03

Gerenciando transações com CDI

Começando daqui? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/jsf-cdi/stages/capitulo-3.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo.

Quem deveria controlar a Transação?

A nossa camada de persistência já está bem melhor graças ao CDI e a injeção de dependências. No entanto, olhando no `DAO` genérico há ainda um código duplicado, estamos gerenciando a transação na mão:

```
public class DAO<T> implements Serializable {

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

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

    public void adiciona(T t) {
        // abre transacao
        em.getTransaction().begin();

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

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

    //restante dos métodos omitido
}
```

Ou seja, o `DAO` está no controle do gerenciamento da transação, mas você já sabe quem deveria gerenciá-la. O CDI existe para inverter o controle, será que ele também pode tomar controle da transação?

Repare que essa questão é um pouco diferente da injeção de dependências. A dependência para gerenciar a transação é o `EntityManager` e ele já está no `DAO`. Queremos apenas que as chamadas de `begin` e `commit` não fiquem no `DAO` e sim sejam centralizadas em uma outra classe.

Centralizar o gerenciamento da transação

Vamos tentar criar essa classe, que se chamará `GerenciadorDeTransacao`, no pacote `br.com.caelum.livraria.tx`, com um método `executaTX`, que executa o `begin` e `commit`:

```
public class GerenciadorDeTransacao implements Serializable {

    public void executaTX() {
```

```
        manager.getTransaction().begin();

        //chama o método do DAO que precisa de TX

        manager.getTransaction().commit();
    }
}
```

Com isso, podemos remover todos os `begin`s e `commit`s da classe `DAO` genérica.

Mas agora criamos dois problemas:

1) Estamos usando o `manager` que não existe nessa classe.

2) Como vamos saber qual método devemos chamar entre begin e commit?

O primeiro problema é fácil de resolver, pois o `manager` é uma dependência. Vamos injetá-la:

```
public class GerenciadorDeTransacao implements Serializable {

    @Inject
    EntityManager manager;

    public void executaTX() {

        manager.getTransaction().begin();

        // chamar os daos que precisam de um TX

        manager.getTransaction().commit();
    }
}
```

Criando a anotação @Transacional

O segundo problema já é muito mais difícil de se resolver, como vamos definir no nosso código os pontos que precisam de uma transação? A ideia então é fazer com que os métodos, de alguma maneira, sinalizem que precisam de uma transação (que precisam da nossa classe `GerenciadorDeTransacao`). Essa sinalização pode ser feita através de uma anotação!

Por exemplo, dentro do `AutorBean`, sabemos que o método `gravar` e `remover` precisam de uma transação. Vamos anotá-los:

```
@Named
@ViewScoped
public class AutorBean implements Serializable {

    @Transacional
    public String gravar() {
        // código omitido
    }
```

```
    @Transacional
    public void remover(Livro livro) {
        // código omitido;
    }
}
```

A anotação `@Transacional` não existe, é uma invenção nossa e por isso devemos criá-la, senão o código nem compila.

```
package br.com.caelum.livraria.tx;

public @interface Transacional {

}
```

Quando criamos uma anotação, deve ficar claro que ela representa uma configuração e não uma implementação. A anotação `@Transacional` existe para configurar no método que é preciso ter uma transação. Nada mais do que isso, quem realmente vai chamar `begin` e `commit` é a nossa classe `GerenciadorDeTransacao`.

Target e Rentention da anotação

Quando definimos uma nova anotação, algumas outras configurações genéricas são necessárias. A JVM deve saber onde esse anotação pode ser utilizada. Existem anotações que podem ser utilizadas em cima da classe (por exemplo `@Named`), existem outras que funcionam em cima do atributo (como `@Inject`). A nossa anotação deve funcionar em cima do método, certo? Então vamos deixar isso explicito:

```
@Target({ ElementType.METHOD })
public @interface Transacional {

}
```

Além disso, devemos definir que a anotação faz parte da execução. Como assim? Talvez um ou outro aluno já viu a anotação `@Override`. Através dessa anotação o compilador do Java sabe que queremos sobrescrever um método e verifica a sintaxe. Ou seja, essa anotação é para o compilador. A nossa anotação é diferente e não há um impacto no compilador, ela deve funcionar na hora de executar (*RUNTIME*):

```
@Target({ ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
public @interface Transacional {

}
```

Ótimo! Do ponto de vista do Java padrão a nossa anotação já está pronta, tanto que nosso código compila perfeitamente. Mas do ponto de vista do nosso objetivo (gerenciar a transação) ainda não resolve.

Associar a anotação com a transação

O que falta é associar a nossa anotação `@Transacional` com a classe `GerenciadorDeTransacao`. É essa a tarefa do CDI, ajudar a criar essa ligação. Uma vez feito, o CDI sabe que, ao encontrar a anotação `@Transacional`, deve chamar o

método `executaTX` do `GerenciadorDeTransacao`. Então vamos lá!

Na classe `GerenciadorDeTransacao`, vamos usar a anotação também:

```
@Transacional
public class GerenciadorDeTransacao implements Serializable {
```

Para nossa surpresa, o código não compila. O problema é que definimos na anotação que ela só pode ser utilizada em cima de um método! Vamos corrigir isso:

```
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface Transacional {

}
```

O `ElementType.TYPE` significa a anotação também está valida em cima da classe.

O código voltou a compilar, fizemos a associação da anotação `@Transacional` com classe `GerenciadorDeTransação`. No mundo CDI, esses tipos de classe que fazem algo **antes** (no nosso caso, executar o `begin`) e **depois** (`commit`) se chamam *Interceptors*. Devemos deixar isso claro no CDI, anotando a classe `GerenciadorDeTransação` com `@Interceptor` e a anotação com `@InterceptorBinding`, essa anotação diz que `@Transacional` está associada a um interceptador:

```
@Transacional
@Interceptor
public class GerenciadorDeTransacao implements Serializable {


@InterceptorBinding
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface Transacional {

}
```

Continuando a execução

Mas isso ainda não é suficiente. Repare que o método `executaTX` tem ainda esse comentário `// chamar os daos que precisam de um TX`! Hum, como vamos fazer isso? Claro que o CDI ajudará, pois é ele quem realmente sabe qual método precisa da transação, através da anotação `@Transacional`.

O CDI passará um parâmetro no método `executaTX`, que guarda a informação de quem precisa da transação. Esse objeto tem um nome "bonito": **contexto de invocação** ou em inglês `InvocationContext`. Através dele, podemos pedir para chamar o método, mas como não sabemos diretamente o nome do método foi dado um nome bem genérico, `proceed()`:

```
public void executaTX(InvocationContext contexto) throws Exception {

    manager.getTransaction().begin();
```

```
        // chamar os daos que precisam de um TX
        contexto.proceed();

        manager.getTransaction().begin();
    }
```

Ao chamar `proceed()` devemos lançar uma exceção: `throws Exception`

Então, quando chamamos `context.proceed()` continuaremos com a execução, ou seja o método será executado. Depois do método executar, a execução volta e comita a transação. Repare o fluxo no método `executa` :

- `begin` da transação
- continuar a execução do método anotado com `@Transacional`
- `commit` da transação

Mas como o CDI, ao ver a anotação `@Transacional` , saberá qual método deve executar? É executado algo **antes e depois** da chamada do método. O mundo do CDI chama o **antes e depois** de *Around* (ao redor), por isso devemos usar a anotação `@AroundInvoke` em cima do método:

```
@AroundInvoke
public void executaTX(InvocationContext contexto) throws Exception {

    manager.getTransaction().begin();

    // chamar os daos que precisam de um TX
    contexto.proceed();

    manager.getTransaction().begin();
}
```

Lidando com retorno no interceptador

Repare também que o método `gravar` do `AutorBean` , por exemplo, retorna uma `String` , mas poderia ser qualquer objeto. Como está o nosso método `executaTX` agora, esse retorno ficaria perdido. O método `proceed` devolve um objeto que representa o possível retorno dos métodos anotados. Vamos então retornar esse objeto no método `executaTX` :

```
@AroundInvoke
public Object executaTX(InvocationContext contexto) throws Exception {

    manager.getTransaction().begin();

    // chamar os daos que precisam de um TX
    Object resultado = contexto.proceed();

    manager.getTransaction().commit();

    return resultado;
}
```

Configuração do interceptador no beans.xml

A implementação do método já está perfeita, **mas não esqueça de anotar também com `@Transacional` os métodos `gravar` e `remover` de `AutorBean`.**

Por fim, precisamos configurar o configurar o `GerenciadorDeTransacao` no arquivo `WebContent/WEB-INF/beans.xml`:

```
<beans>
    <interceptors>
        <class>br.com.caelum.livraria.tx.GerenciadorDeTransacao</class>
    </interceptors>
</beans>
```

Podemos testar a nossa aplicação agora e ver que tudo continua funcionando corretamente! Foi um pouco trabalhoso criar um interceptador, mas criamos algo bem genérico e fácil de reutilizar. Os interceptadores são algo bem úteis e fazem parte de qualquer aplicação web. Bora fazer exercícios?