**SPRING**

* **Inversão de controle com injeção de dependência**

Ao invés de fazer

public class VendaDeProduto {

public void vendeProduto(Produto produto) {

//Todo o código para a venda do produto...

Log log = new Log("Arquivo.txt");

log.grava(produto);

}

}

Fazer:

public class VendaDeProduto {

private Log log;

public void VendaDeProduto(Log logVenda) {

this.log = logVenda;

}

public void vendeProduto(Produto produto) {

//Todo o código para a venda do produto...</span>

log.grava(produto);

}

}

* **Injeção de dependência**

Ao invés de fazer:

<?php

class Notification

{

public function send()

{

// TODO

}

}

class UserCommand

{

public function handle()

{

$notification = new Notification();

$notification->send();

}

}

Fazer:

<?php

class Notification

{

public function send()

{

// TODO

}

}

class UserCommand

{

protected $notification;

public function \_\_construct(Notification $notification)

{

$this->notification = $notification;

}

public function handle()

{

$this->notification->send();

}

}

* **Autowired**

Edit.: para scannear os componentes que implementam a interface (fazer funcionar o Autowiring), tem que adicionar essa linha n oxml:

<context:component-scan base-package=*"com.luv2code.springdemo"*/>

Dispensa definir no applicationContext.xml a implementação da interface que será utilizada. Quando essa notação é utilizada, o Spring busca todas as implementações da interface e aplica a implementação que ele encontrar caso seja só uma, já inventando o valor do objeto criado, sem necessidade de criação do construtor pra injeção. Caso ache mais de uma implementação dessa interface, tem que usar o @Qualifier("nomeDaClasseDesejadaQueImplementaAInterface").

Exemplo sem autowired:

**private** FortuneService fortuneService;

**public** TrackCoach (FortuneService fortuneService) {

**this**.fortuneService = fortuneService;

}

E ainda precisamos definir no applicationContext.xml qual é a classe que implementa a interface FortuneService, dessa forma:

<bean id=*"myCoach"*

class=*"com.luv2code.springdemo.TrackCoach"*>

<constructor-arg ref=*"myFortuneService"*/>

</bean>

Agora, exemplo com autowired:

@Autowired

@Qualifier("randomFortuneService")

**private** FortuneService fortuneService;

Já está usando o @Qualifier porque além da randomFortuneService, temos várias classes que implementam a interface FortuneService como, por exemplo, badFortuneService e happyFortuneService.

Lembrando que a classe que implementa a interface tem que colocar o @Component no inicio pra informar que está implementando, como no exemplo do randomFortuneService:

@Component

**public** **class** RandomFortuneService **implements** FortuneService {

…

}

* **Prototype**

Primeiramente, temos que lembrar que todos os beans criados pelo Spring são Singleton. Logo, se crio duas instancias do mesmo bean, elas apontarão pro mesmo endereço de memória. Para criar duas instâncias do mesmo bean que apontam para endereços diferentes de memória, preciso adicionar a notação @Scope("prototype"). Por default, o Spring já vem com a notação @Scope("singleton").

Exemplo:

@Component

@Scope("singleton")

**public** **class** TennisCoach **implements** Coach{

…

}

* **PostConstruct**

Ao colocar essa notação em um método, ele será executado após o construtor e após a injeção de dependências.

* **PreDestroy**

Ao colocar essa notação em um método, ele será executado antes do bean ser destruído.

**ATENÇÃO:** Pode haver erro ao usar PostConstruct e PreDestroy no Java 9 ou mais novo, por erro de caminho do javax.annotation. Para resolver isso, consultar aula 80 do curso.

**SPRING MVC**

**Como adicionar CSS, JS e imagens nas páginas web usando o Spring?** Consultar aula 117 do curso que informa como fazer isso.

**Binding request param**

Ao invés de utilizar

@RequestMapping("/processFormVersionThree")

**public** String processFormVersionThree(String theName, Model model) {

// read the request parameter from the HTML form

String theName = request.getParameter("studentName");

…

**return** "helloworld";

}

É possível adicionar uma anotação direto nos parâmetros do método, ligando os dados de um campo do formulário que ta chamando esse método a uma nova variável.

@RequestMapping("/processFormVersionThree")

**public** String processFormVersionThree(@RequestParam("studentName") String theName, Model model) {

// notar que agora a variáfvel theName já é setada com o valor // do campo studentName assim que o método é chamado

…

**return** "helloworld";

}

**Parent mapping**

Quando usamos dois @RequestMapping com mesmo valor, resultará em erro. Por isso, temos que implementar o parente mapping, que é dar um pai para cada mapping de forma que mapping de mesmo nome tenha pais diferentes. Esse pai é geralmente dado acima do da definição da classe.

**Request Mapping**

É uma espécie de ID que você da para a classe. Quando você retorna esse id partindo de um método da classe X, ele trabalha de forma parecida com o header do PHP, levando a página Y passada no return do método. Entretanto, o parâmetro que é passado no RequestMapping também pode chamar um controller, que não tem nenhum tipo de View, apenas para processar os dados da página que ta fazendo a requisição (que está dando return Y). Lembrar as regras de ParentMapping quando usando Mapping.

**@ModelAttribute**

Quando passado como parâmetro na definição do método, seta qual é o nome do model que esse método recebe.

**@Valid**

Faz a validação no objeto passado por ModelAttribute

**BindingResult**

Os resultados da validação do @Valid são colocados no BindingResult e conseguimos ver se tem algum erro.

**@NotNull=(message=”is required”)**

Notação feita no model para validação de campo, impedindo que o campo seja nulo e passando a variável message com o valor “is required” para a página que contem o campo que passou o valor erro.

**@Size(min=1)**

Limita o tamanho no campo no model

**@InitBinder**

Remove espaços em branco do início e do final da string passada. Se a string for só espaços em branco, valor dela é alterado pra null.