01

Interceptor

Transcrição

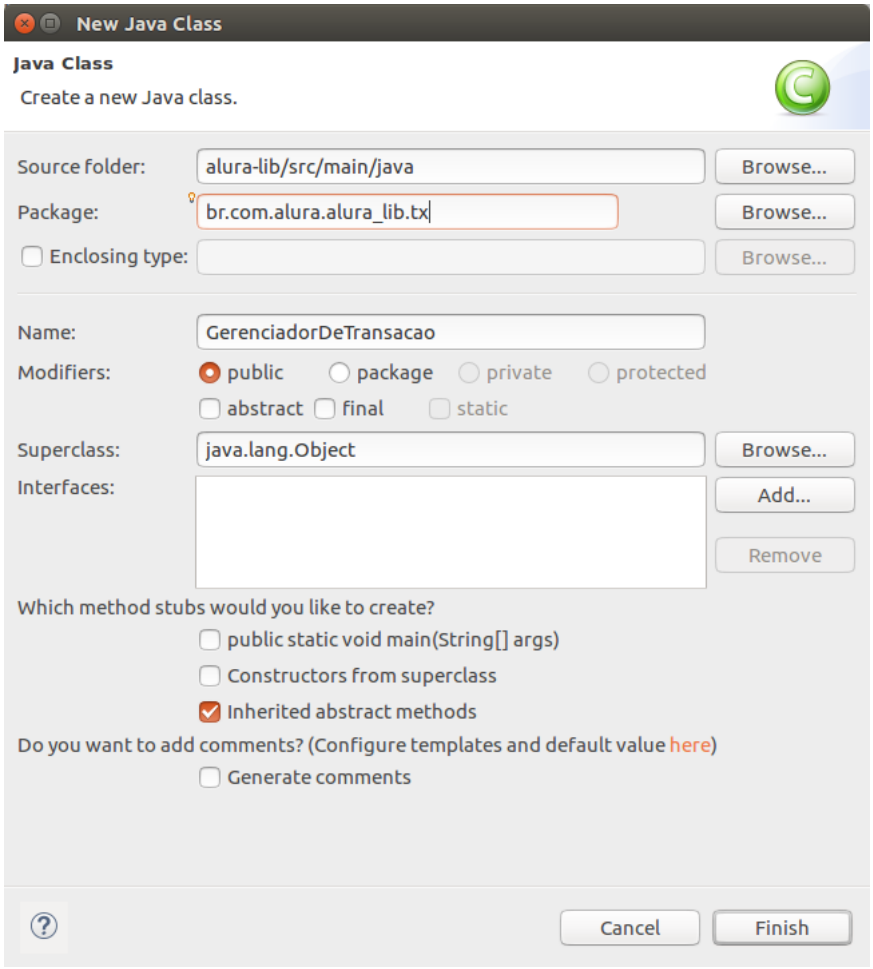
Agora nós temos dois projetos dentro do Eclipse: uma biblioteca e o projeto em que esta é utilizada. Nosso objetivo é criar novas funcionalidades para a biblioteca e utilizá-las no projeto `livraria`.

Dentro do projeto `alura-lib`, copiamos o nosso `DAO`. Todos os métodos do nosso `DAO` que realizam uma operação de modificação no banco (inserir, remover, atualizar) tem um código em comum:

```
public void adiciona(T t) {

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

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

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

Todos esses métodos abrem um transação e depois a *commitam*. Sabemos que a repetição de código é ruim. O ideal será isolarmos o comportamento repetido.

Vamos criar uma nova classe, chamada `GerenciadorDeTransacao`, dentro do pacote `br.com.alura.alura_lib.tx`.



Essa classe vai conter um método que isolará o comportamento repetido. Vamos começar copiando o código que se repete:

```
public class GerenciadorDeTransacao {

    private EntityManager em;

    @Inject
    public GerenciadorDeTransacao(EntityManager em) {
        this.em = em;
    }

    public void executaComtransacao() {
        em.getTransaction().begin();

        // Faz alguma coisa

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

Nós queremos que entre a abertura da transação e o *commit*, alguma lógica seja executada. Como isolamos esse comportamento, no `DAO` vamos remover todas as linhas que abrem e *commitam* a transação. Exemplo de como os métodos ficaram:

```
public void adiciona(T t) {
    em.persist(t);
}
```

```
public void remove(T t) {
    em.remove(em.merge(t));
}

public void atualiza(T t) {
    em.merge(t);
}
```

No projeto `livraria`, na classe `AutorBean`, no método gravar(), temos uma chamada para `adiciona()` de `DAO<Autor>`.

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

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

    this.autor = new Autor();

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

Nosso objetivo é que quando fizermos uma chamada do tipo `autorDao.adiciona(this.autor)`, em vez de chamar diretamente o `adiciona()`, ele passará pelo `GerenciadorDeTransacao`. Desta forma, o `GerenciadorDeTransacao` abre a transação, executa o método `adiciona()` e depois *commita* a transação. Existe uma especificação do Java EE, a especificação de *Interceptors* que nos ajuda a fazer essa tarefa.

Ela intercepta uma requisição, um método, um construtor, antes de destruir um objeto, depois de construir um objeto... Exemplo, ao chegar uma requisição para o método `adiciona()` do `AutorDao`, o `GerenciadorDeTransacao` vai antes abrir a transação, executar o método e *commita-la*. Vamos informar que o `GerenciadorDeTransacao` é um *Interceptor*. Para isso vamos anotá-lo com `@Interceptor`.

```
// ...
import javax.interceptor.Interceptor;

@Interceptor
public class GerenciadorDeTransacao {
// ...
```

Os momentos como a chamada de um método, que conseguiremos interceptar, são chamados de *callback*. No método `executaComtransacao()` precisaremos dizer qual *callback* desejamos interceptar. No caso, utilizamos a anotação `@AroundInvoke`. Além disso, quando trabalhamos com *Interceptors* temos uma interface chamada `InvocationContext` e por meio dela conseguimos invocar o método que está sendo invocado. Para isso basta chamar o método `proceed()`.

Como o método `proceed()` não sabe se tudo vai funcionar como esperado, ele lança uma *exception*. Nesse exemplo, se houver alguma exceção, o ideia é que ele faça o *roolback* da transação iniciada. Portanto vamos adicionar um `try-catch` e tratar isso. A implementação do método será a seguinte:

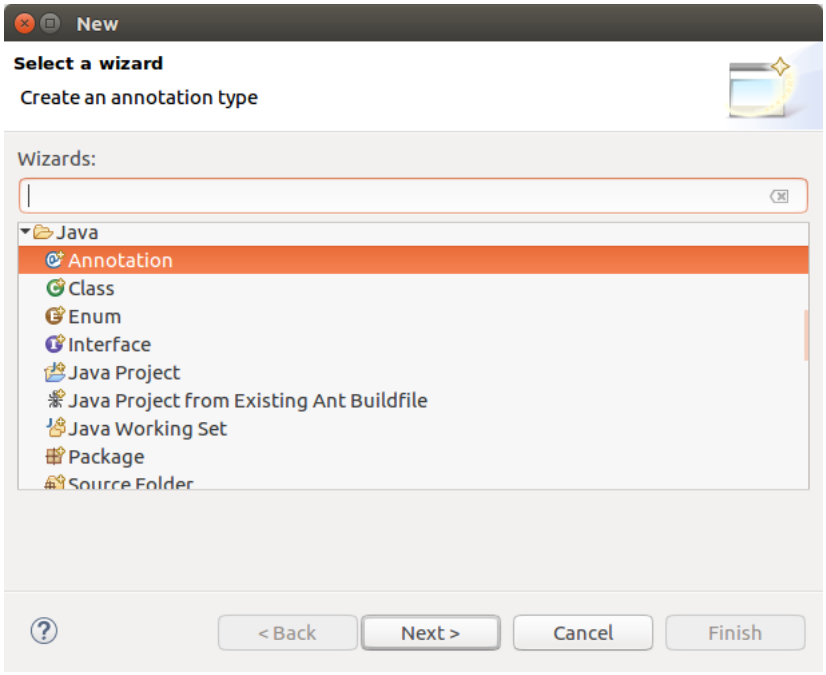
```
@AroundInvoke
public void executaComtransacao(InvocationContext context) {

    em.getTransaction().begin();

    try {
        context.proceed();
        em.getTransaction().commit();
    } catch (Exception e) {
        em.getTransaction().rollback();
        throw new RuntimeException(e);
    }
}
```

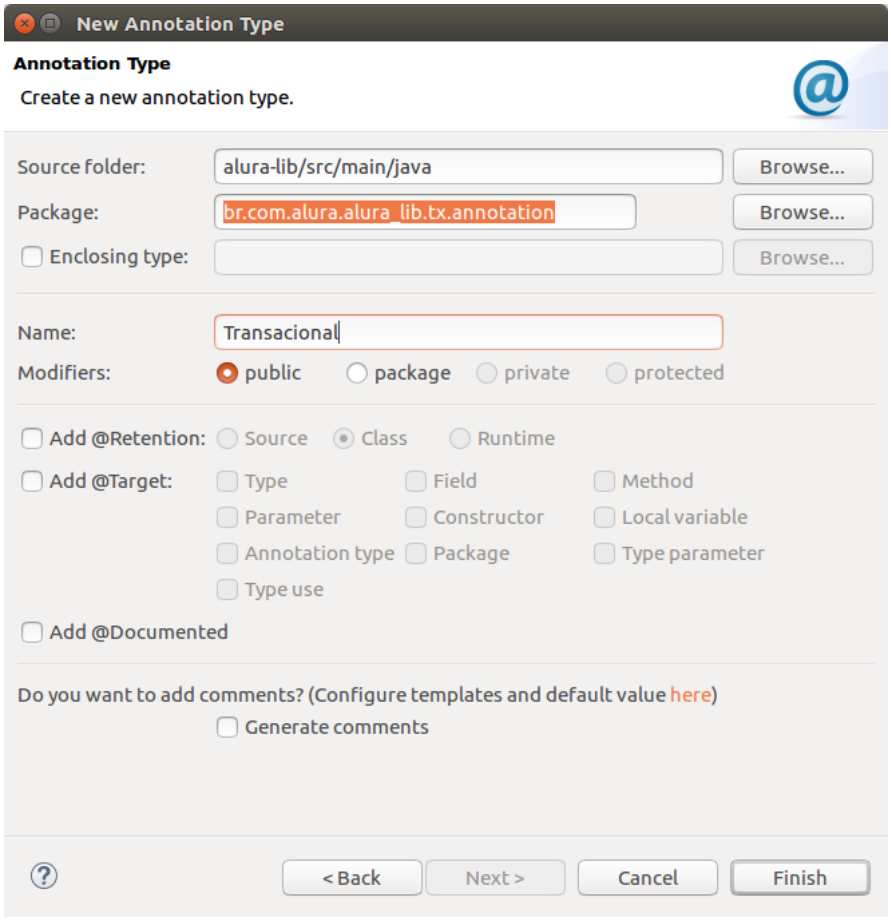
O *Interceptor* vai ficar sempre na frente, porque vai chegar uma requisição, passar pelo *Interceptor*, executar o método que está sendo interceptado e depois voltando pelo *Interceptor*. A execução pode retornar alguma coisa. O método `proceed()` retorna um `Object` e podemos passá-lo para quem está chamando, no caso desse exemplo o `LivroBean`.

```
@AroundInvoke
public Object executaComtransacao(InvocationContext context) {

    em.getTransaction().begin();

    try {
        Object resultado = context.proceed();
        em.getTransaction().commit();

        return resultado;
    } catch (Exception e) {
        em.getTransaction().rollback();
        throw new RuntimeException(e);
    }
}
```

Criando uma annotation

Agora precisamos marcar os pontos onde desejamos ter essas transações. Indicando quais métodos devem ser interceptados. Vamos demarcar essas transações com uma anotação. Dentro do pacote `br.com.alura.alura_lib.tx.annotation`, vamos criar uma annotation chamada `Transacional`.





Sempre que criamos uma nova anotação, precisamos definir duas coisas. A primeira é onde ela poderá ser utilizada (em métodos, campos, construtores, etc). Isso é definido pela anotação `@Target`, onde passamos um array com esses elementos.

O outro ponto é que precisamos definir por quanto tempo essa anotação vai durar. Ela deveria estar disponível somente para o meu compilador utilizar? Ou é necessário que ela esteja no código? Ou vamos precisar dela até o tempo de execução, de forma que em tempo de execução seja possível verificar se a anotação está presente e seja possível tomar uma ação? Este último caso é o que queremos.

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

}
```

Agora vamos utilizar a anotação `@InterceptorBinding` para associar a anotação que estamos criando (`Transacional`) com o `GerenciadorDeTransacao` .

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

}
```

Para finalizar temos que anotar a classe `GerenciadorDeTransacao` com essa a anotação `Transacional` :

```
@Interceptor
@Transactional
public class GerenciadorDeTransacao {
```

Vamos receber um erro porque marcamos que a anotação deve ser utilizada apenas em um método. Vamos adicionar o `ElementType.TYPE` , para indicar que ela pode ser utilizada em cima de uma classe.

```
@InterceptorBinding
@Target({ElementType.METHOD, ElementType.TYPE})
// ..
```

Utilizando a anotação

Agora já podemos utilizar a anotação! Poderíamos ir no nosso `DAO` e anotar métodos que necessitam de transação, como o `adiciona()` , o `remove()` e o `atualiza()` .

Mas vamos imaginar um exemplo: suponha que nós tivéssemos um sistema de estoque, e em `LivroBean` , no método `gravar()` , após adicionar um livro fosse necessário atualizar o estoque - afinal, temos um item a mais. Se der algum problema, seja no momento de gravar o livro ou no momento de adicionar no estoque, iríamos querer que ele revertesse tudo. Precisamos que ele reverta a gravação do Livro e a gravação no estoque.

O ideal é que os métodos do `LivroBean` tenham transação. Desta forma, toda a chamada do método do bean fica envolta em uma transação e se houver mais de uma operação no banco de dados dentro do método, tudo ficará dentro dessa mesma transação.

Para poder utilizar a anotação `@Transacional` no projeto `livraria` , precisamos antes instalar o projeto no repositório local. Aqui já vimos os passos: botão direito no projeto `alura-lib` , "Run As > Maven install". Enquanto no projeto `livraria` , precisamos clicar com o botão direito e "Maven > Update Project".

Agora já é possível, em cima dos métodos `gravar()` e `remover()` do `LivroBean` , utilizar a anotação `@Transacional` :

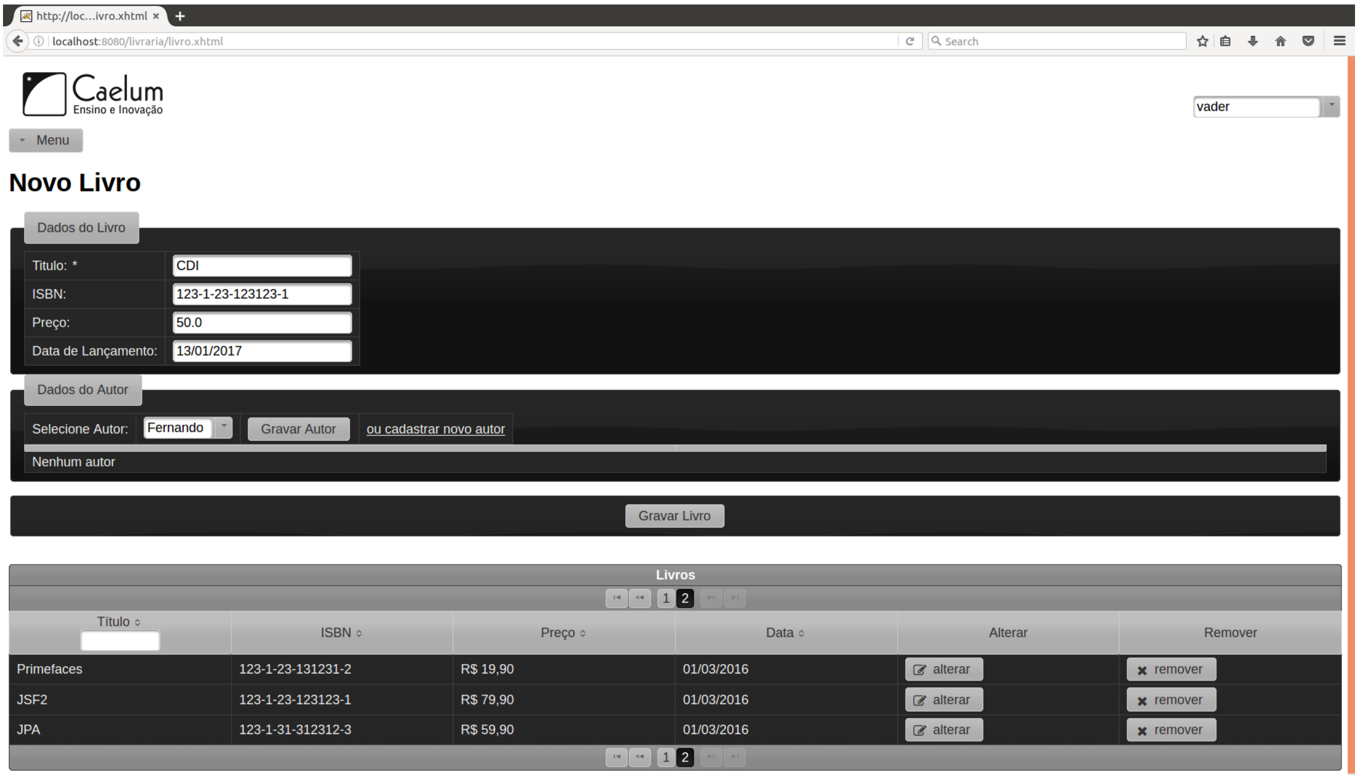
```
@Transacional
public void gravar() {
    System.out.println("Gravando livro " + this.livro.getTitulo());

    if (livro.getAutores().isEmpty()) {
        FacesContext.getCurrentInstance().addMessage("autor",
                new FacesMessage("Livro deve ter pelo menos um Autor."));
        return;
    }

    DAO<Livro> dao = livroDao;
    if(this.livro.getId() == null) {
        dao.adiciona(this.livro);
        this.livros = dao.listaTodos();
    } else {
        dao.atualiza(this.livro);
    }

    this.livro = new Livro();
}

@Transacional
public void remover(Livro livro) {
    System.out.println("Removendo livro");
    DAO<Livro> dao = livroDao;
    dao.remove(livro);
    this.livros = dao.listaTodos();
}
```

Outros métodos que irão precisar da anotação serão os métodos `gravar()` e `remover()` da classe `AutorBean`.

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

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

    this.autor = new Autor();

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

@Transacional
public void remover(Autor autor) {
    System.out.println("Removendo autor " + autor.getNome());
    autorDao.remove(autor);
}
```

Agora vamos clicar com o botão direito no Tomcat e em seguida escolher a opção "Clean". No diálogo de confirmação vamos clicar em "Ok" e em seguida, vamos iniciar/reiniciar o Tomcat.

Vamos cadastrar um novo livro para ver se tudo está funcionando:



O último título cadastrado era o JPA. Quando clicamos no botão "Gravar Livro" e atualizarmos a página, o último livro continuará sendo o de JPA, ou seja, o item não foi gravado.

Se observarmos a saída do console:

```
Gravando livro CDI
Hibernate: select livro0_.id as id1_, livro0_.dataLancamento as dataLanc2_1_, livro0_.isbn as isbn1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
```

Podemos perceber que o Hibernate não executou nenhum `insert`, apenas vários `selects`. Por que nosso *interceptor* não foi utilizado? Porque precisamos habilitar o *interceptor* para que ele funcione. No projeto `livraria`, vamos ao nosso `beans.xml` e adicionar o *interceptor*.

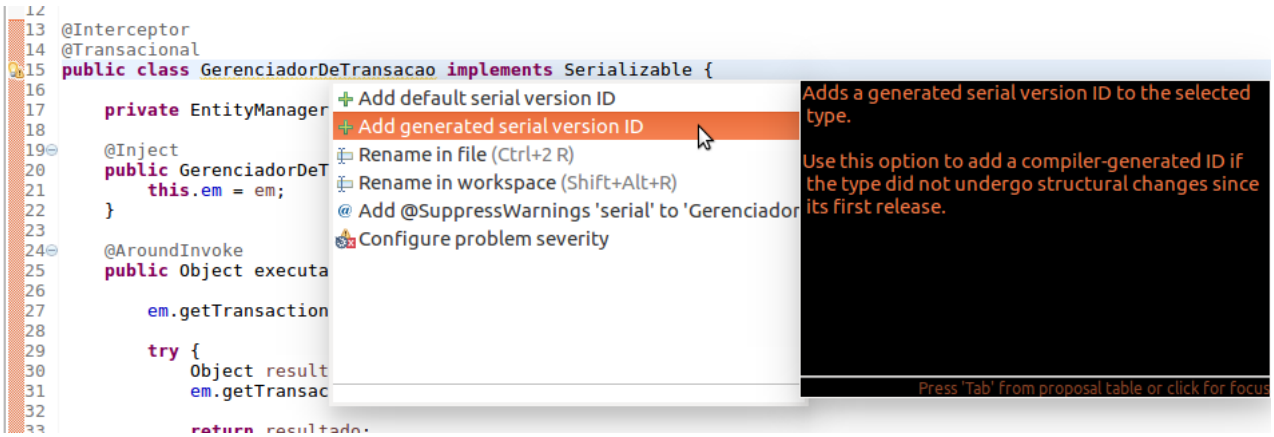
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/bea
        version="1.2" bean-discovery-mode="all">

    <interceptors>
        <class>br.com.alura.alura_lib.tx.GerenciadorDeTransacao</class>
    </interceptors>
</beans>
```

Ao dar um *clean* no Tomcat e reiniciá-lo, recebemos um erro. Ao observar a saída no Console, podemos ver o seguinte:

```
WELD-001527: Managed bean declaring a passivating scope has a non-serializable interceptor.  Bean:
```

Como estamos utilizando um *interceptor* vindo de um `jar`, que não está declarado diretamente no nosso projeto, precisamos que esse *interceptor* seja serializável. No projeto `alura-lib`, vamos fazer com que o *interceptor* implemente a interface `Serializable`. Movemos o cursor para o nome da classe e pressionamos "Ctrl + 1". Em seguida escolhemos a opção "Add generated serial version ID".



Vamos repetir o mesmo processo de antes. Instalar o `alura-lib` no repositório local, rodar um "Maven > Update Project" no projeto `livraria`, e dar um *clean* e reiniciar o Tomcat.

Agora, ao tentar gravar o livro da mesma forma que anteriormente, tudo funcionou e o livro de CDI é listado.

Além de ser possível habilitar o *interceptor* no arquivos `beans.xml` do projeto `livraria`, existe uma outra forma de fazer isso. Vamos comentar as linhas que adicionamos no arquivo `beans.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/be
        version="1.2" bean-discovery-mode="all">
```

```
    <!-- <interceptors>
        <class>br.com.alura.alura_lib.tx.GerenciadorDeTransacao</class>
    </interceptors> -->
</beans>
```

No próprio `GerenciadorDeTransacao`, podemos definir a prioridade do *interceptor* com a anotação `@Priority`, onde passamos um inteiro que define a prioridade. O `Inteceptor` possui uma subclasse chamada `Priority` que possue algumas constantes com valores pré-definidos. Alguns dos valores dessas constantes são:

```
public static final int PLATFORM_BEFORE = 0;
public static final int LIBRARY_BEFORE = 1000;
public static final int APPLICATION = 2000;
public static final int LIBRARY_AFTER = 3000;
public static final int PLATFORM_AFTER = 4000;
```

Em vez de ficar utilizando qualquer valor dentro da anotação `@Priority`, podemos utilizar essas constantes para definir qual será a prioridade do *interceptor*. Dessa forma definimos também a ordem do *interceptor*. Se temos um *interceptor* de 2000, ele será executado antes de um *interceptor* que tenha 2010. Algo que é bastante utilizado, é usar o valor da constante acrescido de um número, para definir um *interceptor* depois do outro:

```
@Priority(Interceptor.Priority.APPLICATION + 50)
```

No nosso caso como temos apenas um *interceptor*, não é necessário adicionar valor. Poderá ficar assim:

```
@Interceptor
@Transacional
@Priority(Interceptor.Priority.APPLICATION)
public class GerenciadorDeTransacao implements Serializable {
```

Vamos repetir o mesmo processo de antes. Instalar o `alura-lib` no repositório local, rodar um "Maven > Update Project" no projeto `livraria`, e dar um *clean* e reiniciar o Tomcat.

Ao tentar remover o livro adicionado de CDI, podemos ver no console que o HIbernate executou um delete, o que significa que o nosso *interceptor* está funcionando.

```
Removendo livro
Hibernate: select livro0_.id as id1_0_, livro0_.dataLancamento as dataLanc2_1_0_, livro0_.isbn as is
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: delete from Livro_Autor where Livro_id=?
Hibernate: delete from Livro where id=?
Hibernate: select livro0_.id as id1_, livro0_.dataLancamento as dataLanc2_1_, livro0_.isbn as isbn1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
Hibernate: select autores0_.Livro_id as Livro1_1_1_, autores0_.autores_id as autores2_3_1_, autor1_
```