MINICURSO: CRUD SPRING BOOT & THYMELEAF

O **Spring Boot** é uma estrutura (framework) para desenvolvimento de aplicativos Java que visa facilitar e acelerar a criação de aplicativos, fornecendo várias anotações para tornar o desenvolvimento mais simples e eficiente. No Spring, para implementar a Injeção de Dependência (DI) e o Controle de Inversão de Controle (IoC), utilizamos os chamados Beans. Esses Beans são as classes que serão gerenciadas pelo Spring, deixando de ser responsabilidade direta do programador. Com o uso de anotações, podemos transferir essa responsabilidade para o Spring, permitindo que ele cuide da criação, configuração e gerenciamento dos objetos. Dessa forma, a aplicação se torna mais modular e flexível, facilitando o desenvolvimento e a manutenção do código.

Abaixo estão algumas anotações no Spring Boot adotadas no projeto:

Quando o Spring inicia sua aplicação, ele escaneia o classpath procurando por classes anotadas com @Component e outras anotações relacionadas, como @Service, @Repository e @Controller. Essas anotações são especializações de @Component e são usadas para categorizar componentes de acordo com suas funções específicas.

Após a detecção dessas classes anotadas, o Spring criará instâncias dessas classes (beans) e as colocará em seu contêiner, tornando-as disponíveis para serem injetadas em outras classes, por meio da Injeção de Dependência.

@SpringBootApplication: É a classe principal do Spring Boot que inicia a aplicação e é responsável por inicializar o contexto do Spring. Nessa classe, é definido o método main, que é o ponto de entrada da aplicação Java.

@Controller: Essa anotação é usada para marcar uma classe como um controlador (controller) Spring que lida com requisições HTTP e fornece respostas JSON/XML. Os métodos dentro dessa classe são mapeados para rotas específicas e são capazes de processar as solicitações recebidas.

@RequestMapping: Essa anotação é usada para mapear métodos em um controlador (controller) para uma rota HTTP específica. Ela permite definir a URL, o método HTTP e outros atributos para associar o método a uma solicitação.

@Autowired: A anotação @Autowired é usada para injetar dependências automaticamente pelo Spring. A anotação avisa ao Spring Framework para injetar uma instância de alguma implementação da interface ProdutoRepository na propriedade repositorio.

@Service: Essa anotação é usada para marcar uma classe de serviço Spring. Ela é do tipo serviço (Service Layer), que possuem, por exemplo, regras de negócios, deixando claro que a classe é um serviço dentro da lógica de negócios da aplicação.

@Repository: Essa anotação define um bean como sendo do tipo persistente para uso em classes de acesso a banco de dados e o armazenamento em banco de dados. Ela indica que a classe é responsável por interagir com um mecanismo de

persistência, como um banco de dados. A partir desta anotação o Spring pode usar recursos referentes a persistência, como tratar as exceções específicas para este fim.

@Component: é uma das anotações fundamentais do Spring Framework e é usada para identificar classes que devem ser tratadas como componentes gerenciados pelo Spring. Ela é usada em conjunto com a funcionalidade de Injeção de Dependência do Spring, permitindo que o Spring crie e gerencie instâncias dessas classes automaticamente.

Quando uma classe é anotada com **@Component**, o Spring a considera um candidato a ser instanciado e gerenciado pelo contêiner do Spring (ApplicationContext). Essa anotação indica que a classe desempenha o papel de um bean (componente) dentro do contexto do Spring.

Mapeamento de Tabela: A anotação **@Entity** é usada para mapear uma classe Java para uma tabela em um banco de dados relacional. Cada instância da classe será persistida como uma linha (registro) na tabela correspondente.

Identificação da Entidade: Uma classe mapeada com **@Entity** deve ter uma chave primária para identificar exclusivamente cada registro na tabela. A chave primária é mapeada usando a anotação **@Id** em um campo da classe.

Nesse exemplo, a classe Produto é mapeada como uma entidade usando **@Entity**. O campo id é marcado com **@Id**, combinado com a anotação **@GeneratedValue(strategy = GenerationType.AUTO)** indicando que é a chave primária da entidade. Juntas, essas anotações definem como o valor da chave primária será gerado. Os outros campos, nome e preco, serão mapeados para colunas na tabela correspondente.

A opção **strategy** dentro da anotação **@GeneratedValue** determina a estratégia a ser usada para gerar os valores da chave primária. No caso de **GenerationType.AUTO**, o provedor JPA (como o Hibernate, por exemplo) escolherá a estratégia mais apropriada com base no banco de dados subjacente.

No entanto, é importante verificar a documentação do provedor JPA específico que você está utilizando e garantir que o banco de dados também suporte a funcionalidade de auto incremento para assegurar esse comportamento. Caso contrário, você pode precisar usar uma estratégia diferente, como **GenerationType.IDENTITY** ou **GenerationType.SEQUENCE**, dependendo das opções suportadas pelo seu banco de dados.

O Spring Data JPA, em conjunto com o Hibernate (uma das implementações do JPA), usa a informação fornecida pelas anotações **@Entity** e outras anotações relacionadas para criar, modificar e consultar dados no banco de dados, permitindo que os desenvolvedores interajam com o banco de dados usando objetos Java e consultas JPQL (Java Persistence Query Language).

A anotação **@GetMapping** é uma das anotações fornecidas pelo Spring Framework para mapear métodos em um controlador (controller) Spring para manipular solicitações HTTP GET. Ela é uma maneira concisa e legível de associar um método específico a uma rota ou URL específica que usa o método HTTP GET.

Mapeamento de Requisição HTTP GET: A anotação **@GetMapping** é usada para mapear um método para manipular solicitações HTTP GET. Isso significa que o método será executado quando o servidor receber uma requisição GET para a URL mapeada.

Definição da Rota: Ao usar **@GetMapping**, você especifica a rota que o método deve manipular. Essa rota pode incluir variáveis de caminho (path variables) e valores de parâmetros de consulta (query parameters).

Parâmetros Opcionais: A anotação **@GetMapping** permite que você defina parâmetros como opcionais adicionando valores padrão a eles. Se um valor não for fornecido na solicitação, o parâmetro será definido com o valor padrão (“ / ”).

A anotação **@PostMapping** é outra anotação do Spring Framework usada para mapear métodos em um controlador (controller) Spring para manipular solicitações HTTP POST. Ela é usada quando desejamos criar ou enviar dados para o servidor, por exemplo, ao enviar um formulário HTML ou enviar dados JSON em uma API.

Principais pontos sobre a anotação **@PostMapping**:

Mapeamento de Requisição HTTP POST: A anotação **@PostMapping** é usada para mapear um método para manipular solicitações HTTP POST. Isso significa que o método será executado quando o servidor receber uma requisição POST para a URL mapeada.

Dados no Corpo da Solicitação: Diferente da anotação **@GetMapping**, em que os parâmetros são passados na URL, ao utilizar **@PostMapping**, os dados são enviados no corpo (body) da solicitação HTTP, geralmente em formato JSON, XML ou formulário.

Envio de Dados: A anotação **@PostMapping** é frequentemente usada em conjunto com um payload (carga útil) no corpo da solicitação, que contém os dados que serão enviados ao servidor.

A anotação **@RequestParam** é usada no Spring Framework para mapear parâmetros de uma solicitação HTTP (geralmente provenientes de uma URL com o método GET ou de um formulário com o método POST) para parâmetros de método em um controlador (controller) Spring. Ela permite que você obtenha valores fornecidos como parâmetros de consulta (query parameters) em uma URL ou dados enviados no corpo (body) de uma solicitação HTTP.

Principais pontos sobre a anotação **@RequestParam**:

Mapeamento de Parâmetros: A anotação @RequestParam é usada para mapear parâmetros de uma solicitação HTTP para parâmetros de método em um controlador Spring.

Obtenção de Valores: Com **@RequestParam**, você pode obter os valores de parâmetros de consulta (query parameters) em uma URL (para solicitações GET) ou dados enviados no corpo (body) de uma solicitação HTTP (para solicitações POST).

Valores Opcionais: Por padrão, os parâmetros anotados com **@RequestParam** são obrigatórios. No entanto, você pode definir um valor padrão usando o atributo **defaultValue**, tornando-os opcionais.

O uso da anotação **@RequestParam** é útil para obter dados dos clientes em uma solicitação HTTP e usá-los para realizar operações no controlador Spring, facilitando a interação com a API ou aplicação web.

A anotação **@PathVariable** é usada no Spring Framework para mapear variáveis de caminho (path variables) presentes em uma URL para parâmetros de método em um controlador (controller) Spring. Ela permite que você capture valores dinâmicos presentes na própria URL e os utilize como parâmetros em seus métodos.

Principais pontos sobre a anotação **@PathVariable**:

Mapeamento de Variáveis de Caminho: A anotação **@PathVariable** é usada para mapear variáveis de caminho presentes na URL para parâmetros de método em um controlador Spring.

Uso em URLs com Variáveis: É comum usar **@PathVariable** quando há valores dinâmicos na URL, como IDs ou nomes de recursos que variam em cada solicitação.

Nome do Parâmetro: Ao usar **@PathVariable**, você precisa fornecer o nome da variável de caminho que deseja mapear para o parâmetro de método correspondente.

Essa anotação é útil para manipular URLs que contêm informações variáveis e capturar esses valores para realizar operações específicas em um controlador Spring. É frequentemente usada em aplicações que usam identificadores únicos (como IDs de banco de dados) na URL para recuperar recursos específicos do servidor.

O atributo required é usado em conjunto com as anotações **@RequestParam** e @PathVariable para especificar se o parâmetro é obrigatório ou opcional na solicitação HTTP.

Quando você define **required = false**, está indicando que o parâmetro não é obrigatório na solicitação. Isso significa que o método poderá ser executado mesmo se o parâmetro não for fornecido.

O **BindingResult** é um objeto usado no Spring Framework para armazenar os resultados da validação de dados que são realizadas em um objeto de comando (command object), geralmente vinculado (bind) a uma solicitação HTTP, como um formulário HTML ou uma requisição JSON.

O **BindingResult** trabalha em conjunto com o objeto de comando (como um @ModelAttribute) e a anotação **@Valid** para executar a validação dos dados recebidos na solicitação. A validação é realizada com base em anotações de validação (como **@NotNull, @Min, @Max, etc.**) colocadas nos campos do objeto de comando.

Quando o Spring executa a validação do objeto de comando, ele verifica se os dados fornecidos na solicitação estão de acordo com as restrições definidas pelas anotações de validação. Se a validação falhar, os erros encontrados são armazenados no objeto **BindingResult**.

Principais pontos sobre **BindingResult**:

Validação de Dados: O **BindingResult** é usado para capturar e armazenar os erros de validação do objeto de comando.

Uso com **@Valid**: É comum combinar **BindingResult** com a anotação **@Valid** para executar a validação automática do objeto de comando.

Tratamento de Erros: O desenvolvedor pode inspecionar o **BindingResult** para recuperar os erros de validação e tomar ações apropriadas, como retornar mensagens de erro ao usuário.

O **BindingResult** é uma ferramenta poderosa para garantir a integridade dos dados recebidos nas solicitações HTTP e fornecer feedback adequado aos usuários quando ocorrerem erros de validação.

Em Spring Framework, o termo "Model" pode referir-se a diferentes conceitos dependendo do contexto em que é utilizado. Os três conceitos principais relacionados ao termo "Model" são:

Modelo de Dados (Data Model):

Neste contexto, o "Model" representa a camada de dados ou a estrutura que define como os dados são organizados e manipulados em um aplicativo. Isso pode incluir classes de entidade (como objetos de domínio) que representam as tabelas do banco de dados, bem como classes de transferência de dados (DTOs) usadas para transportar informações entre diferentes camadas do aplicativo. No Spring Framework, o Spring Data JPA é uma ferramenta comum para simplificar a camada de acesso a dados e mapeamento de objetos para bancos de dados.

Modelo de Domínio (Domain Model):

Neste contexto, o "Model" representa a camada de negócios ou as classes que encapsulam a lógica de negócios do aplicativo. O modelo de domínio contém classes que representam objetos do mundo real e seus comportamentos. Essas classes são independentes do framework e são projetadas para refletir o domínio específico do problema que o aplicativo está resolvendo. O Spring Framework suporta essa abordagem, permitindo que você crie classes de serviço e repositórios para encapsular a lógica de negócios e o acesso a dados.

Modelo MVC (Model-View-Controller):

Neste contexto, o "Model" é uma das três componentes do padrão de projeto Model-View-Controller (MVC). O "Model" representa a camada de negócios ou os dados do aplicativo. No padrão MVC, o Model é responsável por gerenciar os dados e a lógica de negócios. Ele interage com a camada de visualização (View) para apresentar os dados aos usuários e também com a camada de controle (Controller) para processar as ações dos usuários. O Spring Framework é comumente usado para implementar a arquitetura MVC em aplicativos da web, onde os controladores lidam com as solicitações, os modelos gerenciam os dados e as visualizações renderizam a interface do usuário.

O termo "Model" no contexto do Spring Framework pode se referir a:

Modelo de Dados: A camada de acesso e manipulação de dados, geralmente usando classes de entidade e classes de transferência de dados.

Modelo de Domínio: A camada de negócios, contendo classes que representam objetos do mundo real e sua lógica de negócios.

Modelo MVC: A parte do padrão de projeto Model-View-Controller responsável por gerenciar os dados e a lógica de negócios.

A anotação **@NotBlank** é uma anotação de validação fornecida pelo Bean Validation, que é uma especificação que faz parte do Java Persistence API (JPA). Essa anotação é usada para validar campos de texto e garantir que eles não estejam em branco ou consistam apenas em espaços em branco.

Principais pontos sobre a anotação @NotBlank:

Validação de Texto não Vazio: A anotação **@NotBlank** é usada para garantir que um campo de texto não esteja em branco e não consista apenas em espaços em branco.

Trimming: A anotação **@NotBlank** também considera os espaços em branco no início e no final da cadeia de caracteres, e os remove (trims) antes de realizar a validação.

Uso em Campos de Texto: A anotação **@NotBlank** geralmente é aplicada a campos de texto em classes de entidade ou DTOs para garantir que esses campos sejam preenchidos com informações relevantes.

```java
package com.web.minicurso.crudspringthymeleaf.web.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class HomeController {

    // abrir página home
    @GetMapping({"/"})
    public String home() {
        return "home";
    }
}
```

@GetMapping({"/"}): A anotação @GetMapping é usada para mapear o método home() a solicitações HTTP GET. Aqui, a anotação está configurada para mapear a rota raiz **("/")**, indicando que o método responderá a solicitações GET para a URL principal do aplicativo.

public String home(): Este é o método home() que será executado quando a rota raiz for acessada. É importante notar que, se o método não tiver outras anotações ou

configurações específicas, a string retornada será tratada como o nome de uma visualização (view) a ser renderizada. Neste caso, a string "home" provavelmente está relacionada a uma visualização chamada "home.html" ou outro modelo apropriado.

Por exemplo, supondo que você esteja usando o Spring com o Thymeleaf como mecanismo de renderização de visualizações, pode haver um arquivo "home.html" no diretório de visualizações (templates), e esse arquivo será exibido quando a rota raiz do aplicativo for acessada.

O Spring irá mapear a URL **"/"** para o método home(), que retornará a String "home". O mecanismo de renderização (Thymeleaf ou outro) usará essa informação para localizar o template "home.html" e apresentá-lo ao cliente que fez a solicitação.

O arquivo **application.properties** é um arquivo de configuração amplamente usado no desenvolvimento de aplicativos Spring. Nele, você pode configurar várias propriedades e opções do seu aplicativo, permitindo que você personalize o comportamento do Spring Boot de acordo com suas necessidades.

Algumas das configurações comuns que podem ser encontradas no arquivo **application.properties** são:

1. **Configurações do DataSource (Banco de Dados)**:
    - **spring.datasource.url**: URL de conexão com o banco de dados.
    - **spring.datasource.username**: Nome de usuário para autenticação no banco de dados.
    - **spring.datasource.password**: Senha para autenticação no banco de dados.
2. **Configurações de Log**:
    - **logging.level.***: Nível de log para diferentes pacotes ou classes.
3. **Configurações de Internacionalização**:
    - **spring.messages.basename**: Localização dos arquivos de mensagens de internacionalização.
4. **Configurações de Porta do Servidor**:
    - **server.port**: Porta em que o servidor do aplicativo será executado.
5. **Configurações de Caminho do Contexto**:
    - **server.servlet.context-path**: Caminho do contexto da aplicação.
6. **Configurações do Spring Security**:
    - **spring.security.user.name**: Nome do usuário padrão para autenticação em desenvolvimento.
    - **spring.security.user.password**: Senha do usuário padrão para autenticação em desenvolvimento.

7. **Configurações do Cache**:
    - **spring.cache.type**: Tipo de cache a ser usado (por exemplo, "simple" ou "redis").

Essas configurações estão relacionadas ao **Thymeleaf**, que é um mecanismo de template muito utilizado no Spring Framework para renderização de páginas HTML. Vamos explicar o significado de cada uma delas:

- **spring.thymeleaf.cache=false:** Define se o cache do Thymeleaf está habilitado ou desabilitado. Quando definido como "false", o cache é desativado, o que significa que as páginas do template serão reprocessadas a cada solicitação. Essa configuração é útil durante o desenvolvimento, pois permite que você veja alterações imediatamente sem precisar reiniciar o servidor.
- **spring.thymeleaf.mode=HTML:** Define o modo de execução do Thymeleaf. Nesse caso, está configurado como "HTML", o que significa que o Thymeleaf tratará os arquivos de template como páginas HTML padrão.
- **spring.thymeleaf.encoding=UTF-8:** Define a codificação a ser usada ao processar os arquivos de template do Thymeleaf. Neste caso, está definido como "UTF-8", que é uma codificação amplamente usada para suportar caracteres internacionais.
- **spring.thymeleaf.prefix=file:src/main/resources/templates/:** Essa configuração define o prefixo usado para localizar os arquivos de template do Thymeleaf. Neste caso, os templates serão procurados no caminho "src/main/resources/templates/". Isso é comum em projetos Spring Boot, onde o diretório "src/main/resources/" é uma pasta padrão para colocar recursos, incluindo templates Thymeleaf.
- Juntas, essas configurações fornecem uma maneira de personalizar o comportamento do Thymeleaf para que ele renderize corretamente os templates HTML com a codificação desejada e possa ser ajustado conforme necessário para melhorar o fluxo de desenvolvimento. As configurações podem variar dependendo da versão do Spring Boot que você está usando, mas a ideia geral é a mesma

Static resources: Essas configurações estão relacionadas ao gerenciamento de recursos estáticos no Spring Boot. Recursos estáticos, como arquivos CSS, JavaScript, imagens, etc., são arquivos que não são processados pelo servidor, mas são servidos diretamente ao cliente. Essas configurações determinam onde esses recursos estáticos serão buscados e como o cache desses recursos será controlado.

spring.web.resources.static-locations=file:src/main/resources/static/: Essa configuração define o local onde os recursos estáticos serão buscados. Neste caso, os recursos estáticos serão procurados no diretório "src/main/resources/static/". Esse

diretório é uma convenção comum no Spring Boot para armazenar arquivos estáticos que serão servidos diretamente aos clientes.

spring.web.resources.cache.period=0: Essa configuração define o período de cache para os recursos estáticos em segundos. Quando configurado como "0", o cache é desabilitado para os recursos estáticos, o que significa que o navegador do cliente solicitará os recursos novamente a cada nova solicitação, sem armazená-los em cache localmente.

O motivo pelo qual o cache é desabilitado para recursos estáticos (definido como "0") é útil durante o desenvolvimento. Isso permite que você veja as alterações nos arquivos de recursos estáticos imediatamente, sem precisar limpar o cache do navegador.

Lembre-se de que, em ambientes de produção, você pode querer ajustar o período de cache de recursos estáticos para otimizar o desempenho do site e reduzir a quantidade de solicitações ao servidor. No entanto, em ambientes de desenvolvimento, é comum desativar completamente o cache para facilitar a depuração e o desenvolvimento contínuo.

Essas são apenas algumas das configurações possíveis no arquivo **application.properties**. O arquivo é muito flexível e permite personalizar várias partes do seu aplicativo Spring Boot. As configurações específicas que você adiciona ao arquivo dependerão dos requisitos e da estrutura do seu aplicativo.