04

Injeção de dependências com CDI

Começando daqui? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/jsf-cdi/stages/capitulo-2.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Gerenciando DAO e EntityManager com CDI

O CDI já está tomando conta dos nosso *beans*, que eram responsabilidade do JSF, mas a nossa motivação para usar o CDI era na verdade a **camada de persistência**. Mostramos no vídeo anterior que há uma acoplamento forte entre a classe `DAO` e `JPAUtil`, e os *beans* com o `DAO`. Chegou o momento de arrumar a casa e fazer que o CDI tome conta do `DAO` e do `EntityManager`!

A primeira coisa que devemos fazer é mexer na classe `DAO`. Repare que em **todos** os métodos nós damos `new` em `JPAUtil`, o que mostra o acoplamento da nossa aplicação. Vamos remover **todas as linhas** que dão `new` nessa classe. O acoplamento não existe mais, mas a classe `DAO` está cheia de erros, pois cada método depende do `EntityManager` e ele não existe mais!

Como o `EntityManager` é dependência de todos os métodos, dizemos que ele é uma **dependência da classe**. E o que fazemos com essa dependência? Como a classe precisa dela, vamos adicioná-la ao seu construtor;

```
public class DAO<T> {

    private final Class<T> classe;
    private EntityManager em;

    public DAO(EntityManager em, Class<T> classe) {
        this.em = em;
        this.classe = classe;
    }

    // métodos omitidos
}
```

Mas agora todas os *beans* que utilizam a classe `DAO` estão com erro, vamos resolvê-los.

Vamos começar com a classe `AutorBean`, repare que muitos métodos dão `new` no `DAO`, isso significa que o `DAO` será uma dependência da classe! Criamos o atributo, colocamos-o no construtor e substituímos os `new` s (`new DAO<Autor>(Autor.class)`) da classe pelo novo atributo (`this.dao`):

```
public class AutorBean implements Serializable {

    private DAO<Autor> dao;

    public AutorBean (DAO<Autor> dao) {
        this.dao = dao;
    }

    // restante do código
}
```

Só que quem agora administra os *beans* é o CDI, mas o CDI precisa de um construtor padrão, e nós fizemos um construtor que recebe o `DAO` como dependência, aí o CDI não funcionará! A solução para isso é nós pedirmos para o CDI resolver esse `DAO` para a gente.

Quem cria o *bean* quando a aplicação sobe, é o CDI. A ideia agora é utilizar o mesmo pensamento para o `DAO`, quem vai criá-lo é o CDI! Mas como fazer isso?

O CDI vai **injetar** o `DAO`! Injetar significa passar uma instância pronta para usar. Para isso, vamos utilizar a anotação `@Inject` no atributo, quando o CDI vê essa anotação, ele saberá que o *bean* "quer" ou "precisa" de um `DAO` e consequentemente criará uma instância dele e a disponibilizará para nós:

```
public class AutorBean implements Serializable {

    @Inject
    private DAO<Autor> dao;

    // restante do código, agora sem o construtor
}
```

O CDI inverte o controle, inverter significa **resolver a dependência e injetá-la**. Não é mais a classe `AutorBean` que dá um `new` no `DAO`. O CDI está no controle, busca e injeta a dependência.

Essa forma de inversão de controle é chamado de **injeção de dependências**.

Mas lembra que o CDI precisa de um construtor padrão? Pois é, a classe `DAO` não tem um! Por isso vamos mostrar como o CDI lida com o `DAO` criando um DAO específico. O `DAO` específico significa que teremos um `DAO` para cada entidade do modelo, ou seja a classe `Autor` terá `AutorDao` e a `Livro`, um `LivroDao`:

```
public class AutorBean implements Serializable {

    @Inject
    private AutorDao dao;

    // restante do código, agora sem o construtor
}
```

Então vamos criar a classe `AutorDao`, com os métodos que já conhecemos (use o Eclipse para ajudá-lo nessa tarefa):

```
public class AutorDao {

    public Autor buscaPorId(Integer autorId) {
        return null;
    }

    public void adiciona(Autor autor) {
    }

    public void atualiza(Autor autor) {
    }
```

```
    public void remove(Autor autor) {
    }

    public List<Autor> listaTodos() {
        return null;
    }
}
```

Agora precisamos implementar esses métodos. Todos os métodos utilizarão o `EntityManager`, então ele será uma dependência. O que fazemos com as dependências? Injetamos!

```
public class AutorDao {

    @Inject
    EntityManager em;

    // restante do código
}
```

Com isso podemos agora finalmente implementar os métodos. Mas reparem uma coisa, esses métodos já foram implementados no `DAO` genérico, certo? Então porque não utilizá-los? Para utilizá-los, vamos criar um método que cria uma instância do `DAO` genérico para nós, e a guardará num atributo `dao`. Esse método se chamará `init`:

```
public class AutorDao {

    @Inject
    EntityManager em;

    private DAO<Autor> dao;

    void init() {
        this.dao = new DAO<Autor>(em, Autor.class);
    }

    // restante do código
}
```

Pronto, já temos o `EntityManager` e os métodos disponíveis para nós, agora basta chamar o método do `DAO` genérico em cada método correspondente da classe `AutorDao`:

```
public class AutorDao {

    @Inject
    EntityManager em;

    private DAO<Autor> dao;

    void init() {
        this.dao = new DAO<Autor>(this.em, Autor.class);
    }

    public Autor buscaPorId(Integer autorId) {
```

```
        return this.dao.buscaPorId(autorId);
    }

    public void adiciona(Autor autor) {
        this.dao.adiciona(autor);
    }

    public void atualiza(Autor autor) {
        this.dao.atualiza(autor);
    }

    public void remove(Autor autor) {
        this.dao.remove(autor);
    }

    public List<Autor> listaTodos() {
        return this.dao.listaTodos();
    }

}
```

Então o CDI tomará conta do `AutorDao`, porque vamos injetá-lo o `AutorBean`. O CDI criará o `AutorBean` e verá que ele precisa de um `AutorDao`, por isso ele também irá criá-lo. No `AutorDao` o CDI verá que ele precisa de um `EntityManager` e irá instanciá-lo. Agora só precisamos que o CDI chame o método `init` automaticamente quando instancia a classe `AutorDao`, para isso vamos anotar o método com `@PostConstruct`. Com isso o CDI chamará esse método assim que inicializar o `AutorDao`, inicializando também o `DAO` genérico:

```
public class AutorDao {

    @Inject
    EntityManager em;

    private DAO<Autor> dao;

    @PostConstruct
    void init() {
        this.dao = new DAO<Autor>(this.em, Autor.class);
    }

    // restante dos métodos
}
```

Estamos quase lá! Voltando na classe `AutorBean`, vemos que o CDI injeta o `AutorDao` nela, isso significa que ele executa um simples `new AutorDao()`. Já na classe `AutorDao`, o CDI também injeta um `EntityManager`, ou seja, `new EntityManager` ... Mas o `EntityManager` é uma interface, logo não podemos instanciá-lo!

Então precisamos "ensinar" o CDI a instanciar o `EntityManager`, mas como?

Ensinando o CDI a criar um EntityManager

Vamos dar uma olhada na classe `JPAUtil`, que é responsável por levantar o JPA:

```
public class JPAUtil {

    private static EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("livraria");

    public EntityManager getEntityManager() {
        return emf.createEntityManager();
    }

    public void close(EntityManager em) {
        em.close();
    }
}
```

Queremos que o CDI tome conta desse `EntityManager`. Então vamos "dizer" para o CDI que o método `getEntityManager` produz um `EntityManager`! Vamos devagar, o método `getEntityManager` devolve um `EntityManager` novo, então sabe criá-lo, certo? Olhando para os padrões de projeto, podemos dizer que o método é uma fábrica, aplicando o padrão *factory method*. Só que o CDI chama esses métodos de fábrica de ***Producer***. É apenas um outro nome para algo muito comum.

Para deixar claro que o método sabe criar um `EntityManager`, devemos usar a anotação `@Produces` em cima do método:

```
@Produces
public EntityManager getEntityManager() {
    return emf.createEntityManager();
}
```

Com isso, o CDI conhece o método e sabe que ele devolve um `EntityManager`. Mas ele terá uma dúvida, o CDI desejará saber quantas vezes queremos criar um `EntityManager` dentro da aplicação. Iremos produzir um um novo `EntityManager` a cada requisição, para dizer isso ao CDI basta adicionar a anotação `@RequestScoped`:

```
@Produces
@RequestScoped //javax.enterprise.context.RequestScoped;
public EntityManager getEntityManager() {
    return emf.createEntityManager();
}
```

Produzimos um `EntityManager` a cada requisição, mas ainda temos um último problema. Veja na classe `DAO` que **todos** os métodos fecham o `EntityManager`! Ou seja, se removermos um autor, não conseguiremos gravar um novo na mesma requisição, o que não pode acontecer! Só devemos fechar o `EntityManager` depois da requisição. Então vamos **apagar todas linhas que fecham o `EntityManager` (`em.close()`) na classe `DAO`**.

Agora basta avisarmos ao CDI como fechar o `EntityManager`. Já temos um método `close` em `JPAUtil`. E para chamar um método quando a requisição acaba, o CDI possui uma anotação `@Disposes`:

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

Perfeito, o `JPAUtil` também foi "CDI-ficado"!

Por último, como o `AutorBean` sobrevive por mais de uma requisição, ele e seus atributos precisam implementar a interface `Serializable`. Por isso faça as classes `AutorDao` e `DAO` implementar essa interface.

Mas ainda falta fazer a mesma mudança no `LivroBean` e criar o `LivroDao`. Isso vai ficar para o exercício! Vamos começar?

O que aprendemos?

- Gerenciar qualquer classe com CDI;
- Criar um *Producer* com CDI para inicializar o `EntityManager`;
- Injetar a dependências com CDI usando `@Inject`.