02

Injeção de Dependências com CDI

Injeção de Dependências com CDI

Esse conceito de pedir um objeto que precisamos no lugar de criá-lo, que utilizamos com a classe `Result` no capitulo anterior, é um dos principios da Injeção de Dependências! A idéia é que no lugar de você sair criando todos os objetos que a classe precisa (como estamos fazendo com o nosso `DAO` e `EntityManager` do JPA), esses objetos devem ser fornecidos a você.

Repare que isso diminui muito o acoplamento de nossas classes! A minha classe `ProdutoController` não precisa conhecer o processo de criação e dependencias da classe `Result` para poder usá-la. Mas repare que ainda estamos com um nível de acoplamento muito alto nessa classe, já que ela está com toda a responsabilidade de criar o `ProdutoDao`, tendo que criar um `EntityManager` para que isso funcione. Nosso código está muito amarrado, e pelo ponto de vista da orientação a objetos isso é um problema.

Não se preocupe, já vamos aplicar esse mesmo conceito de Injeção de Dependências (ou DI) na construção desses outros objetos. Mas primeiramente, vamos entender um pouco mais como isso é possivel, e conhecer algumas de suas regras.

Para injetar essa dependência utilizamos a anotação `@Inject`. Essa anotação vem do CDI, que é a especificação do java que diz respeito a injeção de dependências. O VRaptor 4 foi implementado tendo o CDI como container interno! Se você quiser conhecer um pouco mais de CDI, você pode se interessar por [esse post do blog da caelum (http://blog.caelum.com.br/use-cdi-no-seu-proximo-projeto-java/)](http://blog.caelum.com.br/use-cdi-no-seu-proximo-projeto-java/).

Para que o CDI consiga gerenciar as nossas classes, só precisamos adicionar o arquivo `beans.xml` dentro do diretório `/META-INF/` de nosso projeto. Para facilitar, já deixamos esse arquivo adicionado para você.

Uma regra importante para utilizarmos essa injeção pelo construtor é que você precisa adicionar um `construtor default` em sua classe! Nosso controller vai ficar assim:

```
@Controller
public class ProdutoController {

    private final Result result;

    @Inject
    public ProdutoController(Result result) {
        this.result = result;
    }

    @Deprecated
    ProdutoController() {
        this(null); // apenas para uso do CDI!
    }

    // outros métodos da classe
}
```

Para evitar que alguém chame esse construtor por engano em testes e até mesmo para deixar explicito que é apenas para uso do CDI, é uma prática comum adicionar um `@Deprecated` nesse construtor vazio, e deixar sua visibilizade `default`.

Uma outra forma de fazer a injeção de dependências é diretamente pelo atributo da classe:

```
@Controller
public class ProdutoController {

    @Inject private Result result;

    // outros métodos da classe
}
```

É uma forma mais enxuta, porém consideramos uma boa prática a injeção pelo construtor. Isso facilita a testabilidade da classe. É claro, você ainda pode testar suas classes que utilizam injeção direta pelo atributo utilizando *reflection*, criando um *setter* para os atributos ou qualquer outro tipo de estratégia para inserir um valor de teste naquele atributo, porém passar a informação pelo construtor é uma saida bem mais simples e elegante.

Diminuindo acoplamento de nossas classes com DI

Vamos aplicar o mesmo conceito de DI para diminuir o acoplamento de nosso `ProdutoController`.

Primeiro, vamos apagar toda a logica de criação do `ProdutoDao` e `EntityManager`, esse código não pertence mais ao nosso `Controller`! Agora, vamos pedir para que o VRaptor (com CDI) nos entregue esse nosso `DAO` pronto! Basta adicionar o parâmetro `ProdutoDao` em nosso construtor, assim como fizemos com o `Result`. No final, nosso controller deve ficar assim:

```
@Controller
public class ProdutoController {

    private final Result result;
    private final ProdutoDao dao;

    @Inject
    public ProdutoController(Result result, ProdutoDao dao) {
        this.result = result;
        this.dao = dao;
    }

    @Deprecated
    ProdutoController() {
        this(null, null); // apenas para uso do CDI!
    }

    @Get("/")
    public void index() {
        System.out.println("primeiro projeto com VRaptor 4!");
    }

    @Get
    public List<Produto> lista() {
        return dao.lista();
```

```
    }

    @Get
    public void formulario() {
    }

    @Post
    public void adiciona(Produto produto) {
        dao.adiciona(produto);
        result.forwardTo(this).lista();
    }

    // outros métodos que eu tenha implementado
}
```

Muito mais simples, não acha? Vamos modificar agora o `ProdutoDao`, para que tudo funcione.

Como precisamos de uma transação antes de fazer todas as operações que precisam de transação em nosso `DAO`, por enquanto vamos abrir e `commit` ar a transação em cada um dos métodos que for necessário (por enquanto o `adiciona` e `remove`). Essa não é a forma ideal de fazer isso, claro, mas logo vamos modificar isso!

```
public class ProdutoDao {

    private final EntityManager em;

    @Inject
    public ProdutoDao(EntityManager em) {
        this.em = em;
    }

    @Deprecated
    ProdutoDao() {
        this(null); // para uso do CDI
    }

    public void adiciona(Produto produto) {
        em.getTransaction().begin();
        em.persist(produto);
        em.getTransaction().commit();
    }

    public void remove(Produto produto) {
        em.getTransaction().begin();
        em.remove(produto);
        em.getTransaction().commit();
    }

    public void busca(Produto produto) {
        em.find(Produto.class, produto.getId());
    }

    @SuppressWarnings("unchecked")
    public List<Produto> lista() {
        return em.createQuery("select p from Produto p").getResultList();
```

```
    }
}
```

Mas repare que para criar um `ProdutoDao`, precisamos passar o `EntityManager` como parâmetro injetado e adicionar um construtor default. Só que agora, quem vai criar o `ProdutoDao` é o CDI! Como ele vai saber criar o `EntityManager` do JPA antes de criar o `ProdutoDao`? Nós sabemos que para criar esse objeto basta chamar a classe `JPAUtil` e usar seu método `criaEntityManager()`. O CDI não.

Produção customizada de classes

Precisamos ensinar o CDI a criar um `EntityManager`! Esse processo é muito simples, basta eu criar uma classe comum, que será nossa `Produtora`, vamos chamá-la de `EntityManagerProducer` e criá-la no pacote `br.com.caelum.vraptor.producers`.

Agora, tudo que precisamos fazer é criar um método que retorna o `EntityManager` criado. Para o CDI saber que precisa chamar esse método para conseguir criar um `EntityManager`, basta adicionar a anotação `@Produces`:

```
package br.com.caelum.vraptor.producers;

public class EntityManagerProducer {

    @Produces @RequestScoped
    public EntityManager criaEntityManager(){
        return JPAUtil.criaEntityManager();
    }
}
```

Atenção: Não crie a classe produtora dentro do pacote `br.com.caelum.vraptor.util`, esse pacote é reservado e não é gerenciado pelo CDI.

Todas as nossas classes são managed beans

Repare que não precisamos usar nenhuma anotação especial para conseguir injetar nossa classe! Como isso foi possível? Como usamos o `CDI` para fazer a injeção de dependências, por padrão ele vai gerenciar todas as classes de nosso projeto! Ou seja, todas as nossas classes são passíveis de injeção de dependências (ou `Managed Beans`).

É claro, se eu não quiser essa opção, posso modificar meu arquivo `beans.xml` para que ele não gerencie nenhuma, ou gerencie apenas as classes com anotações do CDI.

Tempo de vida dos componentes gerenciados

Mas quantas vezes o `CDI` vai criar essas dependências? Quanto tempo essas classes criadas vão existir em memória? Você quem decide ao definir um escopo para sua classe! Por exemplo, se quisermos criar um `ProdutoDao` a cada requisição, basta adicionar a anotação `@RequestScoped` na classe.

```
@RequestScoped
public class ProdutoDao {
```

```
    // código omitido
}
```

Assim, se duas pessoas pedirem injetado o `ProdutoDao` em uma mesma requisição, a mesma instancia será passada. Outros escopos comuns são `@SessionScoped` e `@ApplicationScoped`.

Se você não definir nenhum escopo, por padrão sua classe vai ter o escopo `@Dependent`. Isso significa que a sua classe vai ser instanciada a mesma quantidade de vezes que a classe que a pediu injetada for.