Use CDI no seu próximo projeto Java

Postado dia 23/05/2012 por Sérgio Lopes em Inovação, Java 71

CDI é a especificação do Java EE 6 que cuida da parte de injeção de dependências. E, além de ser oficial e estar incluída em todos os servidores de aplicação, é tão **boa e produtiva** que já tem gente questionando o papel do Spring nos dias de hoje.

O CDI se encaixa muito bem em todo tipo de projeto Java. Se você usa JSF2, usar CDI é natural, como mostramos no nosso curso FJ-26. Mas mesmo para aplicações Web simples, com apenas Servlets, o CDI é um grande ganho.

Habilitando CDI

Habilitar o CDI no projeto é muito simples. Se você já está usando um servidor Java EE 6, basta criar um arquivo vazio chamado `beans.xml` na pasta `META-INF` do seu projeto (ou `WEB-INF` num projeto web). Esse é um simples arquivo de marcação e apenas sua presença já faz com que o servidor habilite o suporte a CDI e escaneie suas classes automaticamente.

Se você estiver usando Tomcat, Jetty ou outro servidor antes do Java EE 6, ainda é possível habilitar o CDI copiando e configurando o JAR de alguma de suas implementações. Vamos usar o Weld, a implementação de referência e a mais usada (já embutida no JBoss e no Glassfish).

No caso do Tomcat, os passos de configuração são:

- Baixar a última versão do Weld no site dele
- Copiar o **weld-servlet.jar** pra pasta WEB-INF/lib
- Criar um arquivo META-INF/context.xml
- Acrescentar as configurações do Weld no web.xml

Feito isso, basta criar o tal arquivo **META-INF/beans.xml** vazio no projeto.

Tudo são beans

Com o CDI habilitado, praticamente todas as classes do projeto são consideradas*managed beans* e, portanto, passíveis de injeção e de serem injetadas. Podemos usar com CDI toda classe pública com um construtor público padrão ou algum que receba parâmetros e esteja anotado com `@Inject`.

Aliás, essa anotação `@Inject` é a base de todo o CDI: é ela quem permite a injeção de dependências e devemos usá-la nos pontos que queremos injeção automática, sejam construtores, setters ou atributos privados.

Essa anotação, embora usada com CDI, faz parte na verdade de uma outra especificação, a JSR330. Essa é uma especificação bastante simples, com apenas algumas anotações relacionadas à injeção de dependências no pacote `javax.inject`. A ideia é ter um conjunto básico de recursos para injeção a serem suportados em todos os frameworks do tipo – além do CDI, Spring, Google Guice, PicoContainer e outros frameworks suportam essas anotações.

Primeira injeção

Imagine que temos uma classe `ProdutoDAO` que cuida de persistir objetos `Produto`:

```
public class ProdutoDAO {
   public void adiciona(Produto produto) {
      // ... implementação
   }
}
```

Essa classe simples, sem nenhuma configuração particular, pode ter objetos injetados em qualquer outro ponto da aplicação. Basta usar a anotação `@Inject`:

```
public class ProdutoController {
    @Inject private ProdutoDAO dao;

    public String inserir() {
        Produto p = // ...
        dao.adiciona(p);
        return "Produto adicionado!";
    }
}
```

Poderíamos também ter feito a própria classe `ProdutoDAO` receber um `EntityManager` da JPA para trabalhar com a persistência:

```
public class ProdutoDAO {
   @Inject private EntityManager manager;

   public void adiciona(Produto produto) {
      manager.persist(produto);
   }
}
```

Produção de objetos

Sempre que encontrar um ponto de injeção, o CDI vai tentar instanciar a classe sendo referenciada. No caso do DAO anterior, ao criá-lo, o CDI vai tentar dar `new` em`EntityManager`.

Mas `EntityManager`, assim como vários outros objetos, não pode ser criado tão facilmente assim. A criação de um `EntityManager` exige a chamada do método`createEntityManager` no `EntityManagerFactory` correspondente. É um objeto com um processo de fabricação particular.

Nos Design Patterns, sempre que encontramos esse tipo de cenário, criamos uma**Factory** para aquele objeto. É um simples método que encapsula a criação não-trivial do objeto em questão. Com CDI é a mesma coisa.

Vamos criar um método em uma classe qualquer do sistema que cuide da produção de objetos `EntityManager`. E podemos indicar ao CDI que queremos que ele use esse método sempre que alguém pedir a injeção de um `EntityManager`. Fazemos isso com a anotação`@Produces`:

```
public class ProdutorEntityManager {
   private static EntityManagerFactory factory = Persistence.createEntiyManagerF

   @Produces
   public EntityManager criaEntityManager() {
      return factory.createEntityManager();
   }
}
```

Agora, todos os pontos de injeção que precisarem de `EntityManager` irão invocar essa fábrica anotada com `@Produces`.

Escopos

Mas quantos EntityManagers vamos criar com o código acima? Por padrão, toda dependência no CDI possui um escopo chamado **Dependant**. Isso é: será instanciada tantas vezes quanto quem estiver chamando for. Certamente não é o que queremos com nosso `EntityManager`.

É usual criar um `EntityManager` por requisição em uma aplicação Web. Podemos indicar isso apenas anotando o método produtor com `@RequestScoped`:

```
public class ProdutorEntityManager {

   @Produces @RequestScoped
   public EntityManager criaEntityManager() {
      // ...
   }
}
```

Agora, um novo `EntityManager` será criado associado a um request. Se mais de uma classe pedir um `EntityManager` durante o mesmo request, a mesma instância será passada, evitando desperdício de recursos. Outros escopos comuns são `@SessionScopede` `@ApplicationScoped`.

Mais: como definimos o escopo da dependência, temos agora uma visão clara de seu ciclo de vida. O objeto durará apenas um request, e será descartado quando o request acabar.

O CDI nos deixa, inclusive, executar algo na fase de descarte da dependência, ao final da requsição. No nossa caso, será muito útil para chamar o `close` no `EntityManager`. Basta criar um método que receba a dependência a ser descartada e anotar o parâmetro com`@Disposes`:

```
public class ProdutorEntityManager {

   @Produces @RequestScoped
   public EntityManager criaEntityManager() {
      // ...
   }

   public void finaliza(@Disposes EntityManager manager) {
      manager.close();
   }
}
```

O método `finaliza` será chamado automaticamente pelo CDI ao final do request e fará o fechamento do `EntityManager`.

Use CDI no seu próximo projeto

O CDI é extremamente completo e poderoso. Esse artigo mostrou apenas o básico e o início do trabalho. Mas seu container de injeção typesafe com anotações simples e produtivas tem muitos outros recursos. Mostrei várias funcionalidades do CDI em umapalestra no JavaOne Brasil que você pode acompanhar a seguir:

Aqui mesmo no blog já mostramos como fazer a produção de dependências de tipos genéricos. Além disso, há recursos como qualifiers, interceptadores, decorators, eventos e muito mais. Mostramos várias dessas funcionalidades no curso FJ-26 da Caelum, inclusive com o **JBoss Seam 3**, uma ótima companhia para o CDI.