade, o Spring Framework permite o desenvolvimento muito mais eficiente de aplicações *web* do que as ferramentas nativas de Java (Servlets, JSP etc.). Essa ferramenta é amplamente adotada pelo mercado por trazer facilidade à configuração com o uso de injeção de dependência, o que permite uma boa modularização no sistema e favorece o desenvolvimento de serviços *web*. Além disso, conta com suporte para tarefas comuns em desenvolvimento *web*, como vinculação de dados (*data-binding*), conversão automática de tipos, validações de eventos, internacionalização e vários outros recursos.

O Spring Boot é uma ferramenta auxiliar ao Spring que visa tornar mais fácil e rápido o processo de criação e configuração de projetos. Seu uso é opcional, mas a principal vantagem é que torna possível criar rapidamente aplicações prontas para produção, com um mínimo de configuração e sem a necessidade de definir manualmente muitos detalhes técnicos. O Spring Boot oferece várias facilidades, como um servidor embutido e um amplo conjunto de ferramentas para ajudar a simplificar o processo de desenvolvimento. Assim, podem ser criadas aplicações que rodam por si próprias, sem necessidade de implantação em servidor *web* externo, como Tomcat.

Instalação e configuração



Para implementação de projeto Spring, as etapas são um pouco diferentes das instalações e configurações-padrão de aplicativos. Como o Spring é um *framework* para projetos Java, é necessário um projeto Java para “instalar” o Spring nesse projeto. Geralmente, isso é feito criando um projeto do zero e, depois, fazendo a instalação do *framework* por meio de algum gerenciador de dependências. Isso pode ser um pouco demorado e muitos erros podem acontecer se cada etapa não for seguida à risca. Sendo assim, será usada outra abordagem.

A forma mais eficaz que se tem para criar um projeto com o Spring é utilizando o Spring Initializr, um serviço *web* que permite aos desenvolvedores criar rapidamente aplicativos Java com o Spring Framework. Ele fornece modelos de projeto pré-configurados, com uma variedade de opções para selecionar e construir o projeto do jeito que precisar. Como o serviço gera automaticamente o código de configuração necessário para iniciar o projeto, você economiza tempo e esforço. O primeiro passo é acessar a página do Spring Initializr (start.spring.io).

The screenshot shows the Spring Initializr web application interface. It features a sidebar with a hamburger menu and a Twitter icon. The main content area is divided into several sections:
 - **Project**: Radio buttons for **Gradle - Groovy** (selected), **Gradle - Kotlin**, and **Maven**.
 - **Language**: Radio buttons for **Java** (selected), **Kotlin**, and **Groovy**.
 - **Spring Boot**: Radio buttons for versions 3.1.0 (SNAPSHOT), 3.1.0 (M2), 3.0.6 (SNAPSHOT), **3.0.5** (selected), 2.7.11 (SNAPSHOT), and 2.7.10.
 - **Project Metadata**: Text input fields for **Group** (com.example), **Artifact** (demo), **Name** (demo), **Description** (Demo project for Spring Boot), and **Package name** (com.example.demo).
 - **Packaging**: Radio buttons for **Jar** (selected) and **War**.
 - **Dependencies**: A section with a button **ADD DEPENDENCIES... CTRL + B** and the text *No dependency selected*.
 - **Footer**: Three buttons: **GENERATE CTRL + G**, **EXPLORE CTRL + SPACE**, and **SHARE...**.
 - **Top Right**: A settings icon and a dark mode toggle (moon icon).

Figura 1 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

Nessa tela, serão definidas todas as configurações iniciais do projeto. Em **Project**, você deve escolher qual gerenciador de dependências utilizará. Um gerenciador de dependências facilita a construção de um projeto, baixando bibliotecas automaticamente a partir de algumas referências configuradas. Durante o curso, você já conheceu o Maven, e é com ele que seguirá nos próximos projetos. Por isso, nos exemplos, normalmente será selecionada a opção **Maven**.

Em **Language**, é possível escolher qual linguagem de programação será utilizada no desenvolvimento do projeto. Nos projetos do curso, será escolhida sempre a opção Java.

Project

☐ Gradle - Groovy ☐ Gradle - Kotlin ☒ **Maven**

Figura 2 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

Language

☒ **Java** ☐ Kotlin ☐ Groovy

Figura 3 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

Em **Spring Boot**, é escolhida a versão da biblioteca que se deseja utilizar. Por padrão, a opção **3.0.6** já vem selecionada por se tratar da última versão do Spring Boot nesta data. É possível selecionar versões mais antigas ou a versão atualmente em desenvolvimento, que se trata de uma versão mais atualizada, mas que pode apresentar alguns bugs em funcionalidades que não tenham sido testadas – não é recomendado selecioná-la. Para garantir que não ocorra nenhuma instabilidade ou imprevisto no desenvolvimento deste projeto, foi selecionada a última versão estável na data de produção deste material, que é a **3.0.6**.

Spring Boot

☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (RC1) ☐ 3.1.0 (M2) ☐ 3.0.7 (SNAPSHOT)
☒ **3.0.6** ☐ 2.7.12 (SNAPSHOT) ☐ 2.7.11

Figura 4 – Site do Spring Initializr

Fonte Spring Initializr (s.d.)

Atenção - As versões do Spring Boot podem variar dependendo de quando você estiver lendo este conteúdo. É recomendado que você utilize a última versão estável da biblioteca sempre que possível.

Em **Project Metadata**, é necessário definir alguns metadados para a criação deste projeto. Veja no infográfico a seguir.



Group: é o nome do grupo ou da organização a qual pertence o projeto.



Artifact: é o nome do artefato gerado pelo projeto. Normalmente é o nome do módulo ou da aplicação.



Name: é o nome descritivo do projeto.



Package Name: é o mesmo campo do pacote raiz do projeto.



Packaging: é o tipo de arquivo que será gerado na compilação do projeto (JAR ou WAR).



Description: é uma breve descrição do projeto.



Java: é a versão do Java que será usada para gerar o projeto.



Veja a seguir um exemplo de preenchimento desses dados.

Project Metadata

Group	<input type="text" value="com.api"/>
Artifact	<input type="text" value="tarefas"/>
Name	<input type="text" value="tarefas"/>
Description	<input type="text" value="API Rest de Projeto de Tarefas."/>
Package name	<input type="text" value="com.api.tarefas"/>
Packaging	<input checked="" type="radio"/> Jar <input type="radio"/> War
Java	<input type="radio"/> 20 <input checked="" type="radio"/> 17 <input type="radio"/> 11 <input type="radio"/> 8

Figura 5 – Site do Spring Initializr

Fonte Spring Initializr (s.d.)

A seção **Dependencies** no Spring Initializr permite que você escolha as bibliotecas e os *frameworks* que deseja utilizar em seu projeto. Para adicionar uma dependência, clique no botão **Add Dependencies**.

Dependencies

ADD DEPENDENCIES... CTRL + B

No dependency selected

Figura 6 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

As dependências disponíveis no Spring Initializr incluem várias bibliotecas populares do ecossistema Spring, como **Spring Web**, Spring Data JPA, Spring Security e Spring Cloud, bem como outras bibliotecas populares, como o Apache Kafka, o Hibernate e o Thymeleaf. Você pode selecionar várias dependências clicando nas caixas de seleção correspondentes. Nos projetos *web* apresentados neste conteúdo, sempre será usado o **Spring Web**.

Spring Web

Press Ctrl for multiple adds

Spring Web WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Reactive Web WEB

Build reactive web applications with Spring WebFlux and Netty.

Spring Web Services WEB

Facilitates contract-first SOAP development. Allows for the creation of flexible web services using one of the many ways to manipulate XML payloads.

Spring HATEOAS WEB

Eases the creation of RESTful APIs that follow the HATEOAS principle when working with Spring / Spring MVC.

Spring for GraphQL WEB

Build GraphQL applications with Spring for GraphQL and GraphQL Java.

Rest Repositories WEB

Exposing Spring Data repositories over REST via Spring Data REST.

Figura 7 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

Projetos Spring MVC ou com banco de dados podem exigir a inclusão de mais bibliotecas de dependência. Ao selecionar uma dependência, o Spring Initializr automaticamente adicionará a versão mais recente daquela dependência ao seu projeto.

É importante notar que adicionar muitas dependências ao seu projeto pode torná-lo mais pesado e aumentar o tempo de construção. Por isso, é importante escolher apenas as dependências necessárias para o seu projeto e mantê-las atualizadas para garantir a segurança e o desempenho do seu aplicativo.

Após selecionar o **Spring Web**, por exemplo, a dependência aparecerá abaixo na lista de dependências.

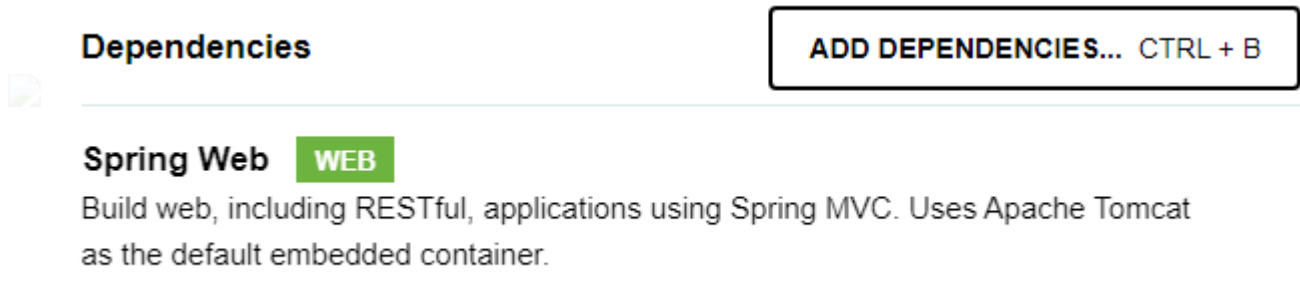


Figura 8 – Site do Spring Initializr

Fonte: Spring Initializr (s.d.)

Após concluir todas as configurações, clique em **Generate**, no rodapé da página, que será feito o *download* de uma pasta compactada (.zip) contendo o projeto completo.

Finalizado o *download*, é possível extrair o conteúdo em seu computador e abrir a pasta do projeto com sua IDE – nesse caso, o *Apache NetBeans IDE*.

Spring MVC

O *framework* Spring conta com uma variedade grande de ferramentas para desenvolvimento *web* e, entre essas ferramentas, uma das mais populares é o Spring MVC, uma implementação do padrão MVC para a construção de aplicações *web*, que usa anotações, convenções, pacotes e pastas para separar as camadas do padrão.

A construção de um projeto Spring MVC pode ser realizada a partir do Spring Boot, que inclui algumas configurações automaticamente. Assim, evita-se a escrita de arquivos XML para configurar. De maneira simplificada, o fluxo de uma requisição no Spring MVC é o seguinte:

Clique ou toque para visualizar o conteúdo.



Request é recebido pelo sistema e tratado internamente por um objeto da classe DispatcherServlet.

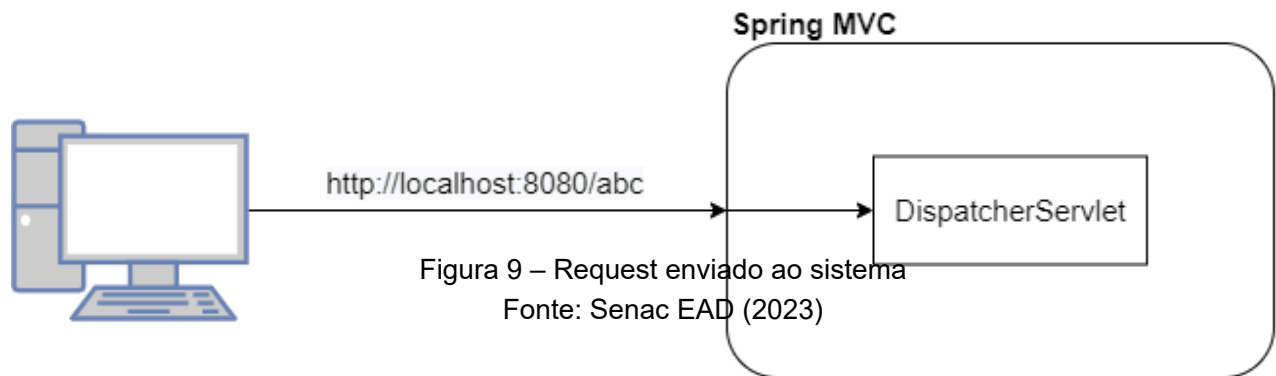
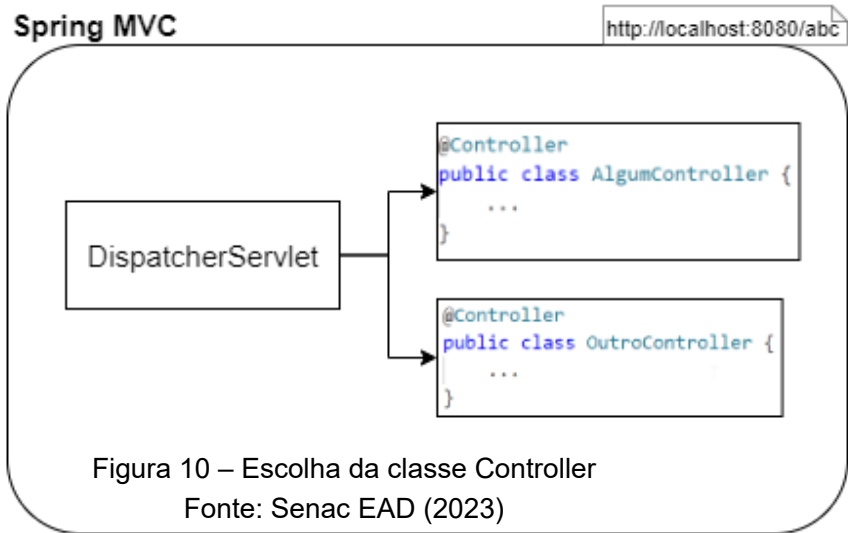
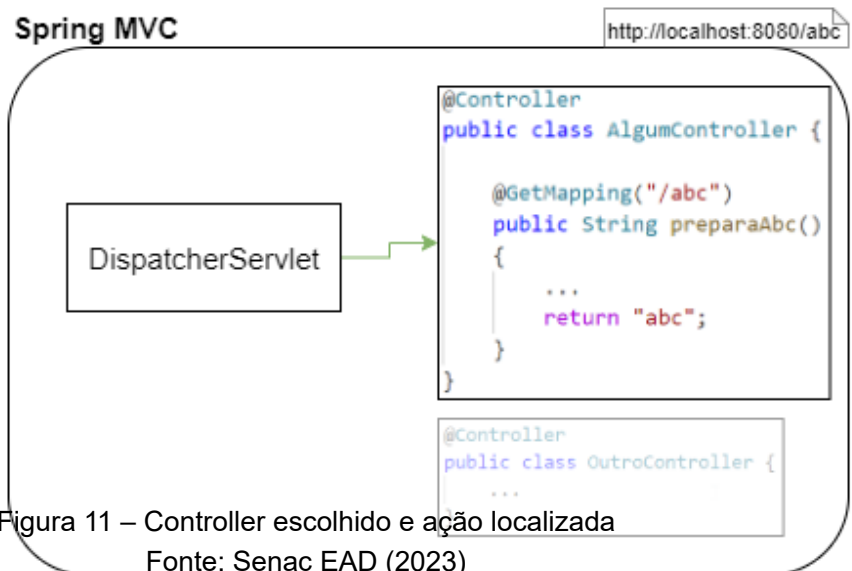


Figura 9 – Request enviado ao sistema
Fonte: Senac EAD (2023)





Spring MVC

http://localhost:8080/abc

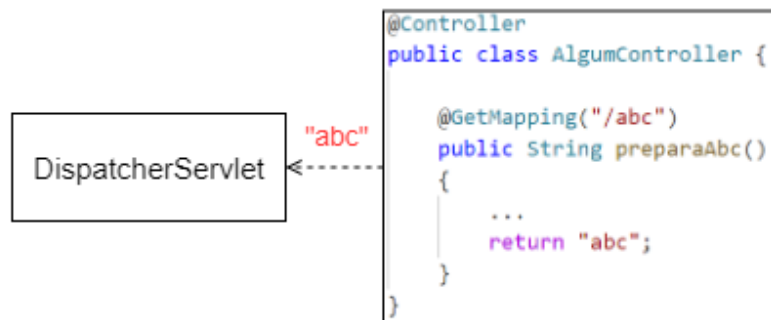


Figura 12 – Retorno do Controller

Fonte: Senac EAD (2023)

Spring MVC

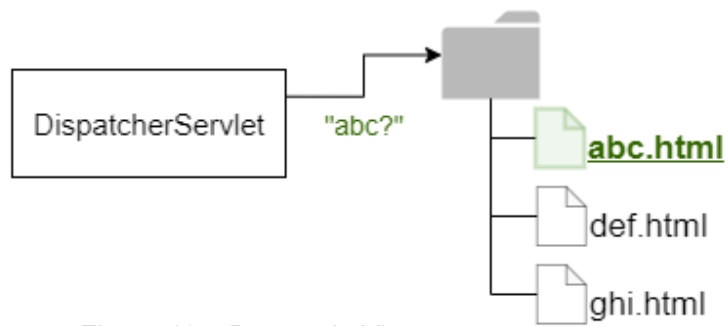
<http://localhost:8080/abc>

Figura 13 – Buscando View
Fonte: Senac EAD (2023)

Spring MVC

http://localhost:8080/abc

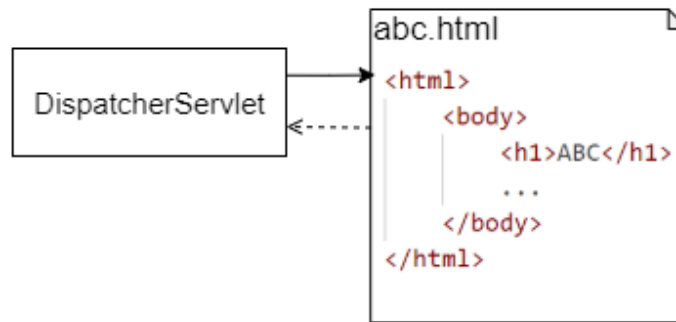


Figura 14 – Formatando HTML da View

Fonte: Senac EAD (2023)

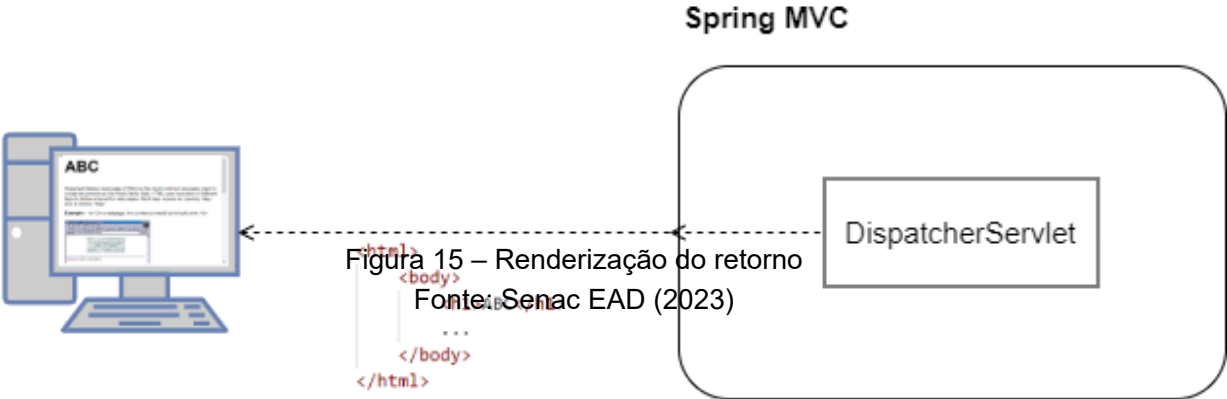


Figura 15 – Renderização do retorno
Fonte: Senac EAD (2023)



Na prática, na maioria das vezes, não é necessário lidar diretamente com o `DispatcherServlet`, mas é importante entender que há algo que orquestra as requisições e as respostas. Esta implementação se concentrará nas classes de controle, nas classes de modelo de negócio e nas páginas.

A seguir, para compreender esse mecanismo, você criará um novo projeto.

Usando o Spring Initializr, crie um projeto com as seguintes configurações:

Project: Maven

Language: Java

Spring Boot: maior versão estável disponível

Group: com.senac

Artifact e Name: projetomvc

Packaging: Jar

Java: 17 (ou versão instalada em sua máquina)

Dependencies: inclua **Spring Web** (categoria *Web*) e Thymeleaf (categoria *Template Engines*)

Baixe o arquivo compactado gerado, extraia e abra a pasta no NetBeans.



Project
☐ Gradle - Groovy
☐ Gradle - Kotlin
☒ Maven

Language
☒ Java
☐ Kotlin
☐ Groovy

Spring Boot
☐ 3.1.0 (SNAPSHOT) ☐ 3.1.0 (RC2) ☐ 3.1.0 (M2)
☐ 3.0.7 (SNAPSHOT) ☒ 3.0.6
☐ 2.7.12 (SNAPSHOT) ☐ 2.7.11

Project Metadata
 Group
 Artifact
 Name
 Description
 Package name
 Packaging ☒ Jar ☐ War
 Java ☐ 20 ☒ 17 ☐ 11 ☐ 8

Dependencies ADD ... CTRL + B
Spring Web WEB
 Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
Thymeleaf TEMPLATE ENGINES
 A modern server-side Java template engine for both web and standalone environments. Allows HTML to be correctly displayed in browsers and as static prototypes.



GENERATE CTRL + G

EXPLORE CTRL + SPACE

SHARE...

Figura 16 – Configurações no Spring Initializr

Fonte: Spring Initializr (s.d.)

A estrutura do projeto, quando aberto no NetBeans, fica conforme a seguir.

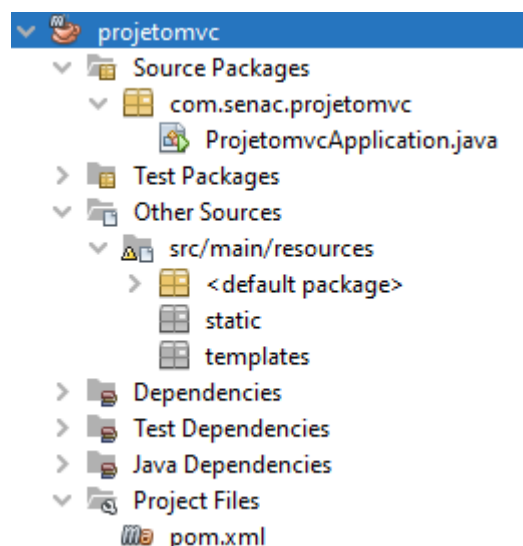


Figura 17 – Projeto Spring MVC

Fonte: NetBeans (2023)

A classe `ProjetoMvcApplication` presente no pacote “com.senac.projetomvc” é a classe que inicia a execução do projeto. Note ainda as pastas “static” e “templates” em “Other Sources” > “src/main/resources” – essa será a camada View, ou seja, nessas pastas, serão incluídos arquivos HTML e CSS.

Neste primeiro exemplo, a implementação será iniciada pela camada View, criando a página HTML que será exibida na tela. Na pasta `src/main/resources/templates`, clique com o botão direito do mouse e selecione `New > HTML File`, e na tela de criação, use como nome “saudacao”. O arquivo “saudacao.html” será criado na pasta, em que será escrito o seguinte código:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Saudação</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

  </head>
  <body>
    <h1>Boas vindas!</h1>
    <p>Parabéns, este é seu primeiro projeto MVC em ação!</p>
  </body>
</html>
```

Dica Ao criar a página HTML no NetBeans, a estrutura básica do documento e as *tags* meta importantes já são incluídas, o que facilita a escrita do código. O IDE ainda conta com sugestão e encerramento automático de *tags*. Use esses recursos para agilizar sua codificação.

Essa é uma página simples, que pode ser acessada localmente abrindo o arquivo direto na pasta em que foi salvo. Um projeto *web*, no entanto, não funciona assim. Em vez disso, há uma URL que aponta para um servidor *web* que administra e entrega o recurso solicitado. No caso deste projeto, ainda não há como acessar o arquivo HTML direto pelo navegador.

Para que isso seja possível, você precisa criar uma classe de controle, pois nela haverá um mapeamento para o endereço de *web* solicitado pelo usuário, que apontará para um método e alcançará o arquivo de visão (HTML). Deve-se, então, programar uma

classe de controle primeiro criando um novo pacote “com.senac.projetomvc.controller” e depois, nesse pacote, criar a classe “MiscController.java”.

Dica Para criar esse pacote, clique com o botão direito do mouse sobre o já existente “com.senac.ProjetoMvc” e apenas complete com “controller” no final.

Na classe MiscController, use o seguinte código:

```
package com.senac.ProjetoMvc.controller;
import org.springframework.stereotype.Controller;
@Controller
public class MiscController {
}
```

O código apenas declara uma classe, mas usa uma anotação especial “@Controller”, proveniente do pacote “org.springframework.stereotype” do Spring. Essa anotação é necessária para que o sistema busque nessa classe qual método deve ser executado a partir da URL, método esse escrito a seguir.

```
package com.senac.ProjetoMvc.controller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
@Controller
public class MiscController {
    @GetMapping("/saudacao")
    public String mostraSaudacao(){
        return "saudacao";
    }
}
```

Há, então, o método “mostraSaudacao()”. O primeiro ponto a observar é a presença da anotação **@GetMapping**, que tem por argumento o valor “/saudacao”. Essa é a configuração que fará com que, quando o usuário digitar na URL <http://localhost:8080/saudacao>, o novo método seja executado (, nestes testes, sempre será localhost:8080).

O método em si não tem nenhum processamento específico, apenas um **retorno** de um texto “saudacao”. Este é exatamente o **nome dessa visão**, o arquivo HTML criado há pouco, e isso não é por acaso: a partir desse retorno, o sistema localizará e exibirá a visão correta – nesse caso, “saudacao.html”.

Você já pode testar esse primeiro exemplo simples. Para isso, clique com o botão direito do mouse sobre a classe “ProjetoMvcApplication” e escolha “**Run File**”. A aba output do NetBeans deve mostrar um *log* semelhante ao da figura a seguir, indicando que o servidor interno Tomcat do Spring está rodando. Note ainda que o processo não encerra, segue rodando, mas você pode interrompê-lo clicando no botão “stop” na lateral esquerda dessa aba.

```
Output - Run (ProjectomvcApplication) x Usages |  
--- exec-maven-plugin:3.0.0:exec (default-cli) @ projectomvc ---  
  
      .  
     /\ /__' _--_-( ) _--_ \\\ \\  
    ( ( )\_ _| '_|'_||'_ _\ \ \ \ \  
   \| \_ )|_|_|_|_|_|_|_|_|_|_| ) ) )  
    ' |_ _|'_ _|_|_|_|_|_|_|_|_|_|_|  
=====|_|=====|_|_/=//_/_/_/  
:: Spring Boot ::                (v3.0.6)  
  
2023-05-11T10:42:46.454-03:00 INFO 17340 --- [           main] c.s.projectomvc.ProjectomvcApplication : Starting Pr  
2023-05-11T10:42:46.458-03:00 INFO 17340 --- [           main] c.s.projectomvc.ProjectomvcApplication : No active p  
2023-05-11T10:42:48.015-03:00 INFO 17340 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat init  
2023-05-11T10:42:48.035-03:00 INFO 17340 --- [           main] o.apache.catalina.core.StandardService : Starting se  
2023-05-11T10:42:48.036-03:00 INFO 17340 --- [           main] o.apache.catalina.core.StandardEngine : Starting Se  
2023-05-11T10:42:48.209-03:00 INFO 17340 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializin  
2023-05-11T10:42:48.211-03:00 INFO 17340 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApp
```

Figura 18 – Oputput do Netbeans com o projeto em execução

Fonte: NetBeans (2023)

Sabendo que o projeto está em execução, é possível ir a um navegador e usar a seguinte URL: `http://localhost:8080/saudacao`. A resposta deverá ser a seguinte:

Boas vindas!

Parabéns, este é seu primeiro projeto MVC em ação!

Figura 19 – Página de boas-vindas

Fonte: Senac EAD (2023)

O que aconteceu aqui?

Sua requisição GET para `http://localhost:8080/saudacao` atingiu o servidor Tomcat iniciado pela execução de seu projeto.

O servidor redirecionou o processamento ao DispatcherServlet, interno ao projeto, que encontrou seu controller “MiscController”.



Na classe de controle, encontrou o método com anotação `@GetMapping("/saudacao")`, ou seja, a URL informada na requisição, e executou `mostraSaudacao()`, retornando o texto “saudacao”.

`DispatcherServlet`, internamente, com base no retorno “saudacao” que recebeu do controller, procurou na pasta `src/main/resources/templates` por um arquivo “saudacao.html” e o encontrou.

Com isso, a classe preparou o conteúdo HTML presente nesse arquivo e retornou como resposta da requisição.

O navegador então renderizou a página na tela.

Note que o nome do arquivo HTML depende do retorno do método executado no *controller*, mas não depende da URL. Experimente trocar o argumento de `@GetMapping` para “/ola”, pare a execução com “stop” na aba “output” e rode de novo a aplicação usando a URL `http://localhost:8080/ola` no navegador. A resposta será idêntica, provando que a URL não interfere no nome dos arquivos de visão (HTML) usados, e vice-versa.

Crie um projeto Spring MVC com duas páginas estáticas apenas. Cada página deve ter um *link* apontando para a outra (em href, use algo como “/pagina1” ou “/pagina2” para referenciar a página).

No exemplo, serve-se apenas uma página estática, mas se fosse ser criado um *site* tão simples, não necessitaria de tamanho aparato. O objetivo de programar *back-end* é criar um sistema *web* dinâmico, que ajusta informações da página HTML de acordo com um processamento ou dados de usuário. Para experimentar de maneira simples esse aspecto, será criado um novo método na classe `MiscController`.

```
@GetMapping("/diahora")
public String mostraDiaHora(){
    return "data";
}
```

Na pasta “templates”, será criado o arquivo “data.html”.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Data e Hora</title>
```

```

<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale
=1.0">

</head>
<body>
    <h1>Que dia é hoje? Que horas são?</h1>
    <p><strong>Data e hora atuais:</strong><span id="spnDataHora">d
d/mm/aaaa hh:mm</span></p>
</body>
</html>

```

Devem ser mostrados dentro do o dia e a hora atuais. Mas enquanto isso não ocorre, trata-se apenas de mais uma página estática. Como enviar dados da camada de controle para a de visão? No método `mostraDiaHora()`, primeiro são calculadas a data e a hora atuais.

```

@GetMapping("/diahora")
public String mostraDiaHora(){
    LocalDateTime diaHora = LocalDateTime.now();
    return "data";
}

```

É necessário `“import java.time.LocalDateTime;”`.

Já existe o objeto `“diaHora”` com a informação necessária. No entanto, ela ainda não foi enviada para a visão. Isso ocorre por meio da classe `Model` do Spring MVC. Cada requisição tem acesso a um objeto dessa classe com informações que trafegarão entre camadas. Neste método, para ter acesso a ele, é necessário incluir um parâmetro do tipo `Model`; o objeto desse parâmetro então receberá a informação que se deseja trafegar.

```

@GetMapping("/diahora")
public String mostraDiaHora(Model model){
    LocalDateTime diaHora = LocalDateTime.now();
    model.addAttribute("dataHora", diaHora.toString());
    return "data";
}

```

Model necessita de `“import org.springframework.ui.Model;”`.

Note no código que primeiro há um parâmetro com nome `“model”` do tipo `“Model”` (o nome pode ser outro, se necessário). `Model` é uma interface do Spring MVC usada para transferir dados da camada Controller para a camada View. Ela trabalha com um mapa de

chaves-valores, ou seja, dados identificados por uma chave única. Por isso, usa-se o método “addAttribute()”, em que são informados uma chave (“dataHora”) e o valor (o objeto diaHora convertido para texto). Dessa forma, está sendo adicionado o atributo “dataHora” que será usado na visão.

Já está sendo enviada a informação pelo controle, mas como recebê-la na visão? Existem algumas abordagens para isso. A mais clássica é usar a linguagem JSP em vez de HTML e aplicar *tags* específicos para essa comunicação.

Neste caso, será usada a biblioteca **Thymeleaf**, incluída na criação do projeto, para ajudar nessa comunicação. A vantagem sobre o JSP é que o Thymeleaf preserva a estrutura HTML com mudanças muito pontuais. Em “data.html”, inclua a seguinte alteração:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Data e Hora</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        <h1>Que dia é hoje? Que horas são?</h1>
        <p><strong>Data e hora atuais:</strong><span id="spnDataHora" th:text="${dataHora}">dd/mm/aaaa hh:mm</span></p>
    </body>
</html>
```

A primeira alteração acontece no *tag* <html>, que recebe atributos para reconhecer os outros atributos especiais de Thymeleaf. A segunda alteração é em : usa-se “**th:text**” para alterar o conteúdo do span; a notação “\${ }” serve para indicar que o valor vem do *back-end* (da interface **Model**, mais precisamente); dataHora é exatamente a chave do valor incluído no objeto de Model. Assim o que th:text="\${dataHora}" faz é trocar o texto “dd/mm/aaaa hh:mm” para o valor da data calculado no *controller*.

Reinicie a execução e digite http://localhost:8080/diahora no navegador. O resultado deve ser parecido com a figura a seguir.



Que dia é hoje? Que horas são?

Data e hora atuais:2023-05-11T12:07:38.505149200

Figura 20 – Resultado da requisição a “localhost:8080/diahora”

Fonte: Senac EAD (2023)

E se você quisesse separar, mostrar em uma linha o dia e em outra a hora? Basta informar duas chaves em Model e incluir modificações no HTML.

Em MiscController:

```
@GetMapping("/diahora")
public String mostraDiaHora(Model model){
    LocalDateTime diaHora = LocalDateTime.now();
    model.addAttribute("data", diaHora.toLocalDate().toString());
    model.addAttribute("hora", diaHora.toLocalTime().toString());
    return "data";
}
```

Em data.html:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Data e Hora</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <h1>Que dia é hoje? Que horas são?</h1>
    <p><strong>Hoje é dia:</strong><span id="spnData" th:text="${data}" ></span></p>
    <p><strong>A hora certa é:</strong><span id="spnHora" th:text="${hora}" ></span></p>
  </body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Data e Hora</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
```



```
<body>
  <h1>Que dia é hoje? Que horas são?</h1>
  <p><strong>Hoje é dia:</strong><span id="spnData" th:text="${data}" ></span></p>
  <p><strong>A hora certa é:</strong><span id="spnHora" th:text="${hora}" ></span></p>
</body>
</html>
```

O resultado será algo próximo a seguinte figura:

Que dia é hoje? Que horas são?

Hoje é dia:2023-05-11

A hora certa é:12:14:54.212565800

Figura 21 – Resultado da requisição a “localhost:8080/diahora” após ajustes

Fonte: Senac EAD (2023)

Ótimo, você já sabe como comunicar dados da camada de controle para a camada de visão, mas, nesse exemplo, foi informado um dado computado pelo próprio método. Então como receber dados do usuário? A maneira mais simples é informando diretamente na URL, como, por exemplo, em “http://localhost:8080/temperatura?valor=20”, em que está sendo informado “20” para o parâmetro “valor”. Coloque isso em prática criando um novo método na classe `MiscController` que receba uma temperatura em Celsius e mostre na tela a temperatura em Fahrenheit e em Kelvin.

```
@GetMapping("/temperatura")
public String converteTemperatura(Model model, int valor){
    float fahrenheit, kelvin;
    kelvin = valor + 273;
    fahrenheit = 1.8f*valor + 32;
    model.addAttribute("celsius", valor);
    model.addAttribute("fahrenheit", fahrenheit);
    model.addAttribute("kelvin", kelvin);
    return "temperatura";
}
```

Nota-se, pelo código, que é muito simples usar parâmetros de requisição, basta defini-los como um parâmetro do próprio método, desde que tenham tipo primitivo ou `String` (nesse caso, há “`int valor`”). O parâmetro de tipos internos do Spring ou de Java Web, como `Model` no código acima, não são considerados como parâmetros de usuário.

No corpo do método, são calculados os valores Fahrenheit e Kelvin e incluídos como atributos de Model os três valores de temperatura, que serão exibidos na visão.

Na pasta “templates”, é criado o arquivo “temperatura.html”.

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Conversor de temperatura</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" /
>

</head>
<body>
  <h1 th:text="'Convertendo ' + ${celsius} + ' graus Celsius'"/>
  <ul>
    <li th:text="${fahrenheit} + ' graus Fahrenheit'"/>
    <li th:text="${kelvin} + ' graus Kelvin'"/>
  </ul>
</body>
</html>
```

Há uma particularidade nesse exemplo na hora de montar o texto dos elementos HTML. Deve-se concatenar um texto fixo com um valor variável que será recebido por Model. Assim, resulta a expressão:

```
'Convertendo ' + ${celsius} + ' graus Celsius'
```

Note a necessidade de **aspas simples**: elas delimitam o texto literal. Além disso, são necessárias **aspas duplas** envolvendo toda a expressão por conta da sintaxe do HTML. Assim, a expressão completa fica:

```
" 'Convertendo ' + ${celsius} + ' graus Celsius' "
```

A atenção a essas aspas é muito importante, pois pode gerar erro na hora de executar o projeto.

Caso você tenha se esquecido de alguma aspa nas expressões de Thymeleaf com concatenação, ocorrerá um erro na aba *output* do NetBeans com a exceção “TemplateProcessingException”. Por exemplo: org.thymeleaf.exceptions.TemplateProcessingException: Could not parse the expression: "\${fahrenheit} + ' graus Fahrenheit' (template: "temperatura" - line 10, col 13).

Agora você pode testar a aplicação. Rode o projeto no NetBeans e experimente a URL “http://localhost:8080/temperatura?valor=15” no navegador.

Convertendo 15 graus celsius

- 59.0 graus Fahrenheit
- 288.0 graus Kelvin

Figura 22 – Resultado do teste de conversão de temperatura

Fonte: Senac EAD (2023)

Um problema acontece se você simplesmente não informar o parâmetro “valor” na URL. Ele é obrigatório, pois “int” não permite valor nulo. Você pode usar a anotação **@RequestParam** para configurar um valor padrão para o parâmetro, além de outras opções. Essa anotação conta com os seguintes elementos opcionais:

defaultValue: valor-padrão quando não é informado na URL. Exemplo:

@RequestParam(defaultValue="0")int valor. Nesse caso, passará o valor 0 ao parâmetro “valor” quando não for informado nada na requisição.

name: nome esperado para o parâmetro na URL. Exemplo:

@RequestParam(name="celsius", defaultValue)int valor. Nesse caso, na URL deveria ser informado `http://localhost:8080/temperatura?celsius=15`.

required: indica se o parâmetro é obrigatório ou não. Exemplo:

@RequestParam(required=true)int valor. Por padrão, tem valor “false”.

Aplique a anotação **@RequestParam** ao parâmetro “valor” do método `converteTemperatura()` na classe `MiscController` de maneira que “valor” receba um valor-padrão quando não informado.

```
@GetMapping("/temperatura")
public String converteTemperatura(Model model, @RequestParam(defaultValue="0")int valor){
    float fahrenheit, kelvin;
    kelvin = valor + 273;
    fahrenheit = 1.8f*valor + 32;
    model.addAttribute("celsius", valor);
    model.addAttribute("fahrenheit", fahrenheit);
    model.addAttribute("kelvin", kelvin);
    return "temperatura";
}
```

Para `@RequestParam`, é necessário incluir `import org.springframework.web.bind.annotation.RequestParam`.

Agora é possível usar a URL `http://localhost:8080/temperatura` para não ocorrer erros.

Lembre-se sempre de parar e rodar novamente o projeto quando fizer alterações e for testá-las.

Crie um projeto Spring MVC com uma página de cálculo, que receba dois valores inteiros pelo parâmetro da URL e mostre na tela a adição, a subtração, a multiplicação e a divisão desses valores.

Thymeleaf também suporta objetos em sua sintaxe para embutir valores em uma página HTML. Agora em um novo exemplo, primeiramente você criará uma classe de camada Model. No projeto “ProjetoMvc”, crie o pacote “com.senac.projetoMvc.controller.model” e, nele, a classe “Pessoa.java”, com o seguinte código:

```
package com.senac.projetoMvc.controller.model;

public class Pessoa {
    String nome;
    String sobrenome;
    int idade;
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public String getSobrenome() { return sobrenome; }
    public void setSobrenome(String sobrenome){ this.sobrenome = sobrenome; }

    public int getIdade() { return idade; }
    public void setIdade(int idade) { this.idade = idade; }
}
```

Agora em `MiscController`, crie uma nova ação, com o seguinte método:

```
@GetMapping("/pessoa")
public String mostraPessoa(Model model, String nome, @RequestParam(defaultValue="0") int idade){
    Pessoa p = new Pessoa();
    if(nome != null && !nome.isBlank()) {
        //extrair sobrenome
        int indiceEspaco = nome.indexOf(" ");
        if(indiceEspaco < 0) //não encontrou caractere de espaço
            indiceEspaco = nome.length(); // assume então a posição do último caractere
    }
}
```

```
        p.setNome( nome.substring(0, indiceEspaco) );
        p.setSobrenome(nome.substring(indiceEspaco, nome.length()) );
    }
    p.setIdade(idade);
    model.addAttribute("pessoa", p);
    return "pessoa";
}
```

Nesse método, recebe-se um nome completo por parâmetro (String nome), quebrando em nome e sobrenome para preencher um novo objeto Pessoa. Também é recebida uma idade (int idade), que será passada ao objeto Pessoa criado.

Agora crie a visão pessoa.html na pasta “templates”.

```
<!DOCTYPE HTML>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Dados da pessoa</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    </head>
    <body>
        <h1>Dados da pessoa</h1>
        <p>Nome:<span th:text="${pessoa.nome}" /></p>
        <p>Sobrenome:<span th:text="${pessoa.sobrenome}" /></p>
        <p>Idade:<span th:text="${pessoa.idade}" /></p>
    </body>
</html>
```

Note que está sendo usado `${pessoa.nome}`, por exemplo, onde “pessoa” é o nome do atributo de Model informado no método `mostraPessoa()` do controle, nesta linha:

```
model.addAttribute("pessoa", p);
```

Ao atributo “pessoa”, então vai um objeto “p” do tipo Pessoa e “nome” é uma de suas propriedades. Assim, `${pessoa.nome}` obtém o valor da propriedade “nome” do objeto de tipo “Pessoa” presente no atributo “pessoa” de Model.

Importante notar que é necessário que, na classe desse objeto, haja *getters* e *setters* para a propriedade referenciada.

Ao testar com a URL “localhost:8080/pessoa?nome=José da Silva&idade=45”, o resultado deve ser próximo ao da figura a seguir.

Dados da pessoa

Nome: José

Sobrenome: da Silva

Idade: 45

Figura 23 – Resultado do teste

Fonte: Senac EAD (2023)

Ao executar uma página, caso apareça a mensagem “Whitelabel Error Page”, informando algo como “There was an unexpected error (type=Internal Server Error, status=500)”, volte ao Netbeans e verifique na aba Output se há Stack de exceção informando erros. A partir dessa informação, você poderá compreender o que aconteceu de errado com a página e corrigir para executá-la.

Em um último teste do “projetomvc”, será mostrada uma página com formulário e estudado como realizar a postagem dos valores digitados pelo usuário para o *back-end*. Primeiramente, será criado, na classe `MiscController`, um método `mostraCadastro()` que responderá pela URL “/cadastro”.

```
@GetMapping("/cadastro")
public String mostraCadastro(Model model){
    model.addAttribute("pessoa", new Pessoa());
    return "cadastro";
}
```

Nada diferente do que você aprendeu até agora.

Será criada a visão “cadastro.html” na pasta “templates”, essa sim com características bem específicas.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">

    <head>
        <title>Cadastro de Pessoa</title>
```

```

<meta http-equiv="Content-Type" content="text/html; charset=UTF-
8" />

</head>
<body>
  <h1>Cadastro de pessoa</h1>
  <form th:action="@{/cadastro}" th:object="${pessoa}" method="POST">

    <p>Nome: <input type="text" th:field="*{nome}" /> </p>
    <p>Sobrenome: <input type="text" th:field="*{sobrenome}" />

    <p>Idade: <input type="text" th:field="*{idade}" /></p>
    <p><input type="submit" value="Cadastrar" /></p>
  </form>
</body>
</html>

```

A página tem um formulário simples com campo para nome, sobrenome e idade, mas algumas marcações do Thymeleaf chamam a atenção:

th:action="@{/cadastro}" forma o atributo “action” do *tag* `<form>`.

th:object="\${pessoa}" associa o formulário ao objeto Pessoa recebido por Model. Caso ele venha com dados, eles aparecerão nos elementos `<input>` associados a cada propriedade do objeto.

th:field="*{nome}" associa um elemento de *input* com uma propriedade do objeto Pessoa associado ao formulário. Isso significa que: 1) os elementos receberão os valores informados pelo objeto ao executar a requisição (e o método `mostraCadastro()`, no caso) e 2) preencherá a propriedade associada do objeto Pessoa que será enviado quando a postagem for realizada.

Por agora, você pode até testar a página, verificar se está tudo certo. Mas, para a postagem de dados, ainda são necessários alguns passos. Será criado um novo método na classe `MiscController`:

```

@PostMapping("/cadastro")
public String recebeCadastro(Model model, @ModelAttribute Pessoa pessoa)
{
    model.addAttribute("pessoa", pessoa);
    return "pessoa";
}

```

É necessário incluir `“import org.springframework.web.bind.annotation.ModelAttribute;”` e `“import org.springframework.web.bind.annotation.PostMapping;”`.

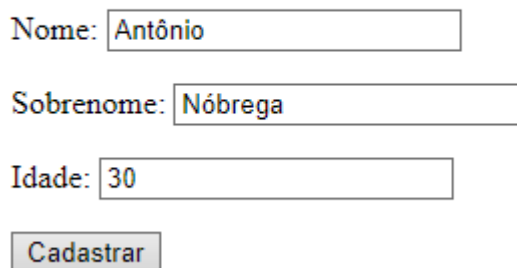
Note aqui duas características importantes. A primeira é a presença da anotação **@PostMapping**. Ela indica que o método só será executado quando a requisição for do tipo “POST” para a URL “localhost:8080/cadastro”. Não por acaso, “/cadastro” também é o valor incluído no atributo “action” do formulário na visão cadastro.html (em `th:action="@{/cadastro}"`). Caso esteja na visão um nome diferente, não alcançará o método e ele não executará.

A outra é **@ModelAttribute**, indicando que os dados do formulário que virão no Model por conta da postagem estarão no objeto “pessoa” do parâmetro. Na prática, significa que o objeto “pessoa” conterá as informações que o usuário digitou no formulário ao realizar a postagem.

Nesse caso, seria possível validar os dados recebidos, calcular sobre eles ou incluí-los no banco de dados. No entanto, como o exemplo é simples, apenas será reexibido o que foi postado e, para isso, usa-se novamente a visão “pessoa.html”. Por isso, ‘return “pessoa”’ ao fim do método.

Enfim teste por completo essas novas funcionalidades, primeiramente acessando “http://localhost:8080/cadastro” e preenchendo os dados.

Cadastro de pessoa



Formulário de cadastro de pessoa com os seguintes campos:

- Nome:
- Sobrenome:
- Idade:
- Botão:

Figura 24 – Página mostrando formulário desenvolvido

Fonte: Senac EAD (2023)

Depois de clicar em “Cadastrar”, os dados passam por “recebeCadastro()”, que redireciona para a página pessoa.html com as informações digitadas no formulário. Assim, o resultado será:



Dados da pessoa

Nome: Antônio

Sobrenome: Nóbrega

Idade: 30

Figura 25 – Resultado após postagem

Fonte: Senac EAD (2023)

Ajuste o projeto de calculadora do desafio anterior para agora receber os dois números a partir de um formulário. Ao submeter, mostre as quatro operações matemáticas sobre eles.

Com isso, você experimentou os aspectos fundamentais do Spring MVC. A seguir, serão aprofundados o Thymeleaf e outros temas do framework, trabalhando em novos exemplos.

THYMELEAF

Nesta seção, será explorada a fundo a biblioteca **Thymeleaf**, uma poderosa ferramenta de renderização de *templates* que permite a criação de aplicações *web* dinâmicas em Java. Ao longo do conteúdo, você foi introduzido ao Thymeleaf e, até mesmo, fez uso dos seus recursos para construir sua aplicação *web*. Agora você entenderá como ele funciona e aprenderá a aplicá-lo em um exemplo de projeto utilizando, em conjunto, o **Spring Web** e os conceitos de padrão de projeto com a arquitetura MVC.

O Thymeleaf é uma biblioteca de *templates* para desenvolvimento de aplicações *web* em Java. Ele permite a criação de páginas HTML dinâmicas, em que é possível integrar facilmente conteúdo estático e dinâmico. O Thymeleaf é altamente flexível, facilitando a criação de *layouts* reutilizáveis e a integração com outros *frameworks* e bibliotecas.

O Thymeleaf é amplamente utilizado no desenvolvimento de aplicações *web* com Java devido a sua facilidade de uso e poder de expressão. Algumas vantagens de se utilizar o Thymeleaf incluem a sua integração com o *framework* Spring, sua sintaxe simples baseada em atributos HTML e a criação de *templates* reutilizáveis e personalizáveis.

Criação de um novo projeto



Antes de começar, será configurado um exemplo de projeto para demonstrar o uso do Thymeleaf. Para isso, será criado um projeto no Spring Initializr com as seguintes configurações:

Project: Maven

Language: Java

Spring Boot: maior versão estável disponível

Group: com.senac

Artifact/Name: exemplothymeleaf

Packaging: Jar

Java: 17 (ou versão superior instalada em sua máquina)

Dependencies:

Spring Web

Thymeleaf

Baixe o arquivo compactado gerado, extraia o conteúdo e abra a pasta no NetBeans.

Estrutura do projeto

Neste projeto, será usada a arquitetura MVC como padrão de projeto. Para isso, é necessário dividir o conteúdo em três camadas: Model, View e Controller. Na teoria, seriam criados três pacotes na estrutura do projeto, mas, na prática, isso não será feito, já que a biblioteca Thymeleaf será responsável pelo conteúdo de *front-end*, ou seja, a camada *View*. Será usada a sua estrutura pré-configurada para desenvolver essa camada. Tal estrutura pode ser acessada no NetBeans, no caminho “Other Sources > src/main/resources”. Se você acessá-la agora, encontrará dois pacotes vazios:

static: pacote destinado a recursos estáticos, como arquivos CSS, Javascript, imagens ou qualquer outro tipo de arquivo multimídia.

templates: pacote destinado aos arquivos HTML que contêm marcações HTML combinadas com expressões e atributos especiais do Thymeleaf. Aqui ficarão os *templates* que serão processados pelo Thymeleaf no lado do servidor para gerar o conteúdo dinâmico que será enviado para o navegador do usuário.

“*Templates*” no Thymeleaf são códigos HTML com elementos próprios da ferramenta para incluir dados de *back-end*.

Isso significa que só será necessário criar dois pacotes em “com.senac.exemplothymeleaf” para a arquitetura MVC: o pacote *model* e o pacote *controller*. A estrutura do projeto ficará da seguinte maneira:

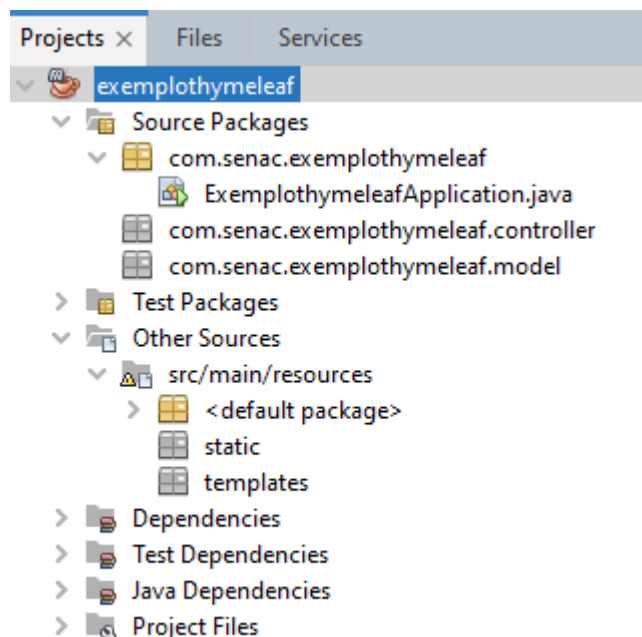


Figura 26 – Projeto exemplothymeleaf no NetBeans

Fonte: Senac EAD (2023)

Nesse exemplo, será criado um projeto de lista de tarefas como contexto para explorar recursos do Thymeleaf, como a manipulação de dados dinâmicos, *loops*, condicionais e formulários. Mas antes, serão criadas mais duas classes muito úteis para esse projeto.

Model

A classe modelo será criada dentro do pacote “model” com o nome “Tarefa” e terá o seguinte código:

```
package com.senac.exemplothymeleaf.model;

public class Tarefa {
    private int id;
    private String descricao;
    private boolean completa;
    public Tarefa() {
    }
    public Tarefa(int id, String descricao, boolean completa) {
        this.id = id;
        this.descricao = descricao;
        this.completa = completa;
    }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    public String getDescricao() { return descricao; }
    public void setDescricao(String descricao) { this.descricao = descricao; }

    public boolean isCompleta() { return completa; }
    public void setCompleta(boolean completa) { this.completa = completa; }
}
```

Controller

A classe de controle será criada dentro do pacote “controller” com o nome “TarefaController” e terá o seguinte código:

```
@Controller

public class TarefaController {
    private List<Tarefa> tarefas = new ArrayList();
    // Abaixo, criaremos nossos métodos novos
}
```

Perceba que, além da estrutura-base, também foi criada uma lista de tarefas, que será bastante útil para métodos futuros.

Manipulação de dados dinâmicos



Uma das principais vantagens do Thymeleaf é a capacidade de manipular dados dinâmicos em *templates* HTML. Esse recurso permite exibir e modificar valores de variáveis, propriedades de objetos e, até mesmo, trabalhar com coleções de dados.

Para exemplificar a manipulação de dados, será criado um arquivo HTML no diretório “src/main/resources/templates” com o nome “exibir-tarefa.html”. Nesse arquivo, haverá o seguinte conteúdo:

```
<!DOCTYPE html>
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Tarefa</title>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <h1>Tarefa</h1>
  <p>Bem-vindo, <span th:text="${nome}" />!</p>
  <br>
  <p>ID: <span th:text="${tarefa.id}" /></p>
  <p>Descrição: <span th:text="${tarefa.descricao}" /></p>
  <p>Completa: <span th:text="${tarefa.completa}" /></p>
</body>
</html>
```

Serão desmembrados alguns trechos desse código:

```
xmlns="w3.org/1999/xhtml"
```

Para começar, veja algumas definições. O XML (*Extensible Markup Language*) é uma linguagem de marcação que permite a criação de documentos com dados estruturados. O XHTML (*Extensible Hypertext Markup Language*) é uma variação do XML que segue as regras de sintaxe do HTML, tornando-o compatível com navegadores *web*.

A declaração do namespace "**xmlns="w3.org/1999/xhtml"**" indica que o documento XML ou XHTML está utilizando os elementos e atributos definidos pelo padrão XHTML, especificados pela *World Wide Web Consortium* (W3C). A finalidade dessa declaração de namespace é garantir que os elementos e os atributos utilizados no documento XML ou

XHTML estejam corretamente interpretados. Na prática, está sendo usada essa declaração para que o NetBeans consiga diferenciar a sintaxe HTML comum e a sintaxe XML. Do contrário, a IDE apontará erros de sintaxe nos trechos de código XML que estiverem no documento “.html”.

Logo em seguida, há outro namespace:

```
xmlns:th="http://www.thymeleaf.org"
```

Nessa linha, existe o atributo **xmlns:th** definido para o elemento **html**. Para entender melhor o que isso significa, veja essa declaração em partes:

xmlns: é uma abreviação de "XML namespace".

th: é o prefixo que está sendo atribuído ao namespace.

"http://www.thymeleaf.org": é o URL do namespace do Thymeleaf. O URL é uma sequência de caracteres que identifica de forma exclusiva o namespace.

Quando o Thymeleaf é usado em um documento HTML, é necessário declarar o namespace do Thymeleaf para que o HTML possa reconhecer e interpretar corretamente os atributos e as expressões Thymeleaf usados no documento. Essa declaração informa ao analisador HTML que todos os atributos que começam com **th:** são atributos específicos do Thymeleaf e devem ser interpretados pelo mecanismo de processamento do Thymeleaf. Isso permite que você use recursos e funcionalidades do Thymeleaf, como expressões, *loops*, condicionais e assim por diante, dentro do seu documento HTML. Portanto, a declaração `html xmlns:th="http://www.thymeleaf.org"` no início do código HTML garante que o Thymeleaf seja ativado e habilitado para processar as *tags* e os atributos Thymeleaf usados no documento.

E ainda nesse código, pode ser visto na prática como exibir o valor de uma variável e como acessar as propriedades de objetos, usando a variável **\${nome}** e o objeto **tarefa**. Mas esses valores ainda não têm nenhum dado, e para que tudo funcione, é necessário fazer essa definição antes de renderizar a página. Portanto, deve ser adicionado no controlador o seguinte trecho de código:

```
@GetMapping("/exibir-tarefa")
public String exibirTarefa(Model model) {
```

```
// Definindo o valor da variável "nome"
String nome = "Lucas";
// Criando uma nova instância da classe "Tarefa" com os valores espe
cificados
Tarefa tarefa = new Tarefa(1, "Aprender como usar dados dinâmicos co
m o Thymeleaf", true);
// Adicionando a variável "nome" ao modelo para ser usada na visuali
zação (view)
model.addAttribute("nome", nome);
// Adicionando o objeto "tarefa" ao modelo para ser usado na visuali
zação (view)
model.addAttribute("tarefa", tarefa);
// Retornando o nome da visualização (view) que será renderizada
return "exibir-tarefa";
}
```

Salve tudo e execute a classe principal do seu projeto para iniciar o servidor *web*.

Após isso, abra o navegador *web* e acesse a URL <http://localhost:8080/exibir-tarefa>.

Tarefa

Bem-vindo, Lucas!

ID: 1

Descrição: Aprender como usar dados dinâmicos com o Thymeleaf

Completa: true

Figura 27 – Resultado do projeto em execução

Fonte: Senac EAD (2023)

Loops

No Thymeleaf, os *loops* são usados para iterar sobre uma coleção de elementos e processá-los repetidamente. O Thymeleaf fornece uma diretiva específica chamada **th:each** para implementar *loops*. Você verá isso em um exemplo prático.

Crie um novo arquivo HTML chamado “lista-tarefas” e adicione o seguinte conteúdo:

```
<!DOCTYPE html>
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Lista de tarefas</title>
</head>
<body>
  <h1>Lista de tarefas</h1>
  <p>Essas são todas as suas tarefas:</p>
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Nome</th>
        <th>Completa</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="tarefa : ${tarefas}">
        <td th:text="${tarefa.id}"></td>
        <td th:text="${tarefa.descricao}"></td>
        <td th:text="${tarefa.completa}"></td>
      </tr>
    </tbody>
  </table>
</body>
</html>
```

Nesse exemplo, você tem uma página HTML que renderiza uma lista de tarefas usando Thymeleaf. Veja a análise do trecho específico que trata do loop:

```
<tr th:each="tarefa : ${tarefas}">
  <td th:text="${tarefa.id}"></td>
  <td th:text="${tarefa.descricao}"></td>
  <td th:text="${tarefa.completa}"></td>
</tr>
```

Aqui, está sendo usada a diretiva **th:each** para iterar sobre a coleção de tarefas **\${tarefas}**. Cada elemento da coleção é atribuído à variável local **tarefa** destacada no código acima.

Dentro do *loop*, está sendo definida uma linha da tabela para cada tarefa. Usa-se a diretiva **th:text** para definir o conteúdo de cada célula da tabela com os atributos da tarefa correspondente. **\${tarefa.id}** representa o ID da tarefa, **\${tarefa.descricao}** representa a descrição da tarefa e **\${tarefa.completa}** representa se a tarefa está completa ou não.

A estrutura básica da tabela é definida fora do *loop*, com as colunas de cabeçalho “ID”, “Nome” e “Completa”. Dentro do *loop*, a tag `<tr>` é repetida para cada tarefa, resultando em uma nova linha na tabela para cada elemento da coleção.

É importante que a variável `${tarefas}` contenha a coleção de tarefas que se deseja exibir na tabela e, para isso, será usada aquela lista de tarefas inserida no controlador quando foi criada a classe `TarefaController`.

Adicione o seguinte trecho de código à classe `TarefaController`:

```
@GetMapping("/lista-tarefas")
public String listaTarefas(Model model) {
    // Criação de objetos Tarefa e adição à coleção tarefas
    tarefas.add(new Tarefa(1, "Aprender como usar dados dinâmicos com o
Thymeleaf", true));
    tarefas.add(new Tarefa(2, "Aprender como usar loops com o Thymelea
f", true));
    tarefas.add(new Tarefa(3, "Aprender como usar condicionais com o Thy
meleaf", false));
    // Adição da coleção tarefas ao modelo
    model.addAttribute("tarefas", tarefas);
    // Retorna o nome da página que será renderizada
    return "lista-tarefas";
}
```

Salve tudo e reinicie o servidor *web*. Ao acessar a URL `http://localhost:8080/lista-tarefas`, você deve ter o seguinte resultado:

Lista de tarefas

Essas são todas as suas tarefas:

ID	Nome	Completa
1	Aprender como usar dados dinâmicos com o Thymeleaf	true
2	Aprender como usar loops com o Thymeleaf	true
3	Aprender como usar condicionais com o Thymeleaf	false

Figura 28 – Listagem de tarefas

Fonte: Senac EAD (2023)

Condicionais



No Thymeleaf, você pode usar condicionais para exibir ou ocultar elementos HTML com base em certas condições. Existem várias maneiras de usar condicionais no Thymeleaf, mas a abordagem mais comum é usar a diretiva `th:if`.

Será usado o arquivo “lista-tarefas.html” para esse exemplo. Nele há uma lista de tarefas `${tarefas}` que será iterada usando a diretiva `th:each`. Para cada tarefa na lista, exibimos seu ID, descrição e se está concluída ou não (`true/false`) em células da tabela. Mas apresentar esse `true/false` não seria muito amigável para o usuário final. Então, serão usadas condicionais para personalizar o *status* da tarefa.

```
<!DOCTYPE html>
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Lista de tarefas</title>
</head>
<body>
    <h1>Lista de tarefas</h1>
    <p>Essas são todas as suas tarefas:</p>
    <table>
        <thead>
            <tr>
                <th>ID</th>
                <th>Nome</th>
                <th>Completa</th>
            </tr>
        </thead>
        <tbody>
            <tr th:each="tarefa : ${tarefas}">
                <td th:text="${tarefa.id}"></td>
                <td th:text="${tarefa.descricao}"></td>
                <td>
                    <!-- Usamos if para verificar se o valor é verdadeiro
o -->
                    <span th:if="${tarefa.completa}">Concluída</span>
                    <!-- Usamos unless para verificar se o valor é falso
-->
                    <span th:unless="${tarefa.completa}">Pendente</span>
                </td>
            </tr>
        </tbody>
    </table>
</body>
</html>
```

É usada a diretiva **th:if** para verificar se o valor de **`\${tarefa.completa}`** é verdadeiro. Se for verdadeiro, é exibida a mensagem "Concluída". Caso contrário, usa-se a diretiva **th:unless** para verificar se o valor é falso e, nesse caso, é exibida a mensagem "Pendente".

Lista de tarefas

Essas são todas as suas tarefas:

ID	Nome	Completa
1	Aprender como usar dados dinâmicos com o Thymeleaf	Concluída
2	Aprender como usar loops com o Thymeleaf	Concluída
3	Aprender como usar condicionais com o Thymeleaf	Pendente

Figura 29 – Lista de tarefas com texto adequado na coluna “Completa”

Fonte: Senac EAD (2023)

Você também pode usar condicionais para iterar sobre uma lista de elementos e exibir apenas os que atendem a determinadas condições. Veja um exemplo:

```
<!-- Usamos a condicional exibir apenas as tarefas pendentes -->  
<tr th:each="tarefa : ${tarefas}" th:if="${!tarefa.completa}">
```

Formulários

Agora você verá como trabalhar com formulários usando o Thymeleaf. Crie um novo arquivo HTML chamado “cadastro” e adicione o seguinte conteúdo:

```
<!DOCTYPE html>  
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">  
  <head>  
    <title>Cadastro de tarefas</title>  
  </head>  
  <body>  
    <h1>Cadastro de tarefas</h1>  
    <form th:action="@{/cadastro}" method="post">  
      <div>  
        <p>ID: <input type="text" th:field="${tarefa.id}" /> </p>  
        <p>Descrição: <input type="text" th:field="${tarefa.descricao}" /> </p>  
      </div>
```

```
<button type="submit">Enviar</button>
</form>
</body>
</html>
```

Repare que, nesse exemplo, está sendo usado o atributo **th:field**. O **th:field** permite associar um campo de formulário a um atributo específico em um objeto. Na prática, ao usar o **th:field**, o Thymeleaf gera o HTML com o atributo **name**. Isso significa que, ao utilizar o “**th:field**”, você evita a necessidade de escrever manualmente os atributos “**name**” para cada campo do formulário, pois o Thymeleaf cuida disso automaticamente com base no atributo associado ao campo, garantindo que o valor do campo seja transmitido corretamente durante o envio do formulário e também ao exibir o formulário preenchido com os valores corretos.

Quando o formulário for enviado, o Thymeleaf fará o *binding* dos valores preenchidos nos campos diretamente nos atributos correspondentes do objeto **tarefa**. Isso permite acessar e usar esses valores no controlador como desejado. Aqui você também é apresentado a uma nova sintaxe do Thymeleaf, que é **@{...}**.

Diferença entre **\${...}** e **@{...}**

No Thymeleaf, as expressões **@{...}** e **\${...}** têm finalidades diferentes:

@{...}: Essa sintaxe é usada para fazer referências a URLs relativas ou absolutas. É comumente usada em *links*, formulários e outras *tags* HTML que exigem URLs. As expressões **@{...}** são avaliadas pelo Thymeleaf e substituídas por URLs correspondentes com base na configuração do aplicativo. A vantagem dessa sintaxe é que ela leva em consideração o contexto da URL e pode lidar com a resolução de URLs relativas automaticamente.

\${...}: Essa sintaxe é usada para acessar variáveis ou expressões avaliadas no contexto do modelo Thymeleaf. As expressões **\${...}** são avaliadas no lado do servidor e substituídas pelo valor correspondente. Essa sintaxe é útil para acessar dados dinâmicos no modelo ou realizar operações em tempo de execução.

Portanto, em resumo, **@{...}** é usado para manipular URLs, enquanto **\${...}** é usado para acessar variáveis e expressões no contexto do modelo.

O que leva ao próximo passo, que é configurar o controlador. Nele, serão criados dois métodos: um para exibir a tela do formulário e outro para processar o envio do formulário. Em `TarefaController`, adicione o código abaixo.

```
@GetMapping("/cadastro")
    public String exibirFormulario(Model model) {
        // Cria uma nova instância da classe Tarefa e a adiciona ao modelo
        model.addAttribute("tarefa", new Tarefa());
        // Retorna o nome do template "cadastro" para exibir o formulário
        return "cadastro";
    }
@PostMapping("/cadastro")
    public String processarFormulario(@ModelAttribute Tarefa tarefa, Model model) {
        // Define a propriedade "completa" da tarefa como "false"
        tarefa.setCompleta(false);
        // Adiciona a tarefa à lista de tarefas (supondo que "tarefas" seja
        // uma lista já existente)
        tarefas.add(tarefa);
        // Adiciona a tarefa ao modelo para ser exibida no template "exibir-
        // tarefa"
        model.addAttribute("tarefa", tarefa);
        // Retorna o nome do template "exibir-tarefa" para exibir os detalhes
        // da tarefa
        return "exibir-tarefa";
    }
```

Agora salve tudo, reinicie o servidor *web* e acesse a URL `http://localhost:8080/cadastro` para testar o formulário.

Ao acessar a tela de cadastro, preencha os campos “ID” e “Descrição” como desejar. Após isso, clique em “Enviar”.

Cadastro de tarefas

ID:

Descrição:

Figura 30 – Formulário de cadastro

Fonte: Senac EAD (2023)

Você será levado a uma nova tela, onde os dados serão exibidos.



Tarefa

Bem-vindo, !

ID: 4

Descrição: Cadastrar uma tarefa.

Completa: false

Figura 31 – Dados da tarefa cadastrada

Fonte: Senac EAD (2023)

Agora que o cadastro e a listagem de dados estão funcionando, você pode comentar alguns trechos de código do projeto para deixá-lo mais consistente. O primeiro deles é no controlador, no trecho que faz a criação de objetos Tarefa e adição à lista de tarefas em **listaTarefas()**. Se manter como está, toda vez que você acessar a página “Lista de tarefas”, o método será acionado e a coleção “tarefas” receberá dados duplicados. Outro método que pode ser editado é o **exibirTarefa()**, comentando o trecho que passa o valor para a variável **nome** e removendo, também, a linha que exibe essa informação na View **exibir-tarefa**. Dessa forma, não haverá um texto “solto” na tela de exibição da tarefa.

Crie um projeto de agenda pessoal que permita cadastrar e consultar nome e telefone de pessoas. O cadastro usará <form> e a consulta <table>. Experimente ainda implementar exclusão de registro.

Navegação entre páginas

Com o Thymeleaf, é possível criar uma barra de navegação reutilizável em várias páginas utilizando fragmentos. Os fragmentos são partes de um *template* que podem ser incluídas em outros *templates*. Veja na prática como fazer isso.

1. Crie um novo arquivo HTML chamado “navbar” e insira o seguinte código:

```
<nav>
```

```
<a th:href="@{/lista-tarefas}">Lista d  
e tarefas</a>  
  
<a th:href="@{/cadastro}">Cadastro</a>  
</nav>
```

2. Agora, nos outros *templates* de página, inclua o fragmento da barra de navegação usando a diretiva **th:insert**, como no exemplo a seguir:

```
<!DOCTYPE html>  
<html xmlns:th="http://www.thymeleaf.org">  
<head>  
<title>Minha Página</title>  
</head>  
<body>  
<header th:insert="navbar.html :: nav"></h  
eader>  
  
<!-- Conteúdo da página -->  
...  
</body>  
</html>
```

Nesse exemplo, o fragmento da barra de navegação é incluído usando a diretiva **th:insert**. A sintaxe "**navbar.html :: nav**" especifica que você deseja incluir o fragmento com o nome "**nav**" do arquivo **navbar.html**. O conteúdo desse fragmento será renderizado no local onde a diretiva **th:insert** está definida no *template* da página.

Certifique-se de que o arquivo **navbar.html** e os *templates* de página estão no mesmo diretório para que o Thymeleaf possa encontrá-los corretamente. Dessa forma, você pode criar uma barra de navegação em um único arquivo e reutilizá-la em várias páginas do seu projeto Thymeleaf. Isso facilita a manutenção e evita a duplicação de código.

Veja a seguir um exemplo de como a sua página “lista-tarefas” deve ficar após essa adição.

[Lista de tarefas](#) [Cadastro](#)

Cadastro de tarefas

ID:

Descrição:

Figura 32 – Formulário com *links* de navegação

Fonte: Senac EAD (2023)

Recursos estáticos

Dentro da pasta de recursos do Thymeleaf, há o diretório **static**, que é utilizado para guardar recursos estáticos que serão referenciados e usados no código HTML, como arquivos CSS, Javascript, imagens, fontes etc.

É de boa prática criar subdiretórios nesse local de acordo com os tipos de recursos que se deseja guardar. Por exemplo, você pode ter os subdiretórios “css”, “js” e “images” para armazenar arquivos CSS, Javascript e imagens, respectivamente. Essa organização ajuda a manter os recursos estáticos estruturados e facilita o acesso e a manutenção.

O Spring MVC tem a capacidade de mapear e servir automaticamente os recursos estáticos do diretório **src/main/resources/static** para as solicitações HTTP correspondentes. Isso significa que você pode simplesmente referenciar os recursos estáticos nos seus *templates* ou páginas HTML e o *framework* se encarregará de servir esses arquivos corretamente.

Veja isso na prática!

Criação de pastas

Para começar, serão criadas três novas pastas: “css”, “js” e “images”. Crie uma nova pasta dentro do diretório **static** chamada “css” por meio do NetBeans clicando com o botão direito do mouse sobre “Other Sources > src/main/resources > static” e selecionando “New” > “Folder...”.

Você perceberá que o diretório passará a se chamar **static.css**. Isso acontece, pois a pasta *static* não tem nenhum arquivo, apenas uma subpasta, então o NetBeans ignora essa pasta vazia. Se você tentar criar uma nova pasta agora, por exemplo, “js”, o NetBeans entenderá que quer criá-la dentro de “css”. Então atenção ao próximo passo!

Na janela de criação da nova pasta, altere o conteúdo “Parent Folder” e remova o trecho que aponta para a pasta “css”. O conteúdo deve ficar “src\main\resources\static” conforme a imagem a seguir.

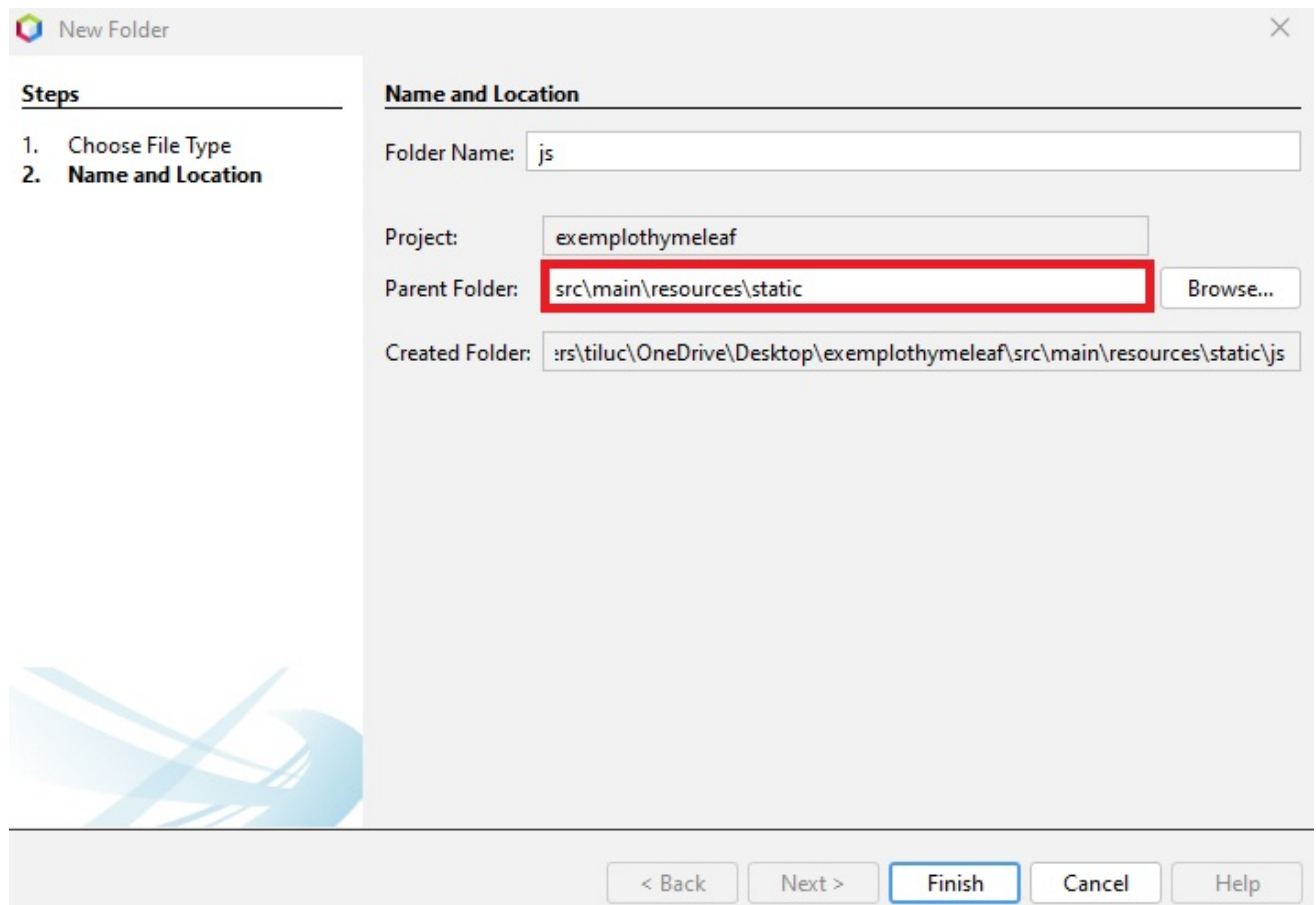


Figura 33 – Campo “Parent Folder” na criação de pasta

Fonte: NetBeans (2023)

Repita o procedimento até que as três pastas tenham sido criadas e você tenha a seguinte estrutura:

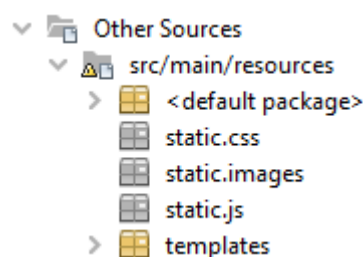


Figura 34 – Estrutura de pastas estáticas

Fonte: NetBeans (2023)



Arquivo CSS

Após isso, crie dentro do diretório “css” um arquivo chamado **style.css**. Esse será o arquivo de estilização.

Para esse exemplo, você fará algo bem simples! Lembre-se de que as tarefas estão sendo listadas com as mensagens “Concluída” e “Pendente”. Destaque esses textos aplicando a cor azul para as tarefas concluídas e laranja para as tarefas pendentes. O código CSS ficará assim:

```
.tarefa-concluida {  
    color: blue;  
}  
.tarefa-pendente {  
    color: orange;  
}
```

Agora volte para o arquivo **lista-tarefas.html**. Nele, serão feitas duas mudanças:

1. Primeiramente, será chamado o arquivo CSS que acabou de ser criado. Para isso, adicione entre as *tags* **<head></head>** o seguinte conteúdo:

```
<link rel="stylesheet" th:href="@{/css/style.css}" />
```

O uso da expressão `@{...}` é uma sintaxe específica do Thymeleaf para resolver caminhos de recursos estáticos. Quando o Thymeleaf processa o *template*, ele substituirá a expressão `@{/css/style.css}` pelo caminho correto para o arquivo CSS no contexto do seu aplicativo. Durante a execução do aplicativo, o navegador carregará o arquivo CSS com base no caminho fornecido pelo Thymeleaf e aplicará as regras de estilo definidas no arquivo "style.css".



O uso da expressão `@{...}` é necessário apenas para arquivos estáticos que estão dentro da estrutura do projeto. Caso você deseje referenciar um caminho externo, como o CDN da biblioteca *Bootstrap*, utilize a sintaxe padrão do HTML.

2. Agora serão aplicadas as classes de estilo nas *tags* `` onde estão os textos “Concluída” e “Pendente”. Esse trecho de código ficará assim:

```
<td>
    <span th:if="${tarefa.completa}" class
="tarefa-concluida">Concluída</span>
    <span th:unless="${tarefa.completa}" c
lass="tarefa-pendente">Pendente</span>
</td>
```

O resultado final será esse:

[Lista de tarefas Cadastro](#)

Lista de tarefas

Essas são todas as suas tarefas:

ID	Nome	Completa
1	Tarefa 1	Pendente
2	Tarefa 2	Pendente

Figura 35 – Lista de tarefas estilizada

Fonte: Senac EAD (2023)

Você pode criar um novo fragmento (igual foi feito com a navegação) para o seu `<head>`. Assim, você não precisará atualizar todos os seus arquivos HTML para chamar o CSS em cada um deles. Essa dica também vale para os *scripts* JS.

Arquivo JS

Crie um novo arquivo chamado **script.js** dentro do diretório **js** e adicione o seguinte trecho de código:

```
function atualizarTarefa(element) {  
    if (element.innerHTML === 'Concluída') {  
        element.innerHTML = 'Pendente';  
        element.className = 'tarefa-pendente';  
    } else {  
        element.innerHTML = 'Concluída';  
        element.className = 'tarefa-concluida';  
    }  
}
```

Veja o que será feito a seguir:

1. Criar a função “atualizarTarefa” para ser acionada quando o usuário clicar sobre o *status* de uma tarefa.
2. Quando o usuário “clique” sobre o *status* da tarefa, será atualizado o valor. Se o *status* estiver “Pendente”, após o clique ficará como “Concluída”, e vice-versa.
3. Além disso, usar o *className* para atualizar a classe de estilo do elemento. Dessa forma, a estilização ainda destacará o texto. Será usada a propriedade *className*.

Perceba que o código não atualizará de fato o conteúdo da lista de tarefas que está no *back-end*. Ele apenas atualizará a visualização no *front-end*.

Agora será atualizado o `lista-tarefas.html`:

```
<!DOCTYPE html>  
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">  
<head>  
    <title>Lista de tarefas</title>  
    <link rel="stylesheet" th:href="@{/css/style.css}" />  
</head>  
<body>  
    <header th:insert="navbar.html :: nav"></header>  
    <h1>Lista de tarefas</h1>  
    <p>Essas são todas as suas tarefas:</p>  
    <table>  
        <thead>  
            <tr>  
                <th>ID</th>  
                <th>Nome</th>  
                <th>Completa</th>  
            </tr>  
        </thead>
```

```

<tbody>
  <tr th:each="tarefa : ${tarefas}">
    <td th:text="${tarefa.id}"></td>
    <td th:text="${tarefa.descricao}"></td>
    <td onclick="atualizarTarefa(this)">
      <span th:if="${tarefa.completa}" class="tarefa-concl
uida">Concluída</span>
      <span th:unless="${tarefa.completa}" class="tarefa-p
endente">Pendente</span>
    </td>
  </tr>
</tbody>
</table>

<script th:src="@{/js/script.js}"></script>

</body>

</html>

```

Além de adicionar o evento “onclick” à *tag* <td> que corresponde ao *status* da tarefa, também foi feita a chamada do arquivo script.js na penúltima linha de código:

```
<script th:src="@{/js/script.js}"></script>
```

[Lista de tarefas Cadastro](#)

Lista de tarefas

Essas são todas as suas tarefas:

ID	Nome	Completa
1	Tarefa 1	Pendente

Figura 36 – Lista de tarefas atualizada

Fonte: Senac EAD (2023)

Arquivos de imagem

Existem duas formas de inserir imagens nas páginas *web*. Uma é por meio do HTML e a outra do CSS.

Para esse exemplo, as imagens foram salvas dentro do diretório **images** criado em `src/main/resources/static/`:



Figura 37 – Ícone de Tarefa

Fonte: Icon Shop (c2020)



Figura 38 – Banner de tarefas

Fonte: Unsplash (s.d.)

Na página “lista-tarefas.html”, adicione o seguinte trecho de código abaixo do título

“Lista de tarefas”:

```

```

Com isso, o arquivo **tarefa-icone.png** será chamado com o seguinte resultado:

[Lista de tarefas Cadastro](#)

Lista de tarefas



Essas são todas as suas tarefas:

ID Nome Completa

Figura 39 – Página de listagem de tarefas agora com um ícone

Fonte: Senac EAD (2023)

Agora você verá como fazer o mesmo utilizando a propriedade `background-image` no CSS. Mas antes serão exploradas algumas situações que podem ocorrer no seu dia a dia como programador.

Imagine que você está trabalhando em um projeto *web* com o Spring Web e o Thymeleaf. Você precisa criar um *banner*, que ficará no topo da página e terá 200px de altura. Esse *banner* precisa ser exibido em todas as páginas do sistema. Veja como seria possível solucionar esse problema.

Primeiramente, o mais apropriado aqui seria usar fragmentos para ter reaproveitamento de código. No caso deste exemplo de projeto, já há um fragmento, que é o arquivo `navbar.html`. Então nele será feita uma pequena adição. No início do código, adicione o seguinte trecho antes da tag

```
<div class="banner"></div>
```



Isso criará uma área que está sendo referenciada com a classe de estilização “banner”. Com isso, não há mais apenas o conteúdo das tags <nav>. Isso significa que é necessário atualizar o trecho em que é chamado esse fragmento nos arquivos HTML, removendo a chamada exclusiva **::nav**. O código atualizado ficará assim:

```
<header th:insert="navbar.html"></header>
```

Agora, crie essa classe no CSS adicionando o seguinte trecho de código:

```
.banner {  
    background-image: url('../images/banner.jpg');  
    background-repeat: no-repeat;  
    background-size: cover;  
    height: 200px;  
}
```

Perceba que aqui não foi utilizada a sintaxe **@{...}** do Thymeleaf, e o motivo é bem simples: o CSS é um arquivo separado e não tem suporte direto ao processamento de expressões do Thymeleaf. Portanto, não se pode usar a sintaxe **@{...}** dentro do CSS. No entanto, pode-se usar a notação relativa de caminho dentro do CSS para referenciar corretamente a imagem. Ao usar **../** no caminho da imagem, volta-se um nível no diretório em relação ao arquivo CSS e, em seguida, pode-se acessar a pasta "static/images" para encontrar o arquivo desejado.

Dessa forma, a imagem deve ser exibida corretamente como plano de fundo com o seguinte resultado:



[Lista de tarefas](#) [Cadastro](#)

Lista de tarefas



Essas são todas as suas tarefas:

ID Nome Completa

Figura 40 – Listagem de tarefas agora com banner

Fonte: Senac EAD (2023)

Confira a seguir o código completo dos arquivos HTML alterados.

navbar.html

```
<div class="banner"></div>
<nav>
  <a th:href="@{/lista-tarefas}">Lista de tarefas</a>
  <a th:href="@{/cadastro}">Cadastro</a>
</nav>
```

lista-tarefa.html

```
<!DOCTYPE html>
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Lista de tarefas</title>
  <link rel="stylesheet" th:href="@{/css/style.css}" />
</head>
<body>
  <header th:insert="navbar.html"></header>
  <h1>Lista de tarefas</h1>
  
  <p>Essas são todas as suas tarefas:</p>
  <table>
    <thead>
      <tr>
```

```
<th>ID</th>
<th>Nome</th>
<th>Completa</th>
</tr>
</thead>
<tbody>
<tr th:each="tarefa : ${tarefas}">
<td th:text="${tarefa.id}"></td>
<td th:text="${tarefa.descricao}"></td>
<td onclick="atualizarTarefa(this)">
<span th:if="${tarefa.completa}" class="tarefa-concl
uida">Concluída</span>
<span th:unless="${tarefa.completa}" class="tarefa-p
endente">Pendente</span>
</td>
</tr>
</tbody>
</table>
<script th:src="@{/js/script.js}"></script>
</body>
```

Bibliotecas externas

No desenvolvimento de projetos de *software*, é comum fazer uso de várias bibliotecas para auxiliar no processo. Neste exemplo de projeto que está sendo desenvolvido, foram usadas duas bibliotecas para essa finalidade: o Spring Web e o Thymeleaf. Mas isso não significa que há apenas essas duas opções.

Por exemplo, no estágio atual deste projeto de tarefas, existe uma sólida implementação do *back-end* e um *front-end* simples, porém funcional, que atende às interações necessárias para acionar as funcionalidades do *back-end*. Para melhorar a interface visual do projeto, seria necessária uma grande dedicação de tempo na construção do CSS que aplica a estilização e torna a página mais bonita. Ou, para economizar tempo, pode-se usar uma biblioteca de estilização para auxiliar nessa tarefa, como o *Bootstrap*.

Agora, como adicionar a biblioteca do *Bootstrap* ao projeto? É o que você verá a seguir.

Existem duas formas de fazer isso:

1. Adicionar a biblioteca via CDN: essa abordagem permite incluir o Bootstrap rapidamente no projeto fazendo a referência dos arquivos com as URLs

públicas disponíveis oficialmente. Para isso, basta adicionar a referência do arquivo CSS na *tag* `<head>` do HTML:

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-KK94CHFLLe+nY2dmCWGMq91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJ0Z" crossorigin="anonymous">
```

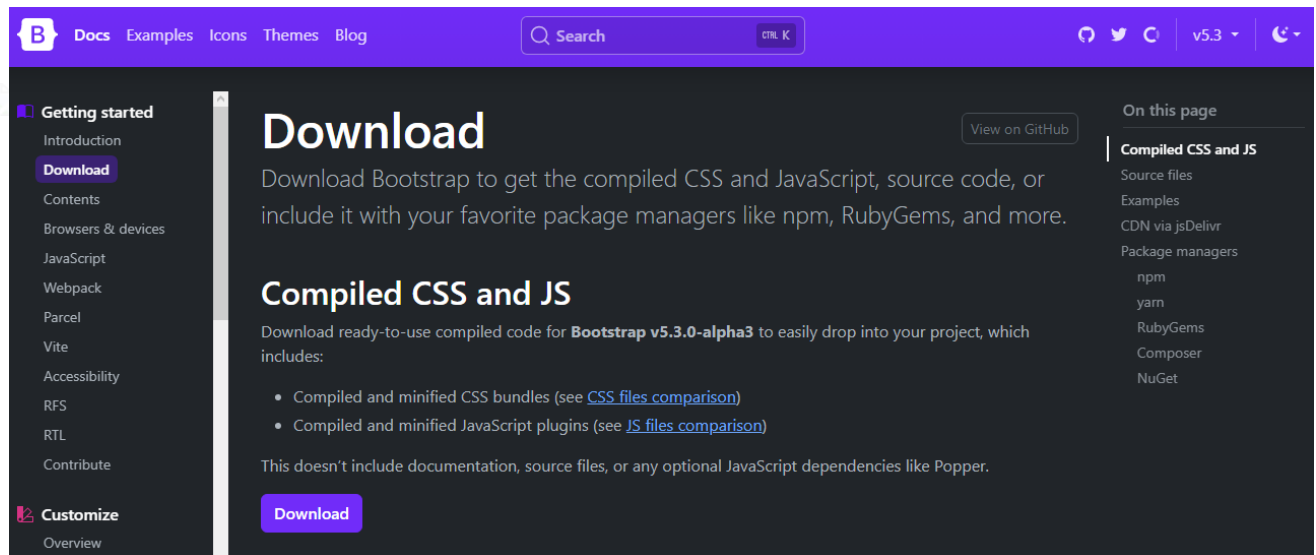
E também adicionar a referência do arquivo Javascript no final do arquivo HTML antes do fechamento da *tag* `</body>`:

```
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ENjd04Dr2bkBIFxQpeoTz1HIcje39Wm4jDKdf19U8gI4ddQ3GYNS7NTKfAdVQSZe" crossorigin="anonymous">
```

Mas fazer isso significa tornar o projeto dependente de um provedor externo para entregar os arquivos, o que pode resultar em problemas de desempenho, disponibilidade e carregamento do *site*. Apesar de essa prática ser conveniente no ambiente de desenvolvimento de *software*, pois permite o uso da biblioteca mais rápido, em algum momento, os arquivos terão que ser adicionados à estrutura do projeto para que o *software* não seja entregue com essa dependência de provedores externos.

2. Adicionar os arquivos ao diretório do projeto: nessa abordagem, é feito o *download* dos arquivos CSS e JS diretamente do *site* oficial da biblioteca e extraídos os arquivos dentro do diretório do `src/main/resources/static` do projeto. Essa é a abordagem mais apropriada, pois o projeto não dependerá de provedores externos para requisitar os arquivos estáticos. Por isso, será usada essa abordagem para este exemplo de projeto.

Primeiramente, é necessário acessar o *site* oficial do Bootstrap e fazer o *download* dos arquivos CSS e JS compilados – você encontrará o botão de *download* desses arquivos na seção “Compiled CSS and JS”. Nesse exemplo, foi usada a versão 5.3, pois essa é a última versão disponibilizada até então da biblioteca.

Figura 41 – Página de *download* do Bootstrap

Fonte: Bootstrap (s.d.)

Após clicar no botão de *download*, será baixada uma pasta compactada (.zip) contendo uma pasta chamada Bootstrap e o número da versão baixada. Extraia essa pasta no diretório “src/main/resources/static” do seu projeto e renomeie apenas para “bootstrap” – isso facilitará ao escrever o nome do diretório no código. Se você fez tudo corretamente, terá a seguinte estrutura aparecendo no seu NetBeans:

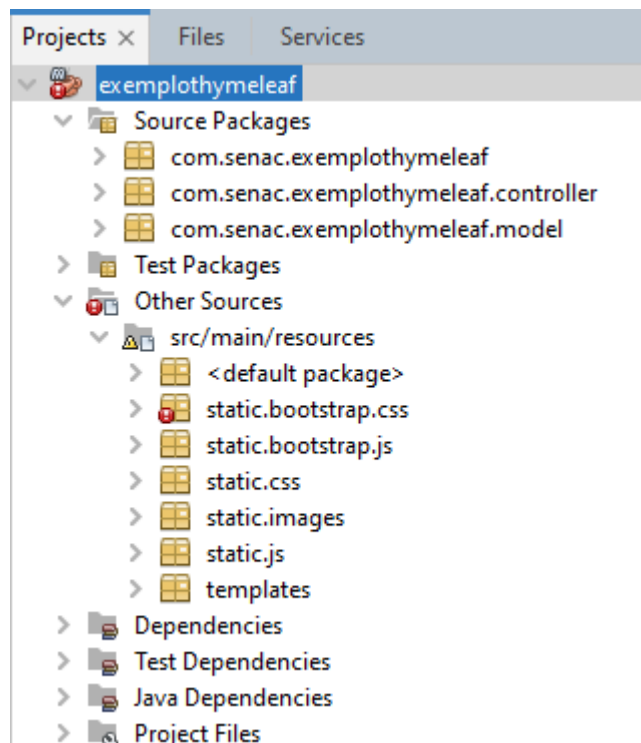



Figura 42 – Estrutura de arquivos do projeto

Fonte: NetBeans (2023)

Se algum erro aparecer no seu “static.bootstrap.css” ou “static.bootstrap.js”, como na  imagem, não se preocupe. Provavelmente, a versão baixada do Bootstrap trata-se de uma versão que ainda está em desenvolvimento e não foi totalmente revisada. Por isso, alguns trechos de código podem conter a sintaxe errada e o NetBeans se encarregará de apontar esses erros, mas nada disso afetará a compilação do seu projeto.

Nesse exemplo, as subpastas “css” e “js” estão dentro da pasta “bootstrap” para deixar os arquivos isolados, e não misturados com os próprios arquivos “css/style.css” e “js/script.js”. É comum as pastas “css” e “js” serem extraídas diretamente no repositório “static”, deixando de lado a pasta “bootstrap” para simplificar a hierarquia de arquivos. Mas essa organização não é uma regra e cada desenvolvedor pode organizar seus arquivos estáticos da maneira que melhor atende à estrutura do seu projeto. Em caso de muitas bibliotecas externas serem utilizadas, por exemplo, é comum ser criada uma pasta “libraries” ou “lib” para se referir a toda biblioteca externa sendo utilizada pelo projeto. Se apenas uma biblioteca externa estiver sendo utilizada, o uso da pasta “libraries” acaba sendo apenas mais um caminho no diretório que precisa ser acessado para chegar aos arquivos da biblioteca desejada. Então tudo depende do que melhor atende às necessidades dos desenvolvedores e melhor se enquadra com a complexidade do projeto.

Agora basta fazer a chamada dos arquivos no projeto. Para esse exemplo, serão adicionados o CSS e o JS do Bootstrap na “página lista-tarefas.html”.

Como os arquivos da estrutura do projeto são estáticos, segue-se com o uso da sintaxe `@{...}` para referenciar o caminho dos arquivos. Então serão usados os seguintes trechos de código para adicionar os arquivos.

Para o CSS:

```
<link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}">
```

Para o JS:

```
<script th:src="@{/bootstrap/js/bootstrap.min.js}"></script>
```

Lembrando que é importante que a chamada desses arquivos ocorra antes da chamada dos arquivos CSS e JS próprios. Do contrário, o CSS do Bootstrap pode reescrever a estilização de uma classe com o mesmo nome que esteja em "style.css", por exemplo. O mesmo vale para as funções do Javascript.

Logo, o código deve ficar assim:

```
<!DOCTYPE html>
<html xmlns="w3.org/1999/xhtml" xmlns:th="http://www.thymeleaf.org">
<head>
  <title>Lista de tarefas</title>
  <!-- CSS do Bootstrap -->
  <link rel="stylesheet" th:href="@{/bootstrap/css/bootstrap.min.css}"/>

  <!-- CSS próprio-->
  <link rel="stylesheet" th:href="@{/css/style.css}" />
</head>
<body>
  <header th:insert="navbar.html"></header>
  <h1>Lista de tarefas</h1>
  <p>Essas são todas as suas tarefas:</p>
  <table>
    <thead>
      <tr>
        <th>ID</th>
        <th>Nome</th>
        <th>Completa</th>
      </tr>
    </thead>
    <tbody>
      <tr th:each="tarefa : ${tarefas}">
        <td th:text="${tarefa.id}"></td>
        <td th:text="${tarefa.descricao}"></td>
        <td onclick="atualizarTarefa(this)">
          <span th:if="${tarefa.completa}" class="tarefa-concluida">Concluída</span>
          <span th:unless="${tarefa.completa}" class="tarefa-pendente">Pendente</span>
        </td>
      </tr>
    </tbody>
  </table>

  <script th:src="@{/bootstrap/js/bootstrap.min.js}"></script>

  <script th:src="@{/js/script.js}"></script>
</body>
</html>
```

Executando o projeto agora, já é possível perceber que tudo está funcionando corretamente devido a estilização do Bootstrap no elemento “body”:

Figura 43 – Página estilizada com Bootstrap

Fonte: Senac EAD (2023)

Incremente o sistema de agenda proposto no desafio anterior, incluindo nele a biblioteca Bootstrap e imagens. Também teste implementar funcionalidades JavaScript onde considerar conveniente e adequado.

Encerramento

O Spring MVC é uma poderosa ferramenta que permite a construção de sistemas *web* completos sob o padrão MVC.

Ferramentas de *templates* são necessárias, e aqui foi trabalhado bastante com o Thymeleaf, uma das ferramentas mais usadas atualmente. Por meio de seus recursos, é possível construir páginas dinâmicas com facilidade, sem alterar a estrutura do documento HTML.

A partir dos exemplos estudados, você poderá experimentar com seus próprios projetos e expandir seus conhecimentos – a própria documentação do Spring pode ser um bom caminho para aprofundar seus conhecimentos.