01 **Novos serviços com Interceptadores**

Transcrição

Durante do treinamento usaremos alguns arquivos de configuração. Se você não baixou ainda, todos disponíveis estão aqui: resources.zip (https://s3.amazonaws.com/caelum-online-public/ejb/resources.zip)

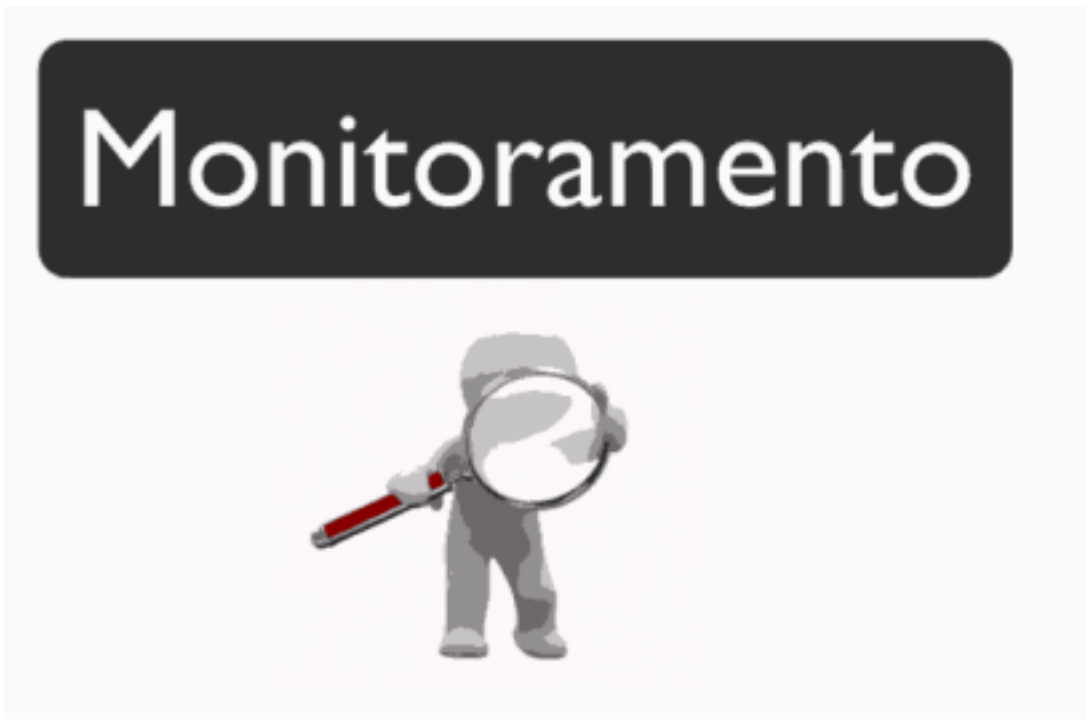
Revisão

No último capítulo vimos como o *EJB container* se comporta quando uma exceção é lançada. Aprendemos que o *container* diferencia entre *Application* e *System Exception*. Por padrão, exceções *unchecked* são do sistema e causam um *rollback* da transação. Ao contrário, exceções *checked* são da aplicação e não causam *rollback*. Há uma anotação, `@ApplicationException`, que reconfigura o padrão referente à exceções da aplicação.

Nesse capítulo vamos conhecer os poderosos ***Interceptors*** (interceptadores).

Monitoramento com Interceptadores

Na maioria das aplicações é preciso fazer um monitoramento, principalmente quando a aplicação não está muito madura ainda. Por exemplo, pode ser necessário monitorar o tempo de acesso ao banco de dados.

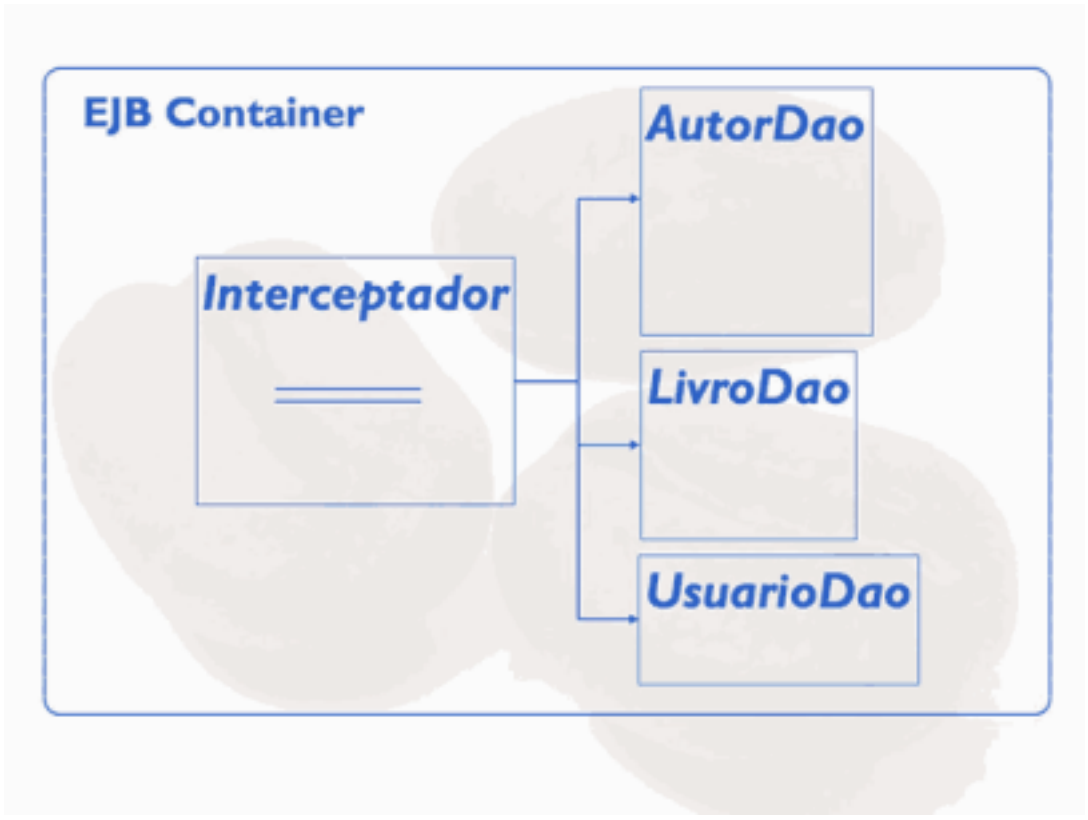


Para implementar esta funcionalidade, a primeira idéia seria abrir cada método e colocar um código para isso. Vamos mostrar isso na classe `AutorDao`. No início do método `salva(...)`, guardamos o retorno do `System.currentTimeMillis()` em uma variável auxiliar para o tempo, e no final do método imprimimos a diferença entre o tempo atual e a variável auxiliar. Veja o código:

```
public void salva(Autor autor) {
    long millis = System.currentTimeMillis();

    // código já existente omitido

    System.out.println("Tempo gasto: " + (System.currentTimeMillis() - millis));
}
```

COPIAR CÓDIGO

O problema é que esse código, mesmo sendo muito simples, repete-se em cada método que faz parte desse monitoramento. Terão muitos DAOs no projeto e isso não só polui o código como também gera problemas de manutenção. Ideal mesmo seria colocar essa parte do código em um só lugar. Para tal existem os interceptadores que separam bem essa responsabilidade em um único lugar.



O primeiro interceptador

Vamos criar uma nova classe dentro do *package* `interceptador`. Chamaremos a classe de `LogInterceptador`. Ela terá apenas um método e poderemos escolher livremente o seu nome, no nosso caso usaremos `intercepta`.

```
public class LogInterceptador {

    public Object intercepta() {

    }
}
```

COPIAR CÓDIGO

Há algumas obrigações na assinatura. A primeira é que o método deve retornar um `Object`. Vamos entender ainda o porquê, mas o importante agora é que dentro desse método faremos o monitoramento desejado. Aquele código que mede o tempo ficará no método `intercepta`, em um lugar só.

```
public Object intercepta() {

    long millis = System.currentTimeMillis();

    // chamada do método do dao

    System.out.println("Tempo gasto:  " + (System.currentTimeMillis() -
millis));
}
```

COPIAR CÓDIGO

Repare que entre essas duas linhas de monitoramento é preciso chamar o método do DAO. Como faremos isso? Temos que ter algum objeto que possui esse poder. Aí entra um objeto especial que estará automaticamente disponível dentro do interceptador e que se chama `InvocationContext`. Através dele podemos continuar a execução da aplicação, ou seja, chamar o método no DAO. O `InvocationContext` possui um método `proceed()` que "prossegue" com o método interceptado. Além disso, ao chamar o `proceed()` é necessário tratar uma

exceção que faremos através do `throws`. Veja o código:

```
public Object intercepta(InvocationContext context) throws Exception {

    long millis = System.currentTimeMillis();

    // chamada do método do dao
    context.proceed();

    System.out.println("Tempo gasto:  " + (System.currentTimeMillis() -
millis));
}
```

COPIAR CÓDIGO

Para finalizar o método falta retornar algo. Novamente entra o método `proceed()` no jogo. O retorno desse método representa o possível retorno do método do DAO. Vamos guardar o retorno em uma variável auxiliar e retornar no final do método:

```
public Object intercepta(InvocationContext context) throws Exception {

    long millis = System.currentTimeMillis();

    // chamada do método do dao
    Object o = context.proceed();

    System.out.println("Tempo gasto: " + (System.currentTimeMillis() - millis));

    return o;
}
```

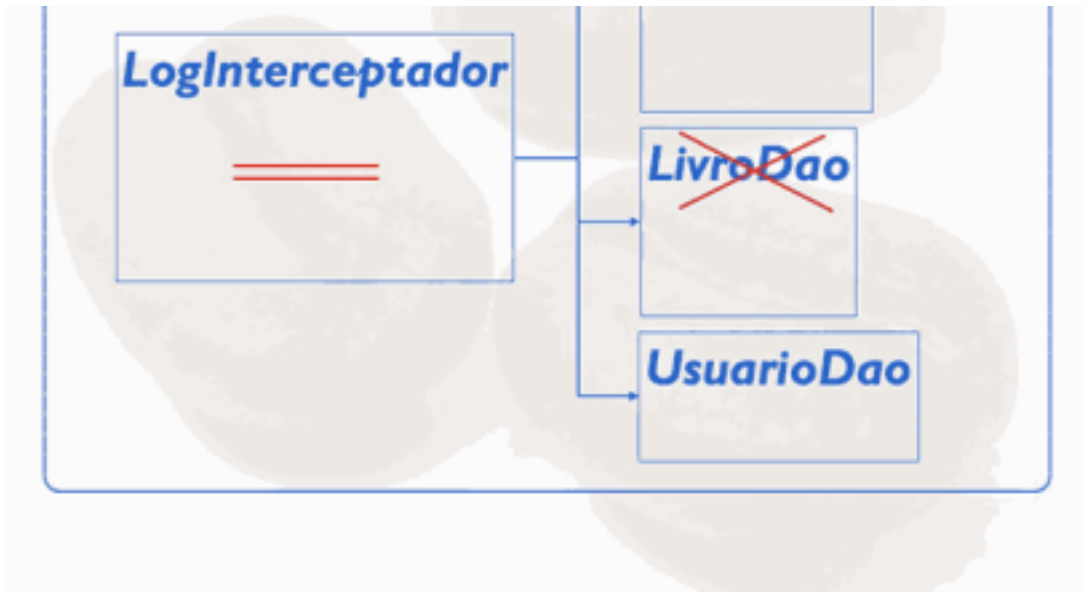
COPIAR CÓDIGO

Para terminar a implementação do interceptador falta uma configuração que deixe claro para o *EJB Container* que o método realmente intercepta o fluxo. Essa configuração é feita através de uma anotação que se chama `@AroundInvoke`:

```
@AroundInvoke
public Object intercepta(InvocationContext context) throws Exception {

    long millis = System.currentTimeMillis();

    // chamada do método do dao
    Object o = context.proceed();

    System.out.println("Tempo gasto: " + (System.currentTimeMillis() - millis));

    return o;
}
```

COPIAR CÓDIGO

Pronto, mas tem ainda um problema: quais são as classes e métodos que serão interceptados? Como o *EJB Container* sabe que queremos, por exemplo, interceptar as chamadas da classe `AutorDao` e não do `LivroDao`?



Isso também deve ser configurado na classe a ser interceptada, nesse caso a classe `AutorDao`, com a anotação `@Interceptors`. A anotação recebe um *array* de classes interceptadoras. Podemos usar a anotação `@Interceptors` em cima da classe ou em cima de um método especifico:

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
@Interceptors({LogInterceptador.class})
public class AutorDao {

    // código omitido
```

COPIAR CÓDIGO

Tudo pronto para testarmos! Vamos republicar a aplicação no servidor e acessa-la pela interface web. Após o *login* estamos na página inicial que usa o `AutorDao` já que carrega os autores para listar no *combobox*. Ou seja, já deve aparecer no console do Eclipse a saída do nosso `LogInterceptador`. Repare bem no fim a mensagem **Tempo gasto: 3**. Ótimo, aparentemente já está funcionando!

Vamos realizar uma vez o cadastro de autores e inserir um novo autor. Ao salvar o autor utilizamos o método `salva` da classe `AutorDao`, que novamente foi interceptado pelo `LogInterceptador`. No Eclipse, no console aparecerá também as novas mensagens.

Trabalhando com o InvocationContext

Ao analisar as mensagens de *log* percebemos um problema, não está claro qual método realmente está associado a mensagem **Tempo gasto**. Vamos melhorar isso alterando o interceptador. O `InvocationContext` possui vários métodos que ajudam a descobrir quais métodos realmente foram interceptados. Vamos aproveitar o objeto `context` para pegar o nome do método:

```
String metodo = context.getMethod().getName();
```

COPIAR CÓDIGO

E o nome da classe

```
String nomeClasse = context.getTarget().getClass().getSimpleName();
```

COPIAR CÓDIGO

E vamos imprimir o nome da classe e método:

```
System.out.println(nomeClasse + ", " + metodo);
```

COPIAR CÓDIGO

Segue uma vez o código completo:

```
String metodo = context.getMethod().getName();
String nomeClasse = context.getTarget().getClass().getSimpleName();
System.out.println(nomeClasse + ", " + metodo);
```

COPIAR CÓDIGO

Vamos reimplantar a aplicação e acessar pelo navegador. Depois do *login* já podemos verificar o console. Agora, antes da mensagem, aparecem a classe e nome do método interceptado. Muito mais fácil de entender. Faremos mais um teste com o cadastro de autores. Muito bem, aqui também aparecem as mensagem sempre com classe e método.

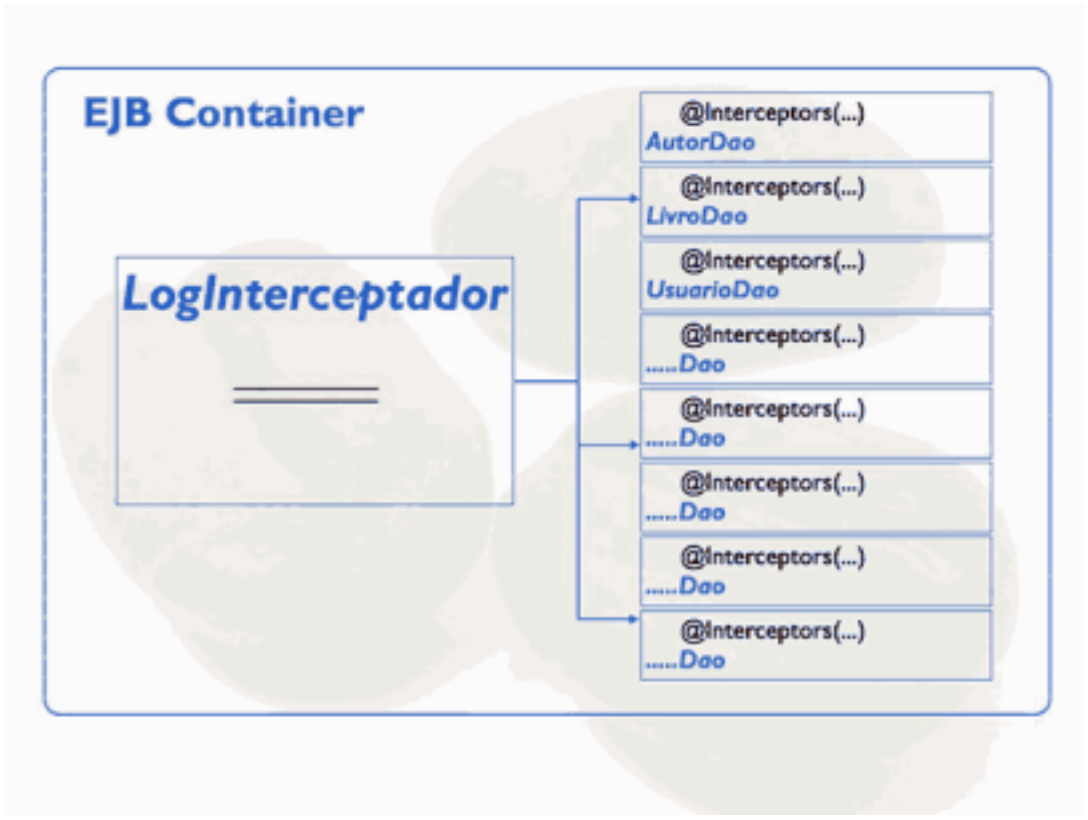
Interceptadores padrões

Por enquanto, o monitoramento está ligando para os métodos da classe `AutorDao`. Caso quiséssemos habilitar o interceptador na classe `LivroDao` também, bastaria repetir a anotação `@Interceptors(...)`:

```
@Stateless
@Interceptors({LogInterceptador.class})
public class LivroDao {
```

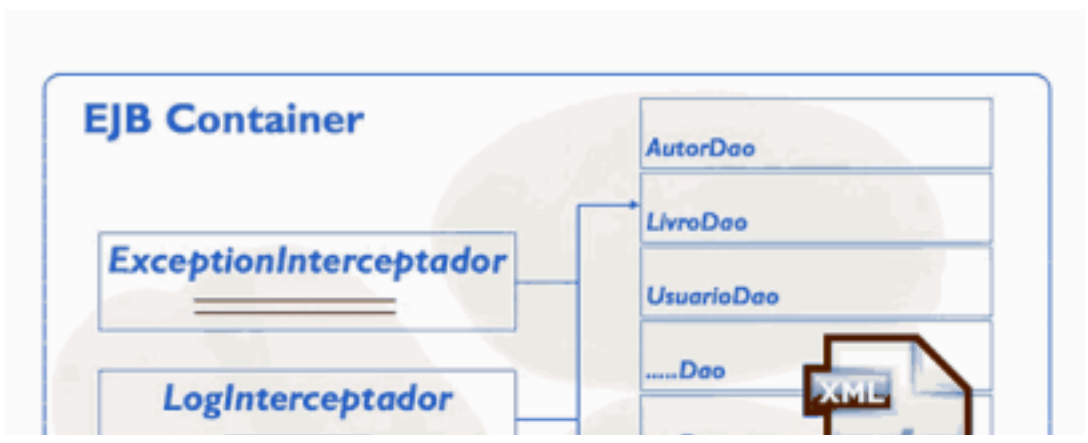
COPIAR CÓDIGO

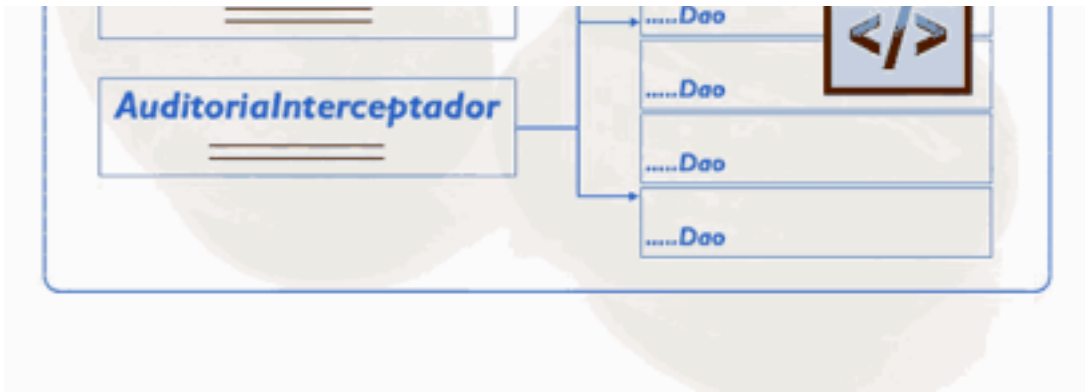
No entanto, isso gera um outro problema. Se existirem 200 DAOs, será preciso abrir todas essas classes para colocar a anotação.



A configuração ficará espalhada pelas classes! Para resolver isso há uma outra forma de associar o interceptador às classes DAOs, através de um XML.

Antes de continuar vamos comentar a anotação `@Interceptors`.





Como XML é trabalhoso, já preparamos ele e estamos disponibilizando na pasta **resources** nos *Downloads*. O arquivo deve se chamar `ejb-jar.xml` e ficar, como se trata de uma aplicação web, dentro da pasta `WEB-INF`. Vamos então copiar o arquivo para esta pasta.

Ao abrir o arquivo podemos ver que o mesmo possui dois elementos principais. O primeiro pode definir uma lista de interceptadores, aqui temos apenas um, o `LogInterceptador`. O segundo define onde os interceptadores são aplicados. Repare o elemento `<ejb-name>` que possui um asterisco. Isso significa que queremos associar o interceptador para todos os EJBs. Invés do asterisco podemos usar o nome do EJB, como por exemplo `AutorDao`.

Vamos salvar o arquivo e republicar a aplicação. Vamos fazer os mesmos testes e verificar se o monitoramento continua aparecendo no console do Eclipse. Ao voltar no Eclipse podemos ver que as mensagens foram impressas.

Vamos testar uma vez o monitoramento para um outro *EJB Session Bean*, agora o `AutorService`. Para isso copiaremos o elemento `interceptor-binding` e no `ejb-name` colocaremos `AutorService`. Repare que não recompilamos o código da classe, apenas alteramos a declaração no XML.

Novamente publicamos e acessamos a aplicação. Após o *login* e de navegar para o cadastro de autores, vamos voltar para o Eclipse. Podemos ver agora também as mensagens relacionadas à classe `AutorService`.

Então, um interceptador permite ligar e desligar um serviço com os *Session Beans*, sem alterar um *bean* especifico. Fizemos apenas um monitoramento, mas poderíamos implementar algo muito mais sofisticado com segurança ou cache. Repare também que seria possível implementar o gerenciamento da transação manualmente, através do `UserTransaction` no interceptador. Podemos injetar qualquer recurso dentro do interceptador. É bom pensar que o interceptador faz parte da funcionalidade do *Session Bean*, só que fica separado e serve para vários *beans*.