02

Injeção de dependências

Acoplamento entre classes

Note que nosso `ProdutoController` é responsável por instanciar o `Dao`. Do ponto de vista da orientação a objetos, isso é um problema, já que essa classe está **fortemente acoplada** à classe `ProdutoDao` e à criação dela:

```
@Resource
public class ProdutoController {

    private final ProdutoDao produtos;
    //outros atributos aqui

    public ProdutoController(Result result) {
        this.result = result;
        this.produtos = new ProdutoDao(); //acoplamento
    }

    //métodos aqui
}
```

Injetando dependências

O ideal é que o controlador não seja o responsável por instanciá-lo. O `ProdutoDao`, como o Result, é uma dependência. Vamos também receber o `ProdutoDao` no construtor. Essa é uma das ideias da **injeção de dependências**, ou DI. Vamos melhorar o código fazendo a classe `ProdutoController` receber suas dependências através do construtor.

```
@Resource
public class ProdutoController {

    private final ProdutoDao produtos;
    //outros atributos aqui

    public ProdutoController(Result result, ProdutoDao produtos) {
        this.result = result;
        this.produtos = produtos; //sem dar "new"
    }

    //métodos aqui
}
```

Componentes com VRaptor

Mas quem será o responsável por injetar essas dependências? O próprio VRaptor! Ele observa o construtor dos seus controladores e passa todas as dependências necessárias para ele. Para que isso funcione, informamos ao VRaptor que a classe `ProdutoDao` é injetável. Basta anotá-la como `@Component`:

```
//outros imports
import br.com.caelum.vraptor.ioc.Component;

@Component
```

```
public class ProdutoDao implements RepositorioDeProdutos  {

     //implementação omitida
}
```

Vamos rodar a aplicação para ver que tudo está funcionando.

O uso de um repositório como interface do Dao

É possível melhorar esse modelo ainda mais. Por enquanto o construtor recebe a classe `ProdutoDao` específica e não uma abstração, uma interface. E veja que a classe `ProdutoDao` implementa `RepositorioDeProdutos` . Excelente, podemos fazer com que o construtor de `ProdutoController` receba um `RepositorioDeProdutos` :

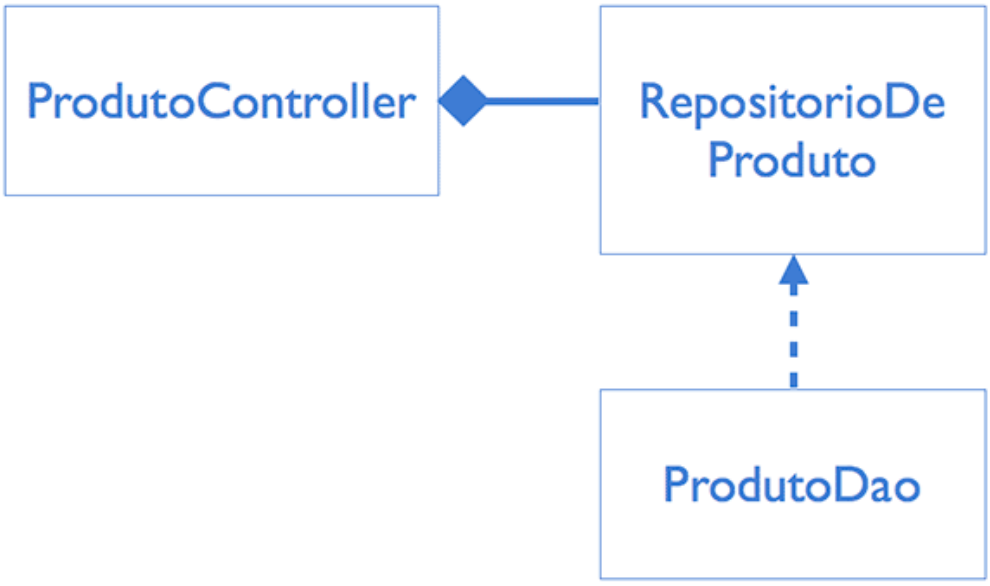
```
@Resource
public class ProdutoController {

    private final Result result;
    private final RepositorioDeProdutos produtos;

    public ProdutoController(Result result, RepositorioDeProdutos produtos) {
        this.result = result;
        this.produtos = produtos;
    }

    //métodos aqui
}
```

Repare que **acoplar com uma interface é muito melhor** do que acoplar com a classe concreta. A interface `RepositorioDeProdutos` não tem nenhuma ligação com o acesso a banco de dados ou qualquer coisa do tipo. Isso permite ao programador escrever um teste automatizado de unidade futuro para esse controlador!



Na prática, o VRaptor saberá que `ProdutoDao` implementa essa interface, e o injetará. Vamos rodar o projeto novamente para testar.

Definição do escopo dos componentes

Ao dizer que o VRaptor pode instanciar a classe `ProdutoDao`, precisamos dizer também qual o **ciclo de vida desse objeto**, ou seja, o tempo em que ele vive. No nosso caso, a classe `ProdutoDao` vive durante uma requisição.

Isso quer dizer que, a cada nova requisição, o VRaptor cria uma nova instância do Dao. Podemos inclusive deixar isso explícito através da notação `@RequestScoped`. O **escopo de requisição é o padrão** do VRaptor.

```
@Component
@RequestScoped
public class ProdutoDao implements RepositorioDeProdutos {
```

Escopo da sessão

Às vezes não queremos que determinado objeto seja instanciado a cada requisição. Um exemplo disso é guardar o objeto que representa o usuário logado. Criar um novo objeto desse a cada requisição não faz sentido. Esse objeto deve, na verdade, viver durante **toda a sessão do usuário**.

Vamos criar a classe `UsuarioLogado` que guarda em uma String apenas o login do usuário logado. Vamos anotá-lo como `@Component`, pois ela é um componente, e como **@SessionScoped** pois ela deve viver na sessão. Vamos também criar getters/setters para esse atributo.

```
@Component
@SessionScoped
public class UsuarioLogado {

    private String login;
    //outros dados sobre o usuário estariam aqui, senha, grupos etc

    //getter e setter
}
```

Exemplo de login

Agora vamos criar um `LoginController` e receber esse `UsuarioLogado` no construtor. Não esqueça de anotar essa classe como `@Resource`. O método `loga()` deve setar um usuário qualquer nesse usuario logado. Vamos passar "caelum" para ele. O valor deveria vir de uma formulário de login junto com a senha de usuário. No método `loga()` deveríamos executar a autenticação, verificar se o usuário realmente existe no sistema e, em caso positivo, setar o valor no objeto `usuarioLogado`:

```
@Resource
public class LoginController {

    private final UsuarioLogado usuarioLogado;

    public LoginController(UsuarioLogado usuarioLogado) {
        this.usuarioLogado = usuarioLogado;
    }

    public void loga() {
        //receber os valores de um formulário de login e verificar se usuario existe
        this.usuarioLogado.setLogin("caelum");
    }
}
```

Agora vamos criar um método `exibe()` que apenas retorna o usuário logado na JSP usando o Result. Recebemos o Result no construtor e incluímos o login do `usuarioLogado`. Veja o código completo:

```
@Resource
public class LoginController {

    private final UsuarioLogado usuarioLogado;
    private final Result result;

    public LoginController(UsuarioLogado usuarioLogado, Result result) {
        this.usuarioLogado = usuarioLogado;
        this.result = result;
    }

    public void loga() {
        this.usuarioLogado.setLogin("caelum");
    }

    public void exibe() {
        result.include("usuario", this.usuarioLogado.getLogin());
    }
}
```

Vamos criar 2 JSPs para o `loga()` e para o `exibe()`. O `loga.jsp` apenas exibe uma mensagem de confirmação. O `exibe` mostra o usuário que está logado. Ambos os arquivos ficam dentro da pasta `WEB-INF/login`:

loga.jsp

```
<html>
  <body>
    Login efetuado com sucesso!
  </body>
</html>
```

exibe.jsp

```
<html>
  <body>
      Usuário logado: ${usuario}
  </body>
</html>
```

Vamos rodar a aplicação e chamar primeiro o `exibe()` através do:

(http://localhost:8080/vraptor-produtos/login/exibe)http://localhost:8080/vraptor-produtos/login/exibe (http://localhost:8080/vraptor-produtos/login/exibe)

Repare que não há usuário logado. Agora chamamos o `loga()`:

(http://localhost:8080/vraptor-produtos/login/loga)http://localhost:8080/vraptor-produtos/login/loga (http://localhost:8080/vraptor-produtos/login/loga)

Sucesso. Agora, mais uma vez, acessamos o `exibe()`:

(http://localhost:8080/vraptor-produtos/login/exibe)http://localhost:8080/vraptor-produtos/login/exibe (http://localhost:8080/vraptor-produtos/login/exibe)

Veja que apareceu "caelum"! Perceba que o valor ficou armazenado mesmo entre request diferentes, pois esse objeto vive no escopo de Sessão. Para fazer isso acontecer, bastou anotarmos o `@SessionScoped`.