01 **Lidando com Exceções**

Transcrição

No último capítulo falamos sobre o gerenciamento da transação. Vimos que o EJB Container usa o JTA (*Java Transaction API*) para o gerenciamento. O JTA é apenas um coordenador de transações que já vem com o servidor de aplicação.

O uso do JTA na aplicação é transparente e não exige muita configuração. Por padrão, qualquer chamada ao Session Bean já é transacional. Vimos que a anotação `@TransactionAttribute` permite configurar detalhes sobre o gerenciamento. Por exemplo, podemos definir que sempre é preciso ter uma nova transação (*REQUIRED*) ou que ela é obrigatoria (*MANDATORY*).

Também vimos que é possível gerenciar a transação programaticamente através do objeto `UserTransaction` , algo que deve ser evitado ao máximo já que o gerenciamento pelo container é simples e atende a maioria dos cenários.

Lidando com exceções

Já percebemos que podem acontecer exceções durante a execução da aplicação, mas como o container EJB lida com elas? Além disso quais são as formas que o desenvolvedor tem para mitigar um problema?

Vamos imaginar que dentro do DAO, além de cadastrar o autor no banco, também seja feito uma chamada para um serviço externo (um web service por exemplo). Nessa comunicação a rede pode falhar ou o serviço pode ficar desligado temporariamente. Esse são problemas que não dependem da aplicação.

Ou seja, nesse caso inevitavelmente vai ocorrer uma exceção. Para simular

este problema, vamos lançar uma exceção no fim do método `salva()` da classe `AutorDao`:

```
throw new RuntimeException("Serviço externo deu
erro!");
```

COPIAR CÓDIGO

Vamos republicar a aplicação e acessá-la pela interface, algo nada novo pra nós. Depois do login, vamos testar o cadastro de autores que executa justamente o método que causa a `RuntimeException`.

O resultado não é uma surpresa. No console do Eclipse aparece o *Stacktrace*, ou seja, a pilha de execução com a exceção em cima dela. Podemos ver que nossa exceção foi quem causou o problema.

Ainda no console vamos subir mais um pouco, quase no início. Aí podemos ver que a nossa exceção foi "embrulhada" em uma outra do tipo `EJBTransactionRollbackException`. O nome indica que foi feito um rollback da transação.

É importante deixar isso claro, pois antes de lançar a exceção já usamos o JPA para persistir o autor. Sem *rollback* o autor estaria salvo no banco. Para ter certeza disso, vamos verificar o MySQL no terminal. Nele executaremos um *select* para verificar a tabela `Autor`. Como já esperávamos, não foi salvo o autor por causa do lançamento da exceção.

Exceções da Aplicação

Vamos comentar a `RuntimeException` dentro da classe `AutorDao` e abrir a classe `AutorService`. Em uma aplicação real, essas classes de serviços são utilizadas para coordenar as chamadas de regras de negócios.

É claro que ao chamar alguma regra também podem aparecer alguns problemas. Alguns deles são previstos e farão parte do negócio. É comum que uma validação falhe e um valor não seja salvo, pois uma regra específica não permite. Repare que este tipo de problema não é relacionado com a

infraestrutura e sim com o domínio da aplicação.

Vamos simular isso uma vez e causar uma nova exceção, mas agora uma exceção do tipo `LivrariaException`:

```
throw new LivrariaException();
```

COPIAR CÓDIGO

Como essa classe ainda não existe no projeto, o Eclipse reclama e o código não compila. Podemos facilmente resolver isso gerando a classe com a ajuda da IDE. Basta digitar `ctrl+1` na linha com o erro de compilação e o Eclipse abre o diálogo para a criação da classe.

Repare também que a classe `LivrariaException` já estende a classe `Exception`. Vamos confirmar o diálogo para criar a classe. Na nova classe o Eclipse ainda mostra um alerta em amarelo, que não representa um erro de compilação e é irrelevante para o projeto.

Voltando para a classe `AutorService`, surge agora um outro problema. Ao lançar a exceção `LivrariaException` é preciso deixar o tratamento explícito. Essa exceção, diferente do exemplo anterior, é do tipo *checked*, ou seja, é necessário o uso do *try-catch* ou *throws*.

No nosso caso vamos colocar o *throws* na assinatura do método. Isso significa que agora a classe `AutorBean` não compila mais, já que a chamada do serviço causa uma exceção *checked*. Faremos a mesma coisa usando na assinatura do método o *throws*. Pronto, tudo está compilando.

Vamos testar a aplicação, *full publish* para atualizar o JBoss e depois abrir o navegador. Após o login, novamente testaremos o cadastro de autores. Vamos tentar salvar o autor *Mauricio Aniche*.

Já podemos ver que o autor não aparece na tabela da interface web. Como esperado, no Eclipse aparece a exceção igual ao exemplo anterior. Ao analisar o console podemos ver a `LivrariaException`, mas dessa vez ela não foi "embrulhada" dentro de uma `EJBTransactionRollbackException`.

Para ter certeza, vamos verificar o banco de dados. Novamente selecionaremos todos os autores da tabela `Autor`. Para nossa surpresa o autor *foi salvo*! Mesmo sendo lançado uma exceção na pilha de execução, foi feito um commit na transação!

Ao atualizar a interface web, podemos ver que realmente aparece o autor. Como, então, o container lida com as exceções?

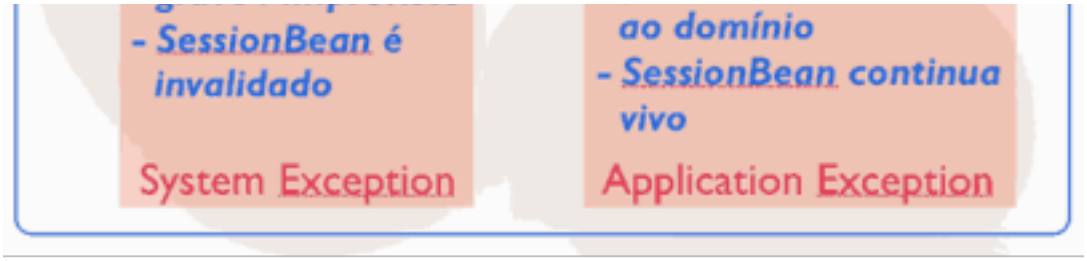
System e Application Exceptions

Vimos dois comportamentos diferentes do container referentes a exceção. O primeiro exemplo foi uma exceção do tipo *unchecked que causou um rollback*, e o segundo exemplo usou uma exceção *checked que não causou rollback*.

Pelo ponto de vista do container, o primeiro exemplo representa uma *System Exception*, algo grave e imprevisto. *System Exception* sempre causam *rollback*. Além disso, aquele Session Bean que lançou a exceção é invalidado e retirado do pool de objetos.

O segundo exemplo representa uma *Application Exception*. Que é uma erro que pode acontecer durante a vida da aplicação e é relacionado ao domínio. Por isso não causa *rollback* e nem invalida o Session Bean.

Por padrão, qualquer exceção *unchecked é uma System Exception* e qualquer exceção *checked é uma Application Exception*. Isso é o padrão do EJB Container, mas como já vimos anteriormente, esse padrão pode ser reconfigurado.



Configurando Application Exceptions

Vamos abrir a classe `LivrariaException` e deixar explícito que ela é uma *Application Exception*. Para isso usaremos a anotação `@ApplicationException` que possui atributos para redefinir o comportamento referente a transação. Vamos fazer uma configuração para que essa *Application Exception* cause sim um *rollback*:

```
@ApplicationException(rollback=true)
public class LivrariaException extends Exception{

}
```

COPIAR CÓDIGO

Como sempre, vamos testar o novo comportamento. Ao cadastrar um autor pela interface web é lançado uma `LivrariaException` . Novamente a exceção aparece no console do Eclipse e repare também que essa exceção não foi embrulhada. Até aqui é tudo igual. No entanto, ao verificar o banco de dados, percebemos que o autor não foi salvo, ou seja, foi feito um rollback da transação.

Por fim, uma vez declarado a `LivrariaException` como `@ApplicationException` , podemos deixar ela *unchecked*. Isso significa que não precisamos estender a classe `Exception` e sim `RuntimeException` . Assim, o compilador não obriga o desenvolvedor a fazer um tratamento explicito da exceção. Podemos, então, apagar aquelas declarações *throws* na assinatura dos métodos no `AutorBean` e no `AutorService` .

Vamos fazer os exercícios?