01 Ciclo de vida dos Session Beans

Transcrição

No último capítulo, vimos como preparar o ambiente e começar a usar os EJBs. Vimos o primeiro tipo de EJB, o Session Bean Stateless. Para configurar uma classe como Session Bean usamos a anotação `@Stateless` acima da classe. Essa configuração mínima já faz com que o EJB Container tome conta (ou controle) esse objeto.

Nas classes *Bean* usamos a anotação `@Inject` para receber o EJB pronto para ser utilizado. Chamamos de *Injeção de dependência* essa forma de receber uma instância.

Ao iniciar o servidor podemos ver no console os endereços dos 3 EJB Session Beans que já configuramos.

Nesse capítulo vamos aprender mais sobre *Ciclo da vida* dos Session Bean. Vamos entender como o EJB Container controla os objetos e como interferir nesse controle quando necessário.

Callbacks

Vamos voltar à classe `AutorDao` e adicionar um método que chamaremos `aposCriacao`. Nele faremos um simples *syso* para imprimir uma mensagem:

```
void aposCriacao() {
    System.out.println("AutorDao foi criado");
}
```

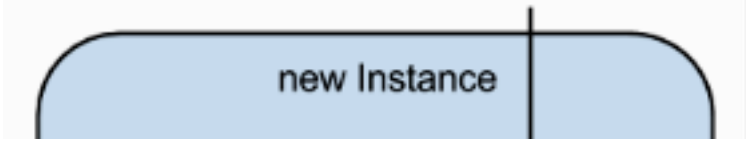
COPIAR CÓDIGO

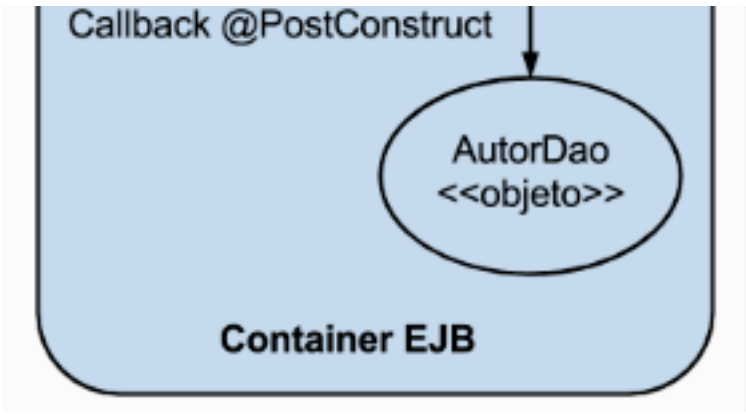
Este método não precisa ser público, pois ele não será chamado pela classe `AutorBean`. Basta anotá-lo com `@PostConstruct` e ele será chamado pelo próprio EJB Container:

```
@PostConstruct
void aposCriacao() {
    System.out.println("AutorDao foi criado");
}
```

COPIAR CÓDIGO

Assim que o Container cria e inicializa o Session Bean, o método `aposCriacao` é executado. Esse tipo de método ligado ao ciclo de vida do Session Bean também é chamado de *Callback*.





Vamos testar a funcionalidade e publicar a aplicação. Assim que o servidor recarregá-la, vamos limpar o console e chamá-la pela interface.

Ao acessar a URI pelo navegador e passar pela tela de login, podemos ver que o *combobox* com os autores está populado, ou seja, o *EJB Container* já criou o `AutorBean`. Isso fica claro no console do Eclipse. Repare que aparece a saída *AutorDao foi criado*. O EJB Container instanciou o Session Bean e chamou o método *callback*.

Thread safety

Vamos testar mais um pouco a aplicação e acessar pela interface web a página dos autores, navegando entre as abas. Muito bem. Voltando ao Eclipse, podemos ver no console que continua apenas uma saída do nosso método *callback*. Isso significa que apenas um Session Bean foi criado, já que o container sempre chama o callback na criação.

Vamos testar isso melhor e simular um pouco a execução lenta do método `salva no AutorDao`.

Primeiro colocaremos um `Syso` no inicio do método e um `Syso` no final do método para saber quando a execução começou e quando terminou.

Segundo, vamos travar a execução da *thread* atual usando o comando `Thread.sleep(..)`. No nosso exemplo, o *thread* atual vai dormir por 20 segundos. O método `sleep(..)` exige um tratamento de erro, por isso é preciso fazer um `try-catch` e podemos gerá-lo pelo Eclipse. Pronto.

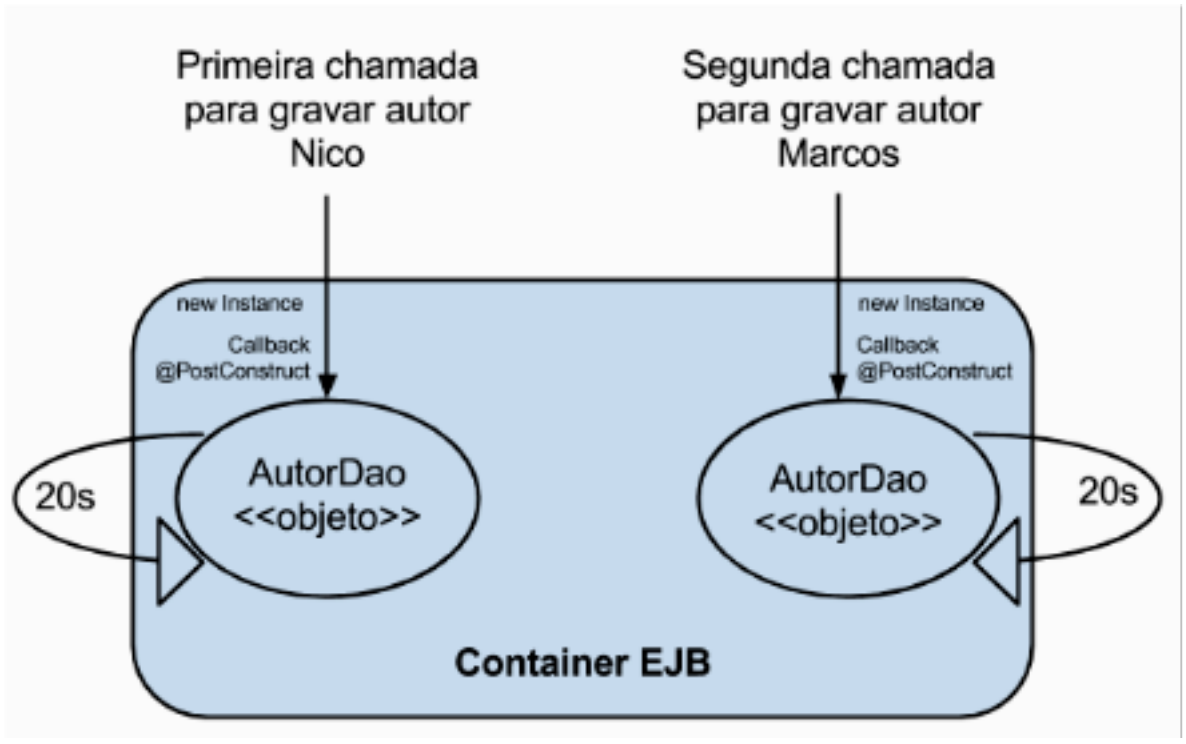
Depois da alteração, faremos o `publish` obrigatório para recarregar a aplicação. Depois de ter limpado o console, vamos acessar a aplicação web pelo navegador. Logo após login, vamos para a página de autores para cadastrar um autor.

Ao salvar, podemos perceber que o nome do autor não aparece imediatamente na lista de autores abaixo. Isso acontece, pois a thread para salvar o autor ainda está em execução. Travamos por 20 segundos. Repare no console que o `AutorDao` foi criado e o método `salva()`, que está sendo executado, não terminou ainda.

Vamos rapidamente abrir uma nova aba e recarregar a aplicação para cadastrar mais um autor. Ao salvar novamente a thread parou, mas podemos observar no console que mais um `AutorDao` foi criado, pois apareceu a saída do callback. Ou seja, como o primeiro objeto Session Bean estava em uso, o EJB Container decidiu criar mais um para atender a chamada. Só depois, quando os 20s passaram, aparece a última mensagem no console e consequentemente o autor é listado na

interface.

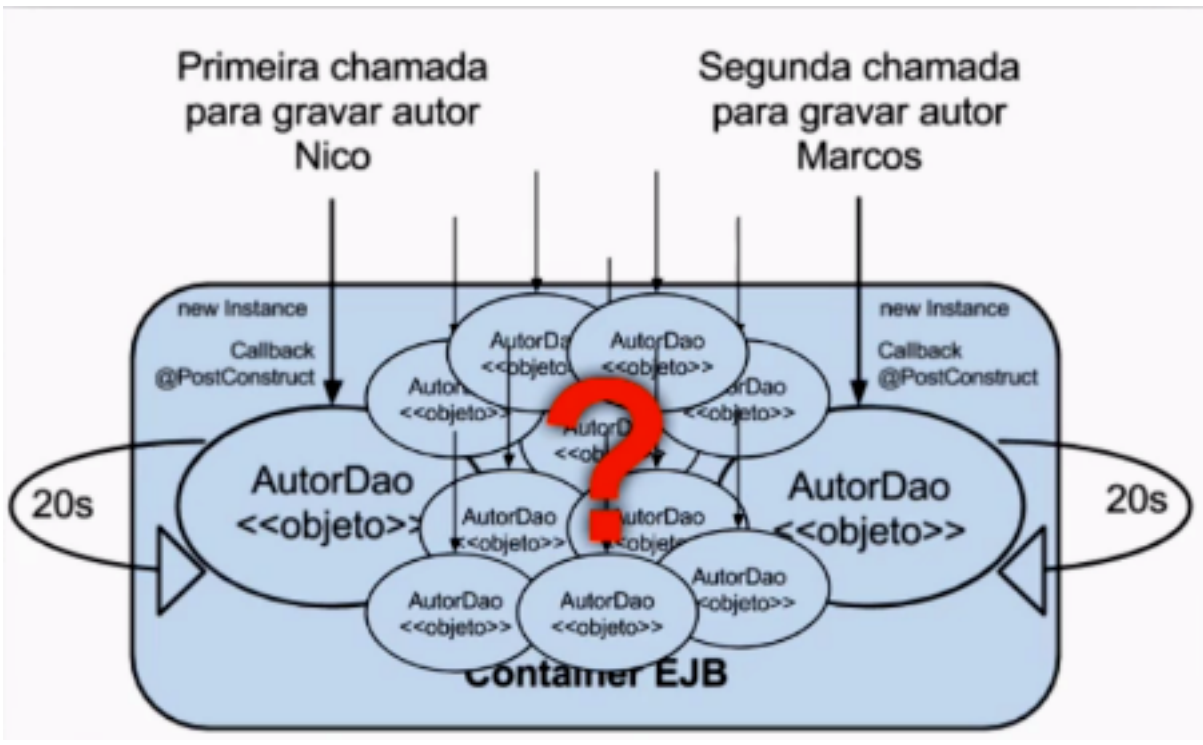
Esse pequeno exemplo mostrou que um Session Bean não é compartilhado entre *Threads*. Apenas uma thread pode usar o nosso `AutorDao` ao mesmo tempo. Um Session Bean é automaticamente *Thread safe*. *Thread safety* é um dos serviços que ganhamos de graça ao usarmos EJBs.



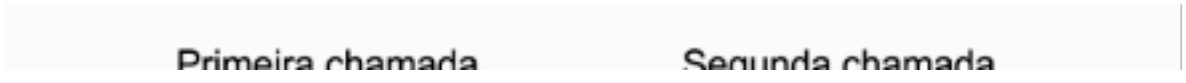
Pool de Objetos

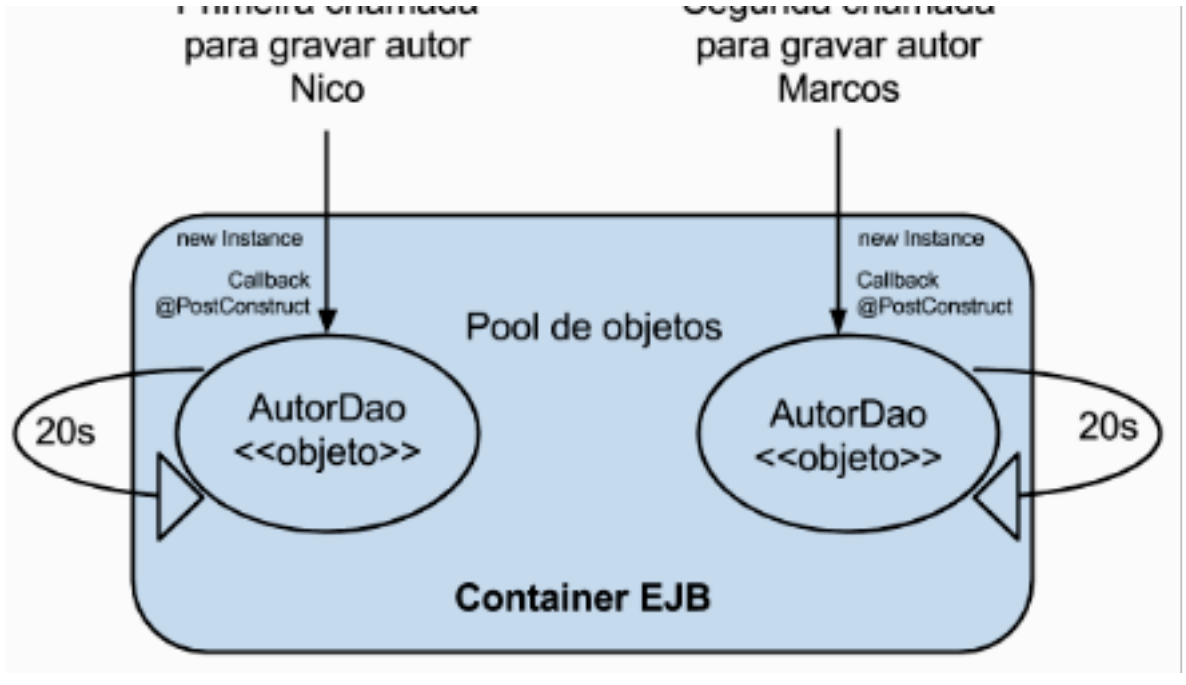
Vimos que o *EJB Container* criou um segundo objeto do tipo `AutorDao`, pois o primeiro estava sendo usado. *EJB Container* fez isso para melhorar o desempenho, já que o *Sesson Bean* não é compartilhado.

A pergunta é, quantos objetos o *EJB Container* serão criados? Se eu tiver 100 *threads* ao mesmo tempo, 100 objetos `AutorDao` serão criados em memória?



O *EJB Container* automaticamente fornece um *pool* de objetos que gerencia a quantidade do *Session Beans*. A configuração desse *pool* se encontra no arquivo de configuração do JBoss AS, ou seja, é totalmente específico.





Vamos abrir o arquivo `standalone.xml` da pasta de `standalone/configuration`. Nele procuraremos o elemento `<pools>`. Dentro desse elemento, encontraremos a configuração do *pool* para *session beans*. Repare o atributo `max-pool-size` que define a quantidade de objetos no *pool*. No nosso caso são 20 instancias.

```
<pools>
        <bean-instance-pools>
              <strict-max-pool name="slsb-strict-max-pool" max-
pool-size="20" instance-acquisition-timeout="5" instance-
acquisition-timeout-unit="MINUTES"/>
              <!-- outros elementos omitidos -->
        </bean-instance-pools>
</pools>
```

COPIAR CÓDIGO

Vamos fazer um teste e configurar o *pool* com apenas 1 instancia. Basta colocar o valor no atributo `max-pool-size`. Como alteramos o arquivo principal de configuração é necessário reiniciar o servidor. No final limparemos o console para acompanhar melhor o trabalho do *EJB Container*.

```
<pools>
        <bean-instance-pools>
              <strict-max-pool name="slsb-strict-max-pool" max-
pool-size="1" instance-acquisition-timeout="5" instance-acquisition-
timeout-unit="MINUTES"/>
              <!-- outros elementos omitidos -->
        </bean-instance-pools>
</pools>
```

COPIAR CÓDIGO

Vamos fazer o mesmo teste, ou seja, acessar a nossa aplicação através de duas abas, simulando duas ações de usuário executado ao mesmo tempo. Após ter feito o login, vamos para a aba autores. Não podemos esquecer de atualizar a outra aba também. Quando estiver pronto podemos salvar o primeiro autor.

No console, aparecem as mensagens do *callback* e do método `salva(...)`. Até aqui tudo igual. Vamos para a segunda aba para salvar um outro autor. Ao salvar e analisar o console NÃO aparece a mensagem do *callback*. Isso faz sentido, pois configuramos para que exista apenas um objeto `AutorDao`. Enquanto a primeira ação não finalizar, não é possível acessar esse objeto, pois, como já mencionamos, os *Session Beans* são *thread safe*. Repare que o console mostra que na execução foi um depois do outro, já que existe apenas um objeto do tipo `AutorDao` em memória.

Repare que a configuração do tamanho do *pool* influencia diretamente na escalabilidade da aplicação. Ter apenas um objeto `AutorDao` em memória significa que só podemos atender um chamado por vez. Por isso faz sentido, para objeto DAO, aumentar a quantidade de objetos no *pool*.

Por fim, vamos desfazer a alteração no arquivo `standalone.xml` e configurar novamente 20 instancias no *pool*. Como mexemos no xml, não podemos esquecer de reiniciar o servidor. No método `salva(..)` também vamos comentar o código que mandou o `Thread` atual dormir. Pronto.

Singleton Beans

Já aprendemos como configurar as classes DAOs, agora vamos atacar a classe `Banco`. O `Banco` ainda não é um *EJB Session Bean*. Podemos facilmente mudar isso e colocar a anotação `@Stateless` em cima da classe como vimos nos exemplos anteriores

```
@Stateless
public class Banco {
```

COPIAR CÓDIGO

E, por exemplo, no `AutorDao`, vamos injetar o banco. Para tal, não devemos instanciar o `Banco` e sim, usar a anotação `@Inject`:

```
@Inject
private Banco banco;
```

COPIAR CÓDIGO

Voltando para classe `Banco`, vimos, no exemplo anterior, que existe um *pool* de objetos para *Session Beans*. Ou seja, como o `Banco` é um *Session Bean*, teremos, no máximo, 20 instancias em memória.

Nesse caso pode surgir a pergunta, faz sentido ter todas essas instancias dessa classe? Claro que não! Apesar do fato de que essa classe só existe para simular um banco de dados, não faz sentido nenhum ter mais do que um objeto dessa classe. É preciso ter apenas um único objeto para simular o banco.

Felizmente podemos configurar isso sem mexer na configuração XML do JBoss AS. Basta usar a anotação `@Singleton`:

```
@Singleton  // do package javax.ejb
public class Banco {
```

COPIAR CÓDIGO

Para ter certeza que não existem mais instâncias, vamos fazer o mesmo teste. Na classe `Banco` adicionaremos um método de *callback* usando a anotação `@PostConstruct`:

```
@PostConstruct
void aposCriacao() {
    System.out.println("acabou de criar o Banco");
}
```

COPIAR CÓDIGO

Se tudo estiver configurado, podemos iniciar o JBoss AS para testar a aplicação com o novo *Singleton Session Bean*. Na saída do console, podemos ver que o *EJB Container* já entrou e carregou o `Banco`. Repare que existe uma saída parecida com a dos outros *Session Beans*.

Só falta limpar o console e acessar a interface web. Após o login, o conteúdo de ambos os *callbacks* é impresso no console; aquele do `AutorDao` e esse novo do `Banco`.

Vamos cadastrar uma vez um Autor pela interface para verificar se realmente existe uma única instancia do Banco. Como esperado, o console indica que o *callback* foi só uma vez.

Eager Initialization

Session Beans do tipo *Singleton* são tipicamente usados para inicializar alguma configuração ou agendar algum serviço. Fazer isso só faz sentido no inicio da aplicação, ou seja, quando o JBoss AS carrega a aplicação e já queremos que o *Session Bean* seja criado para carregar todas as configurações.

Por padrão um EJB é carregado sob demanda (*lazy*), mas através da anotação `@Startup` podemos definir que queremos usar o *Singleton Bean* desde o início da aplicação:

```
@Singleton //do package javax.ejb
@Startup
public class Banco {
```

COPIAR CÓDIGO

Para testar, vamos recarregar a aplicação, ou seja, *Full Publish* aba *Servers* do Eclipse. Após ter carregado a aplicação, aparece no console a saída do *callback*. O Banco já é criado e inicializado pelo EJB Container.

Aquela inicialização com `@Startup` também é chamada eager initialization e a testaremos nos exercícios.