Desenvolvimento Web III

Professor Jefferson Chaves
jefferson.chaves@ifpr.edu.br

ATIVIDADE AVALIATIVA

Lista de Exercícios 2

O objetivo desta atividade é desenvolver uma aplicação didática de controle de tarefas (Todo List), com o intuito de compreender a separação entre as camadas `Model`, `View` (utilizando templates no Spring) e `Controller`. A proposta visa demonstrar, na prática, como ocorre a troca de dados entre essas camadas, evidenciando o papel de cada uma na arquitetura MVC.

Além disso, a aplicação permitirá entender como o `Controller` manipula os dados provenientes da `Model` e os envia para as `Views`, bem como como as informações inseridas pelos usuários nas `Views` (formulários, links e demais interações) são capturadas e processadas pelos `Controllers`.

Desenvolvedores web devem compreender o fluxo completo de uma requisição. Assim, durante o desenvolvimento da aplicação, é importante observar com atenção como os dados trafegam entre as camadas. De forma simplificada, esse processo pode ser descrito da seguinte maneira:

1. O usuário envia dados por meio de um formulário.
2. O `Controller` recebe a requisição, a processa e atualiza a lista de tarefas.
3. A `View` é atualizada e exibe as informações modificadas.

O projeto completo para consulta, pode ser acessado no repositório público https://github.com/IFPR-WebIII/TaskManager.

1. Criando sua aplicação

Inicialmente implemente uma aplicação web usando o framework Java Spring (você pode seguir o passo a passo visto em aula, clicando aqui).

Inclua as seguintes dependências:

- spring-boot-starter-web: para criar as rotas da aplicação e demais recursos para disponibilizar páginas web;
- spring-boot-starter-thymeleaf: para permitir o uso do Thymeleaf, que serve como motor de templates (para renderizar os arquivos HTML com dados dinâmicos, geralmente vindo da controller).
- spring-boot-devtools: módulo (opcional) que ajuda os desenvolvedores durante o processo de desenvolvimento, oferecendo funcionalidades que aumentam a produtividade.

Seu arquivo `pom.xml` deve se parecer com (podem haver outras dependências):

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
</dependencies>
```

O arquivo `pom.xml` é o arquivo de configuração principal de um projeto Maven, uma ferramenta de automação de build usada principalmente em projetos Java. Ele define todas as informações e dependências necessárias para construir, empacotar e gerenciar o ciclo de vida do projeto.

2. Criação da estrutura do Projeto

Com o projeto criado, organizaremos o projeto com uma divisão padrão. Crie os seguinte pacotes dentro da pasta `src/main/java/{Group Id}/{Artifact Id}/`:

- `models` → onde está a classe `Task` (que representa cada tarefa).
- `controllers` → onde está a classe `TaskController` (controla as rotas e as ações).
- `templates` (já existe em `src/main/resources`) → onde ficam os arquivos HTML com Thymeleaf (nossas páginas visuais).

3. Criação da classe Task.java

A classe `Task` é o modelo de dados que representa as tarefas que desejamos gerenciar. Embora não seja obrigatório (Poderíamos usar um monte de Strings para representar uma `Task`), o Spring consegue relacionar melhor os dados quando associados a uma classe que representa os modelos de dados.

Implemente dentro do pacote `models` uma classe chamada `Task.java`. Essa classe deve possuir os seguintes atributos:

```
private Long id;
private String titulo;
private String descricao;
private LocalDate date;
```

Implemente os métodos getters e setters para acessar e modificar esses atributos. Caso sua classe tenha mais de um construtor, é necessário que ela tenha também um construtor vazio.

> O uso de Getters e Setters é um assunto polêmico a alguns anos. Leia um pouco mais sobre isso clicando aqui.

4. A classe TaskController.java

Após a criação da classe `Task`, implementaremos uma classe chamada `TaskController`. O objetivo dessa classe é expor funcionalidades que permitam gerenciar tarefas por meio de requisições HTTP, como adicionar, listar e remover itens da lista.

Como ainda não discutimos outras camadas da aplicação, faremos uma concessão acadêmica: os dados serão armazenados em uma estrutura do tipo `ArrayList` diretamente dentro da própria `Controller`. Embora essa prática não seja recomendada em aplicações reais, do ponto de vista didático essa abordagem pode ser útil, pois:

- Simplifica o foco da atividade, permitindo a compreensão do fluxo entre requisições, processamento e resposta;
- Evita complexidade prematura com bancos de dados ou camadas de serviço (veremos nas próximas aulas);
- Facilita a visualização direta de como os dados são manipulados e refletidos nas Views.

No futuro, construiremos outras aplicações seguindo boas práticas de separação de responsabilidades, incluindo a introdução de uma camada de serviço e persistência adequada com banco de dados.

A classe `TaskController` deve possuir as seguintes rotas para as funcionalidades principais:

Método HTTP	Caminho	Função
GET	"/tasks", "/" e " "	Exibe todas as tarefas (lista)
GET	"/tasks/create"	Exibe o formulário de nova tarefa
POST	"/tasks/create"	Salva uma nova tarefa na lista
GET	"/tasks/edit/{id}"	Exibe o formulário para editar uma tarefa
POST	"/tasks/edit"	Atualiza a tarefa existente
GET	"/tasks/delete/{id}"	Remove a tarefa da lista

4.1. Crie a classe TaskController

Dentro do pacote `controllers`, crie a classe conforme o exemplo abaixo. Adicione o atributo tasks do tipo ArrayList:

```
@Controller
@RequestMapping("/tasks")
public class TaskController {
    private List<Task> tasks = new ArrayList<>();

    public TaskController (){
        Task t1 = new Task(1L, "Task1", "Desc T1", LocalDate.now());
        Task t2 = new Task(2L, "Task2", "Desc T2", LocalDate.now());
        Task t3 = new Task(3L, "Task3", "Desc T3", LocalDate.now());

        tasks.add(t1);
        tasks.add(t2);
        tasks.add(t3);
    }

    //outros métodos vêm aqui
}
```

Repare que essa classe está anotada com `@RequestMapping("/tasks")`. Isso significa que todos os métodos dessa classe deverão ser acessados pela URL `http://localhost:8080/tasks/{rota_do_metodo}`.

Observe também o uso do construtor da classe para pré-popular as `tasks` com três tarefas. Fazer isso nos ajudará a observar os resultados da página de listagem de tasks.

5. Listando as tasks

Para listar as tasks vamos criar e expor um método chamado `listTask`. Esse método tem como objetivo obter todas tasks armazenadas no ArrayList e retornar uma página HTML com essas informações montadas:

```
@GetMapping({ "", "/", "/tasks" })
public String listTask(Model model) {
   model.addAttribute("tasks", tasks);
   return "task-list";
}
```

Inicialmente, observamos pela anotação `@GetMapping({ "", "/", "/tasks" })` que o método responde a requisições HTTP do tipo GET para três caminhos diferentes, útil para tornar o acesso à página mais flexível:

- a raiz ("" ou "/")
- ou o caminho /tasks.

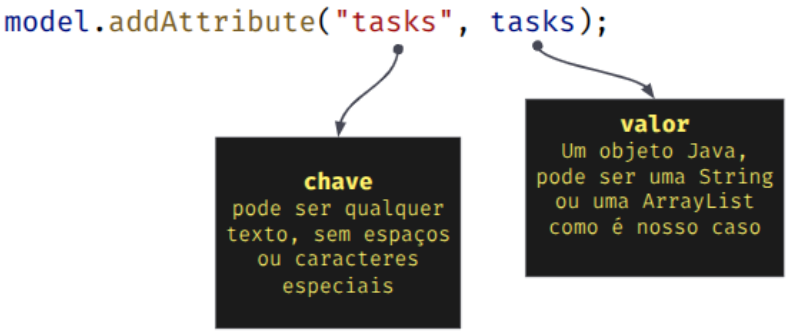
Observe também a assinatura do método: o método retorna uma `String`, que representa o nome do template HTML (`View`) que será renderizado. Já o parâmetro `Model model` é usado para passar dados da Controller para a `View` (um template Thymeleaf, por exemplo).

A lista de `tasks` precisa de alguma forma serem enviados para a View que será renderizada como uma página HTML. Isso será feito por meio do objeto model:

```
model.addAttribute("tasks", tasks);
```

Esse objeto (model), como mencionado anteriormente, é uma "ponte" entre a Controller e a View. O objeto model faz parte do escopo da requisição. Isso significa que esse objeto está disponível para ser acessado a qualquer momento, em qualquer lugar, enquanto durar a requisição. Assim que a View é renderizada e a resposta enviada ao navegador, o objeto model é descartado.

O método `addAttribute` do objeto model, permite adicionar objetos a uma lista do tipo "chave - valor" carregado por esse objeto.



Sua vez! Para seguirmos com nosso projeto, inclua a instrução `model.addAttribute("tasks", tasks)` ao método `listTask` e inclua também a instrução `return "task-list"`. Pronto! Agora você poderá acessar os dados de `tasks` dentro da página `task-list.html` por meio da chave "tasks".

5.1. Crie página `task-list.html`

O objetivo da página `task-list.html` é apresentar todas as tarefas em uma tabela. Para cada tarefa, será exibido o título, descrição, botão de editar e de excluir. Um exemplo dessa página pode ser visto a seguir:

```
<table class="table table-bordered">
    <tr>
        <td>Id</td>
        <td>Título</td>
        <td>Data</td>
        <td>Opções</td>
    </tr>

    <tr th:each="t : ${tasks}">
        <td th:text="${t.id}">Id</td>
        <td th:text="${t.titulo}">Título</td>
        <td th:text="${t.data}">Data</td>
        <td>
            <a th:href="@{/tasks/edit/{id}(id=${t.id})}">editar</a>
            <a th:href="@{/tasks/delete/{id}(id=${t.id})}">excluir</a>
        </td>
    </tr>
</table>
```

O elemento `<tr>` representa uma linha da tabela. Como não sabemos quais e quantas tarefas existem, já que a qualquer momento podem ser cadastradas novas tarefas, usamos a diretiva `th:each` do Thymeleaf que itera sobre uma lista de objetos. Aqui, ele percorre a variável `${tasksList}`, que deve conter uma lista de tarefas.

Para cada item `t` da lista é uma variável temporária que representa a tarefa atual. As colunas de cada linha são preenchidas de forma dinâmica por meio das diretivas th:text, por exemplo `<td th:text="${t.id}">Id</td>`. O conteúdo da célula será

substituído pelo valor de ${t.id}. O texto "Id" será ignorado no resultado final (só será exibido se o Thymeleaf estiver desativado).

Outro ponto de atenção deste formulário é o trecho em que são apresentados os links de opções:

```
<td>
   <a th:href="@{/tasks/edit/{id}(id=${t.id})}">editar</a>
   <a th:href="@{/tasks/delete/{id}(id=${t.id})}">excluir</a>
</td>
```

Esse trecho apresenta, para cada linha, os links para edição e exclusão de uma task. Repare na diretiva `th:href`. Seu objetivo é montar dinamicamente um link (URL).

A expressão `@{}` é uma diretiva de URL do Thymeleaf. Ele serve para criar URLs que se adaptam ao contexto da aplicação. Por exemplo, se a aplicação estiver em desenvolvimento, utilizando endereço de loopback na porta 8080, tal como http://localhost:8080, o link gerado será http://localhost:8080`/tasks/edit.` Mas se a aplicação estiver implantada em no endereço `www.site.com`, o link será `www.site.com/tasks/edit/ ... )`.

O trecho `{id}` em `/tasks/edit/`**`{id}`** indica que esse valor será dinamicamente substituído pelo valor `(id=${t.id}).`

Sua vez! Crie o arquivo `task-list.html` dentro da pasta templates, e depois de adicionar o corpo do HTML, inclua a tabela em sua página, conforme o exemplo acima.

6. Cadastrando tasks

Para realizar o cadastro de `tasks` vamos criar e expor outro método na classe `TaskController` chamado `createTask`. O processo de cadastro será feito em duas etapas: i) a primeira etapa será a implementação de um método que permita chamar o formulário de cadastro de taks; ii) a segunda etapa é implementar outro método para receber os dados vindos do formulário de cadastro de `tasks`.

6.1. Chamando o formulário de Cadastro

Esse método é responsável por chamar um formulário. Muitos se perguntam: faz sentido criar um formulário apenas para isso? Não faria mais sentido chamar o arquivo `.html` diretamente? A resposta depende: chamar o arquivo diretamente pode até ser mais simples. Contudo, temos uma estrutura arquitetural em camadas, em que a Controller é responsável por decidir qual View renderizar. Seguir esse padrão mantém o código organizado, reutilizável e testável. Além disso, hoje o método só exibe o formulário. Mas e amanhã? Você pode precisar enviar uma lista de categorias para formulário, pode querer verificar permissões de acesso a tela ou mesmo preencher valores padrões. Pode querer redirecionar se o usuário não estiver logado.

Sua vez! Vamos implementar o método abaixo que renderiza o formulário de cadastro da `task`:

```
@GetMapping("/create")
public String createTask() {
   return "task-create";
}
```

6.2. O formulário de Cadastro

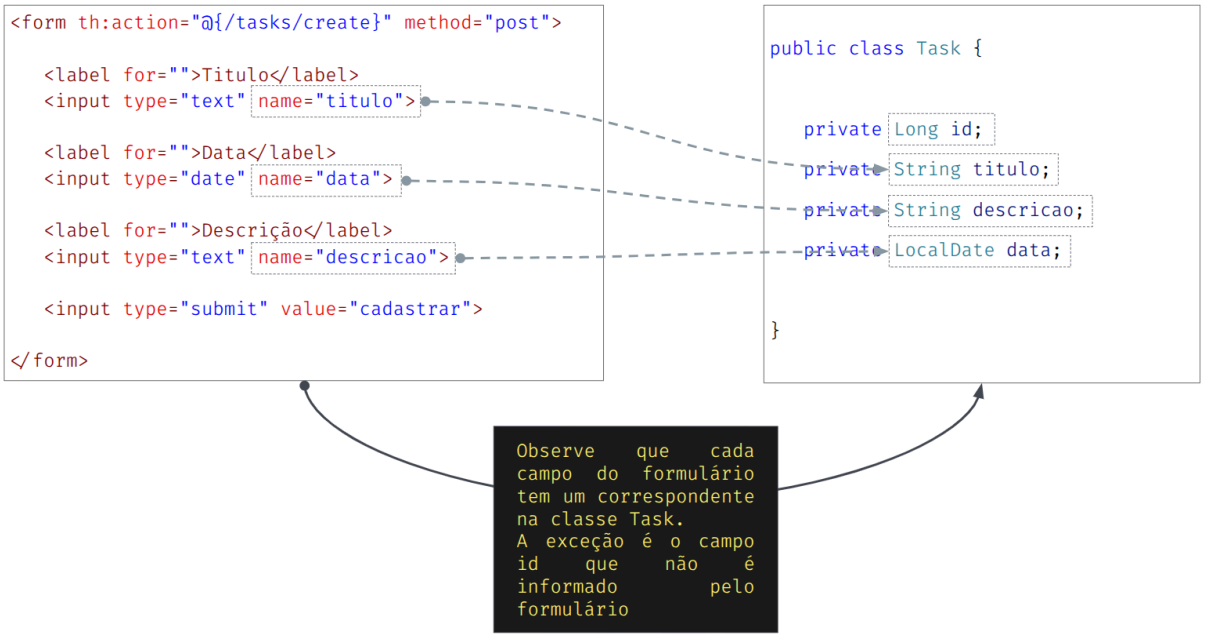
De forma bem simplificada, essa página deve exibir um formulário com os campos: título, data e descrição. Observe o formulário abaixo:

```
<form th:action="@{/tasks/create}" method="post">

   <label for="">Titulo</label>
   <input type="text" name="titulo">

   <label for="">Data</label>
   <input type="date" name="data">

   <label for="">Descrição</label>
   <input type="text" name="descricao">

   <input type="submit" value="cadastrar">
```

```
</form>
```

O endereço informado no atributo `action` do formulário indica que os dados preenchidos pelo usuário serão enviados para o rota `@{/tasks/create}`. Embora seja a mesma rota que exibe o formulário, o atributo `method` indica a mudança do verbo HTTP para POST.

Outro detalhe importante é que para cada input do formulário a propriedade `name` deve estar idêntica aos atributos da classe Task, conforme imagem abaixo:



Quando feito dessa forma, o próprio Spring trata de receber os dados do formulário e instanciar um objeto do tipo `Task`.

Ainda precisamos de um último ajuste: lidar com datas é um desafio em razão dos padrões adotados. Podemos instruir o Spring informando o padrão de data adotado por meio da anotação `@DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)`.

Essa atualização pode ser realizada incluindo a anotação `@DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)` ao atributo `data` da classe `Task` conforme imagem abaixo:

```java
public class Task {

    private Long id;
    private String titulo;
    private String descricao;

    @DateTimeFormat(iso = DateTimeFormat.ISO.DATE_TIME)
    private LocalDate data;
}
```

Sua vez! Crie o arquivo `task-create.html` dentro da pasta templates, e depois de adicionar o corpo do HTML, inclua o formulário em sua página, conforme os exemplos acima. Adicione também a anotação de padronização de data ao atributo.

O padrão ISO8601 é um padrão de datas, independente de linguagem. Saiba mais sobre esse padrão aqui.

Atividades

1. Implemente as funcionalidades para edição e exclusão de tasks;
2. Adicione um novo atributo que permita informar o status de um `task` (em andamento, concluído ou cancelado). Utilize uma enumeração.
    a. No momento da criação da taks, o status deve ser “em andamento”;
    b. No formulário de edição, permita a alteração também do status.
3. Crie uma página que exiba todas as informações de uma `task` incluindo a descrição. Na tabela que lista as tarefas, faça com que o nome da tarefa seja um link para essa página.
4. Crie uma rota que permita a exibição apenas das tarefas em andamento.
5. Crie um link na página de listagem de Tasks que aponte para a rota de cadastro de tasks. Utilize a diretivas `@{}` para criação dos links.