



01 Gerenciamento de Transações com JTA



Transcrição

No último capítulo, vimos como usar o JPA com EJB. O uso dentro de uma aplicação foi bastante simples, basta injetar o *EntityManager* para utilizar os métodos que acessam a persistência com JPA.

Vimos também que o JPA delega uma boa parte das configurações para o servidor JavaEE. É o JBoss quem fornece uma *DataSource* e que encapsula os detalhes da configuração do driver e do pool de conexão.

Nos exercícios, migramos todos os nossos DAOs para usar o *EntityManager*. Os DAOs foram injetados nos *Beans* através da anotação `@Inject`, sempre seguindo as boas práticas de injeção de dependências.

Vamos subir uma vez o servidor para mostrar que tudo continua funcionando. Enquanto o JBoss está iniciando, abrimos o terminal para se conectar com o MySQL. Todas as tabelas foram criadas com sucesso. E, ao executar um "select" na tabela `Usuario`, podemos ver que já existe um usuário cadastrado com o login *admin* e a senha *pass*.

Caso não tenha criado esse usuário basta executar o SQL seguinte:

```
insert into Usuario(login, senha) values('admin', '
```

[COPIAR CÓDIGO](#)

Por fim, vamos testar a aplicação pela interface gráfica. Após o login, vamos voltar ao Eclipse para verificar o console. Podemos ver o SQL gerado no console.

Como não tem nenhum autor, nem livro cadastrado, vamos inserir um pela interface. O Autor será o Paulo Silveira, e o livro será "Arquitetura Java".

Novamente, voltando para o console do Eclipse, podemos analisar o SQL gerado. Agora aparecem também os *inserts* gerados pelo JPA. Ótimo, o JPA está configurado e utilizado corretamente.

Transação JTA

Já conseguimos manipular os dados no banco através da nossa aplicação. Mas, como isso funcionou já que em nenhum momento nos preocupamos com o gerenciamento de uma transação? Isso é importante pois o MySQL precisa ter uma transação para realmente gravar os dados. A resposta é que o EJB Container automaticamente abriu e consolidou a transação sem ser necessário deixar isso explícito no código. Mais um serviço disponível para os EJBs!

Isso é bem diferente caso utilizemos o JPA fora de um EJB Container. Nesse caso seria necessário gerenciar a transação na mão, ou seja, usando os métodos `begin()` e `rollback()`. Podemos testar isso rapidamente, basta tentar usar o método `getTransaction()` de `EntityManager` para chamar `begin()` e `commit()`:

```
//é ilegal chamar getTransaction() dentro do EJB C  
manager.getTransaction().begin();  
manager.persist(autor);  
manager.getTransaction().commit();
```

[COPIAR CÓDIGO](#)

Vamos testar esse código uma vez e reiniciar o servidor JBoss. Após o login, navegamos para página de cadastro dos autores. Ao salvar um autor, aparece no console do Eclipse uma exceção. Repare que a mensagem da exceção deixa bem claro que não podemos utilizar o método `getTransaction()`.

Então, como e qual transação devemos utilizar? No final já tem alguma transação rodando! Um dica está no `persistence.xml`. Ao abrir e revisar a declaração do endereço do *data-source*, podemos ver o elemento *jta-data-source*. *JTA* significa *Java Transaction API* que é um padrão JavaEE que se preocupe com o gerenciamento da transação dentro de um servidor JavaEE. Para ser mais correto, o JTA é coordenador de transação e é ele quem vai coordenar a transação do JPA.

Antes de continuar e aprender mais sobre JTA, vamos apagar as chamadas do `getTransaction()` dentro do DAO.

Gerenciamento da transação com JTA

O JTA, então, é a forma padrão de gerenciar a transação dentro do servidor JavaEE e já funciona sem nenhuma configuração. Este padrão se chama *CONTAINER MANAGED TRANSACTION* (CMT).

Podemos deixar a nossa intenção explícita e configurar o gerenciamento pelo container. Para tal existe a anotação

`@TransactionManagement` que define o tipo de gerenciamento da transação, no nosso caso *CONTAINER*:

```
@Stateless
@TransactionManagement(TransactionManagementType.C
public class AutorDao {
```

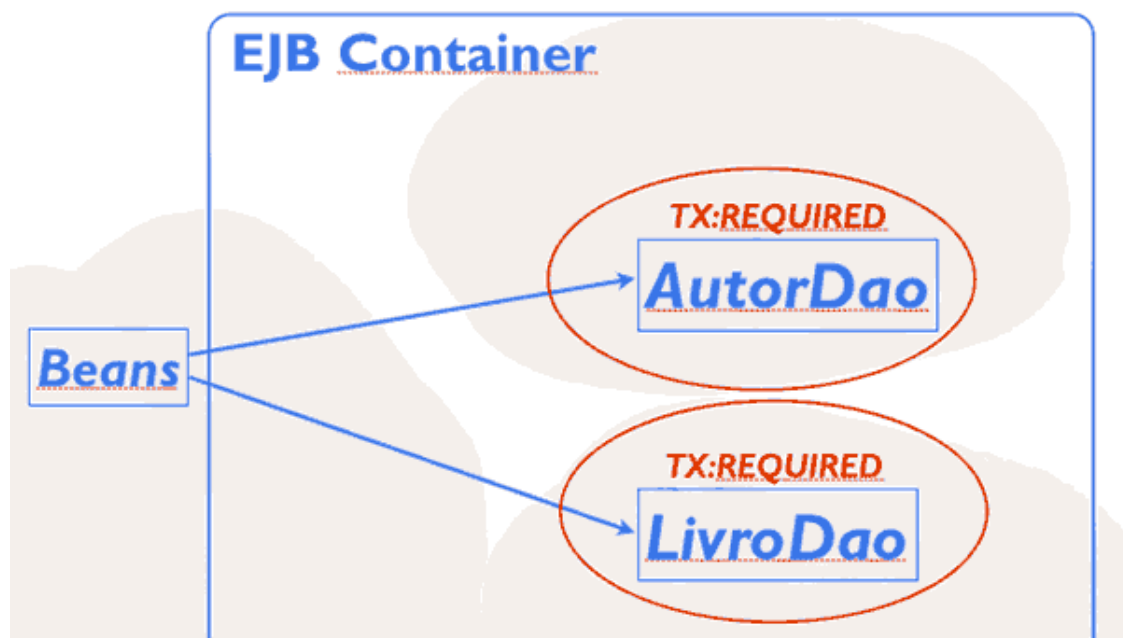
[COPIAR CÓDIGO](#)

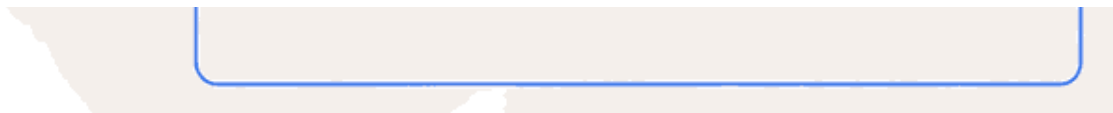
Essa configuração é totalmente opcional e serve apenas para fins didáticos. Igualmente, podemos deixar explícito o padrão da configuração (*atributo*) para cada método. Para isso existe a anotação `@TransactionAttribute` :

```
@TransactionAttribute(TransactionAttributeType.REQ
public void salva(Autor autor) {
```

[COPIAR CÓDIGO](#)

REQUIRED significa que o JTA garante uma transação rodando quando o método é chamado. Se não tiver nenhuma transação, uma nova é aberta. Caso já tenha uma rodando, a atual será utilizada. De qualquer forma, sempre é preciso ter uma transação (*REQUIRED*).





É importante ressaltar que o tipo de gerenciamento `CONTAINER` e o atributo `REQUIRED` já é o padrão adotado para um Session Bean, então não é necessário configurar. Ou seja, ao testar e republicar a aplicação, tudo deve continuar funcionando. Vamos acessar a aplicação e fazer um teste. Após login, podemos cadastrar um autor sem problemas. Ótimo.

TransactionAttribute

Através da anotação `@TransactionAttribute`, temos acesso a outras configurações. A primeira a testar é o *MANDATORY*. *MANDATORY* significa obrigatório. Nesse caso, o container verifica se já existe uma transação rodando, caso contrário, joga uma exceção. Ou seja, quem faz a chamada deve abrir uma transação.

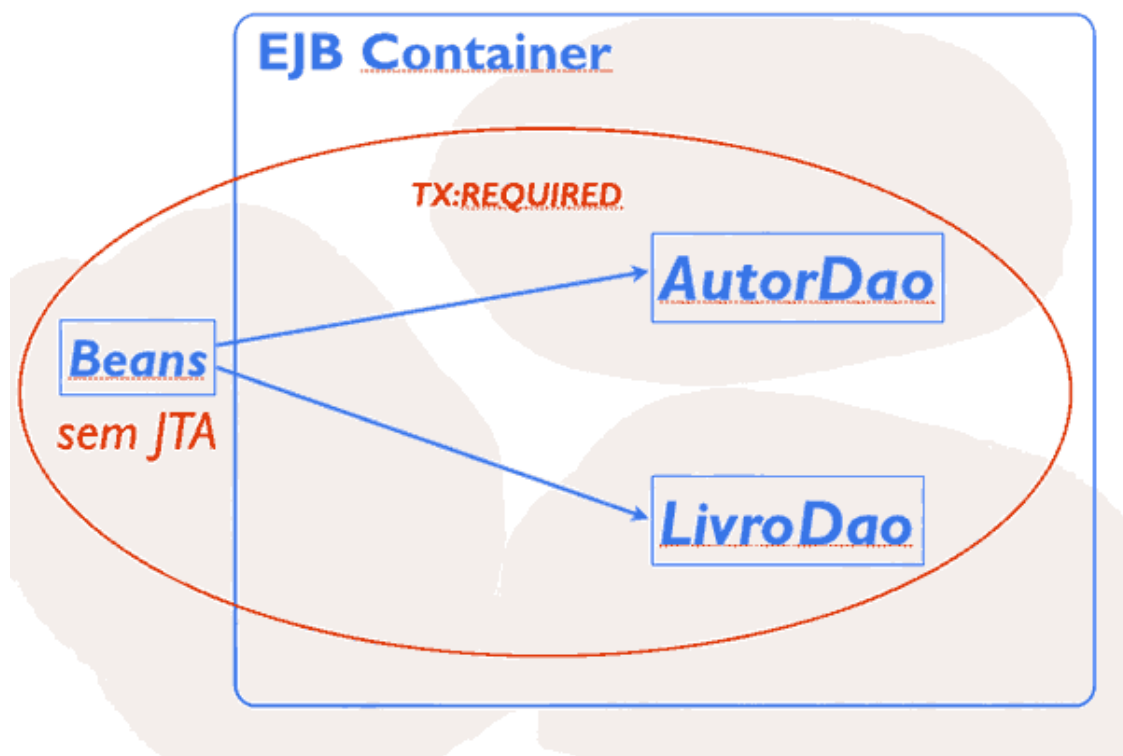
Vamos testar isso. Depois de ter republicado a aplicação pelo Eclipse, podemos acessar a interface web. Após login, vamos diretamente para o cadastro de autores. No entanto, ao cadastrar um autor, recebemos uma exceção. No console do Eclipse aparece o nome e a mensagem da exceção. Nesse caso foi lançado um `EJBTransactionRequiredException`.

Então, quando e como devemos utilizar o *MANDATORY* ?

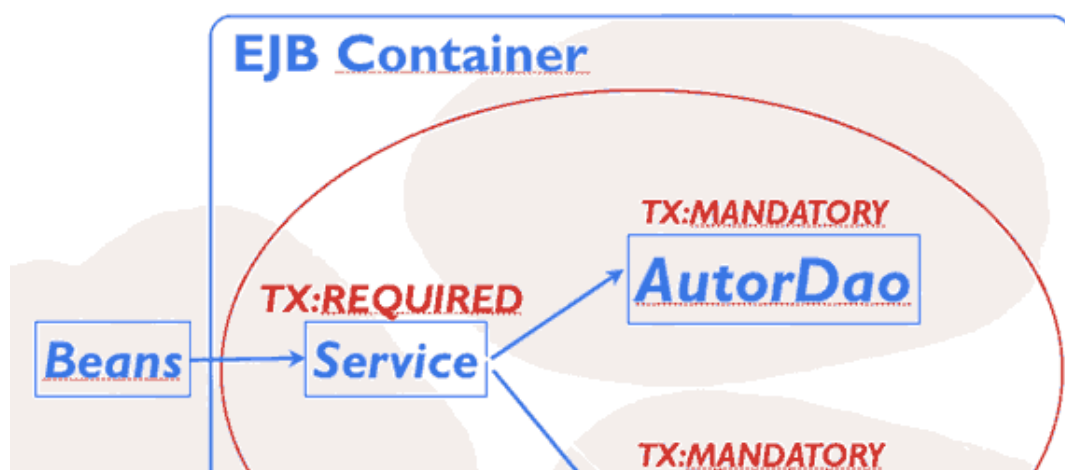
Normalmente, os DAOs não são o lugar ideal para abrir uma nova transação. Ao usar um DAO é preciso ter uma transação rodando. Quem faz a chamada precisa se preocupar com isso e abrir uma transação para o DAO funcionar.

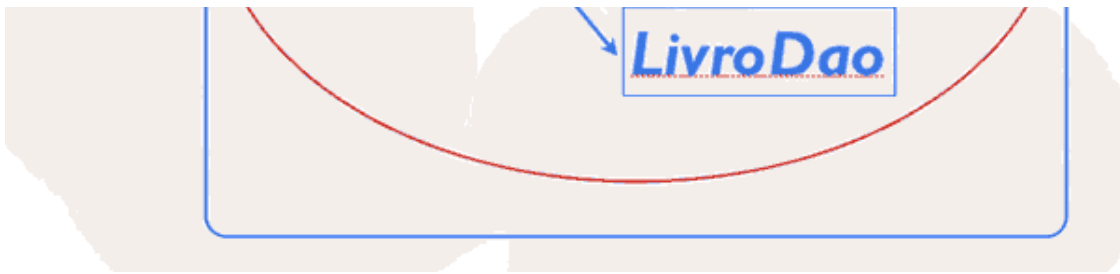
Serviços como Transaction boundary

Repare que na nossa aplicação são os BEANS que usam os DAOs, por exemplo o `AutorBean`. O problema aqui é que os BEANS não são EJBs (não são Session Beans) e por isso não têm acesso ao JTA.



Para resolver isso vamos criar uma classe intermediária, uma classe `AutorService` que fica entre os Beans e os DAOs. A classe `AutorService` também será um Session Bean e responsável por abrir uma nova transação. É ela quem recebe um `AutorDao` injetado e delega a chamada:





Na classe `AuthService`, vamos primeiro injetar o `AuthService` ::

```
@Stateless
public class AuthService {

    @Inject AuthService dao;
```

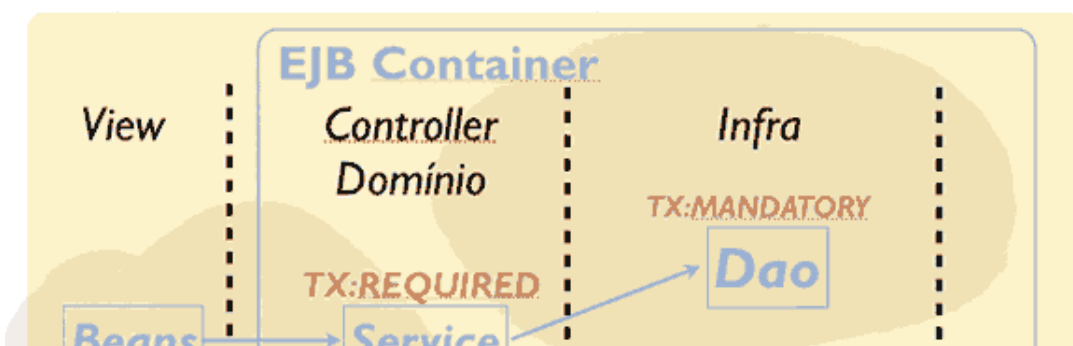
[COPIAR CÓDIGO](#)

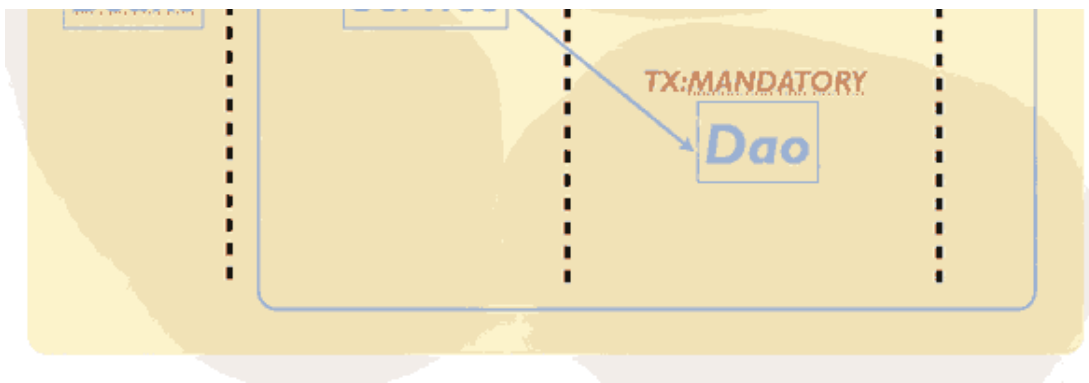
O método `adiciona(..)` recebe um `autor` e delega a chamada para o DAO:

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

[COPIAR CÓDIGO](#)

Nesse método poderiam ficar mais regras ou chamadas de regras de negócios. É muito comum ter essa divisão de responsabilidade entre bean, serviço e DAO em um projeto real. O bean possui muito código relacionado ao JSF (*view*), o serviço é o controlador na regra de negócio e o DAO possui o código de infraestrutura.





Por fim, falta refatorar a classe `AutorBean`, pois ela usa ainda `AutorDao`. Vamos injetar o `AutorService` e renomear a variável `dao`. Vamos também gerar o método `todosAutores()` que faz apenas a delegação para o DAO.

Como já falamos, o padrão `TransactionAttribute` é `REQUIRED`, ou seja, ao chamar o método `adiciona(..)` será aberta, automaticamente, uma nova transação.

Revisando, o bean recebe a chamada da tela, delega para o serviço que abre a transação e delega para o DAO.

Vamos atualizar a aplicação e testar pela interface web. Depois de ter passado pelo login, vamos tentar cadastrar mais um autor. Ao cadastrar, nenhuma exceção foi lançada e o autor aparece na tabela. Dessa vez, o `AutorDao` foi chamado dentro de uma transação existente. Ótimo.

Outros atributos

Além do `REQUIRED` e `MANDATORY`, há outros atributos disponíveis. O `REQUIRES_NEW` indica que sempre deve ter uma nova transação rodando. Caso já exista, a transação atual será suspensa para abrir uma nova. Caso não tenha nenhuma rodando, será criada uma nova transação.

Outro atributo `NEVER` é quem indica que jamais deve haver uma transação em execução. Isso pode ser útil para métodos que obrigatoriamente devem ser executados sem contexto transacional. Vamos testar isso uma vez para mostrar o funcionamento. Novamente vamos publicar e acessar a aplicação. Ao cadastrar o autor, recebemos uma exceção. A mensagem deixa claro que a configuração do `TransactionAttributeType` não permite a chamada dentro da transação.

Além disso, existem outros atributos como `SUPPORTS` e `NOT_SUPPORTED` que veremos nos exercícios.

Gerenciamento da transação programaticamente

O gerenciamento da transação pelo container é uma das vantagens do EJB e sempre deve ser a maneira preferida de se trabalhar. Contudo existe uma outra forma, parecida com aquela mostrada baseado no `EntityManager`. Essa forma permite o controle programaticamente, chamando *begin()* ou *commit()* na mão.

Para o EJB Container *aceitar* o gerenciamento da transação programaticamente, é preciso reconfigurar o padrão. Ou seja, ao invés de usar `CONTAINER` na anotação `TransactionManagement` usaremos `BEAN`, porque o Session Bean vai gerenciar a transação (também é chamado *BEAN MANAGED TRANSACTION*). Assim também podemos apagar a anotação `@TransactionAttribute` que não faz mais sentido.

Para realmente gerenciar a transação, é preciso injetar um objeto com este papel. Para este propósito existe a interface `UserTransaction` do JTA. Basta injetar o objeto através da anotação `@Inject`:

```
@Inject UserTransaction tx;
```

[COPIAR CÓDIGO](#)

`UserTransaction` possui os métodos clássicos relacionados com o gerenciamento da transação como `begin()`, `commit()` e `rollback()`. O problema é que exige um tratamento excessivo de exceções *checked* que poluem muito o código.

Vamos colocar as chamadas dos métodos `begin()` e `commit()` dentro de um *try-catch*. O Eclipse ajuda nessa tarefa e gera automaticamente o bloco de tratamento. No nosso exemplo, para simplificar o entendimento, vamos capturar qualquer exceção, ou seja, *fazer um* `catch(Exception)`:

```
public void salva(Autor autor) {  
  
    //...  
    try {  
        tx.begin();  
        manager.persist(autor);  
        tx.commit();  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    //...  
}
```

[COPIAR CÓDIGO](#)

Falta testar a aplicação. Vamos publicar as alterações e acessar a aplicação pelo navegador. O objetivo é cadastrar mais um autor para ver se o gerenciamento da transação realmente continua funcionando. Ao inserir, o autor aparece na tabela, o que indica que foi cadastrado com sucesso.

