02

# Introdução ao CDI

### Transcrição

Agora que já temos nosso projeto funcionando, vamos entender o que é o CDI e como ele pode nos ajudar.

O CDI é uma especificação do Java EE de contexto e injeção de dependências (**C**ontexts and **D**ependency **I**njection). O CDI gerencia as nossas dependências e as injeta quando são necessárias em algum ponto do nosso código.

Agora vamos ver por que precisamos ter essa injeção de dependências no projeto. Vamos tomar como exemplo a classe DAO e dar uma olhada no método `adicona()`.

Se observarmos o método `adiciona()` do ponto de vista de suas responsabilidades, veremos que o método faz muitas coisas. A primeira linha de código ria um `EntityManager` e, em seguida, é aberta uma transação. Depois o objeto é salvo de fato no banco chamando o método `persist()`. Em seguida, a transação é fechada para depois o fechar o `EntityManager` com a chamada do `close()`.

```
public void adiciona(T t) {

    // consegue a entity manager
    EntityManager em = new JPAUtil().getEntityManager();

    // abre transacao
    em.getTransaction().begin();

    // persiste o objeto
    em.persist(t);

    // commita a transacao
    em.getTransaction().commit();

    // fecha a entity manager
    em.close();
}
```

Qual é o problema de um método possuir várias responsabilidades dentro dele? O problema é que não temos um método coeso. Ou seja, o fato de que possua várias responsabilidades faz com que ele conheça várias classes. No caso do método `adiciona()`, ele conhece a classe `JPAUtil`, e dentro dentro do `EntityManager` - que deveria conhecer apenas o método `persist()`, já que a intenção deste é persistir um dado -, ele conhece métodos que gerenciam transações, por exemplo.

Por que não é legal conhecer essas classes e esses vários métodos? Vamos utilizar como exemplo o método `getEntityManager` da classe `JPAUtil`.

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

Imagine que criamos uma funcionalidade na qual o método `EntityManager` agora recebe uma String que irá no dizer para qual banco ele irá gerar o `EntityManager`. Vamos adicionar um novo parâmetro no método, chamado `db` e é do tipo

`String`. Dentro do método teríamos que implementar uma lógica em que dada a `String`, ele criaria o `EntityManager` para um banco específico. No nosso caso basta apenas que o método receba a `String`.

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

A partir do momento que passamos a receber uma `String`, o DAO irá parar de compilar. O método `getEntityManager()` espera receber uma `String` agora, mas não antes, e por esse motivo não havíamos fornecido qualquer argumento. Por isso é que temos um problema que o método `adiciona()` (e os outros métodos do DAO) conheçam detalhes demais da classe `JPAUtil`.

DAO.java

```
public void adiciona(T t) {

    // consegue a entity manager
    EntityManager em = new JPAUtil().getEntityManager();

    // abre transacao
    em.getTransaction().begin();

    // persiste o objeto
    em.persist(t);

    // commita a transacao
    em.getTransaction().commit();

    // fecha a entity manager
    em.close();
}

public void remove(T t) {
    EntityManager em = new JPAUtil().getEntityManager();
    em.getTransaction().begin();

    em.remove(em.merge(t));

    em.getTransaction().commit();
    em.close();
}

public void atualiza(T t) {
    EntityManager em = new JPAUtil().getEntityManager();
    em.getTransaction().begin();

    em.merge(t);

    em.getTransaction().commit();
    em.close();
}

public List<T> listaTodos() {
    EntityManager em = new JPAUtil().getEntityManager();
    CriteriaQuery<T> query = em.getCriteriaBuilder().createQuery(classe);
    query.select(query.from(classe));

    List<T> lista = em.createQuery(query).getResultList();

    em.close();
    return lista;
}
```

Esse é um problema de acoplamento: estamos muito acoplados à classe `JPAUtil`, e qualquer alteração que se faça nela, refletirá na classe DAO. Será que apenas a classe `DAO` parou de funcionar? Se olharmos no nosso projeto, temos a classe `UsuarioDao`, que também não funciona mais por ter a seguinte linha de código.

```
EntityManager em = new JPAUtil().getEntityManager();
```

A classe `UsuarioDAO` conhece detalhes da classe `JPAUtil` e qualquer alteração na assinatura dos métodos ou construtor dessa classe irá refletir também na classe `UsuarioDao`.

O que fazer para resolver esse problema? Uma das formas que temos é simples: quando paramos para pensar na responsabilidade da nossa classe `DAO`, ela não precisa criar um `EntityManager`, mas sim do objeto pronto para que ela possa utilizar. Então, se precisamos de um objeto, podemos pedir o objeto pronto por meio do construtor.

Vamos remover a criação do `EntityManager` do método adiciona:

```
public void adiciona(T t) {

    // criação do EntityManager removida
```

```
        // abre transacao
        em.getTransaction().begin();

        // restante do código
    }
```

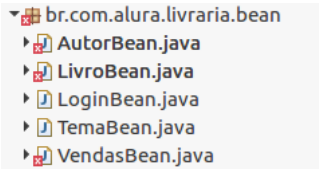
Agora vamos adicionar um parâmetro a mais no nosso construtor que será justamente o `EntityManager`. Além disso vamos atribuir o valor da variável para um atributo da classe. Dessa forma o método `adiciona()` agora irá utilizar esse atributo (`em`):

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

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

Após essa mudança, o método `adiciona()` já não precisa mais conhecer detalhes da classe `JPAUtil`. Ainda temos alguns problemas, pois ainda sabemos como criar uma transação e *commitar* a transação, por exemplo. Mas agora não precisamos mais conhecer a classe `JPAUtil`, que cria esse `EntityManager`. Apenas queremos o objeto pronto para utilizarmos.

Porém quando salvamos o arquivo `DAO.java` com essas alterações, quebramos outras classes. Se olharmos nossos *beans*, quebramos o `AutorBean`, o `LivroBeans`, e o `VendasBean`. E por quê? Mais uma vez temos problemas de acoplamento.




Se verificarmos a classe `AutorBean`, na seguinte linha de código no método `carregarAutorPelaId`:

```
    this.autor = new DAO<Autor>(Autor.class).buscaPorId(autorId);
```

Está sendo instanciado um objeto da classe `DAO`. Precisamos agora suprir essa dependência que o construtor da classe `DAO` espera receber. Precisamos injetar essa dependência.

```
    this.autor = new DAO<Autor>(Autor.class, manager).buscaPorId(autorId);
```

Isso que fizemos, de receber a dependência de uma classe pelo construtor em vez de criá-las, chama-se **inversão de controle**. Nós invertemos o controle da criação do objeto. A gerencia da criação do objeto não é mais responsabilidade da classe que necessita dele. A classe apenas espera o objeto pronto.

Do outro lado, temos alguém que está injetando a dependência da inversão que fizemos. Como invertemos o controle no `DAO`, no `AutorBean`, por exemplo precisamos passar essa dependência.

É justamente nessa parte que o CDI irá no ajudar. Vamos declarar que precisamos de uma informação e o CDI põe a informação pra gente. Ainda temos problema no `AutorBean`, pois vimos que instanciar a classe não é muito interessante

porque geramos um acoplamento com a nossa classe, já que estamos instanciando.

A classe `DAO` é utilizada em `AutorBean` nos métodos `carregarAutorPelaId()` e `gravar()`. Talvez seja interessante declarar como uma dependência também, já que precisamos da classe `DAO` pronta para executarmos as tarefas.

```
public void carregarAutorPelaId() {
    this.autor = dao.buscaPorId(autorId);
}

public String gravar() {
    System.out.println("Gravando autor " + this.autor.getNome());

    if(this.autor.getId() == null) {
        new dao.adiciona(this.autor);
    } else {
        new dao.atualiza(this.autor);
    }

    this.autor = new Autor();

    return "livro?faces-redirect=true";
}

public void remover(Autor autor) {
    System.out.println("Removendo autor " + autor.getNome());
    dao.remove(autor);
}

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

Seria necessário adicionar um construtor na classe `AutorBean`, bem como adicionar um atributo que irá receber a referência passada no construtor:

```
private DAO<Autor> dao;

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

A classe `AutorBean` agora compila. Então recebemos a dependência do `DAO` e utilizamos em várias partes da classe. E não precisamos criar a dependência. O CDI vai fazer essa ponte, onde declaramos a dependência e ele irá injetar a dependência para nós.

Na próxima aula vamos ver como configurar o CDI no nosso projeto.