argonavis
tecnologia e arte

EJB
enterprise javabeans

JAVA EE 7

Helder da Rocha

O texto foi elaborado como material de apoio para treinamentos especializados em linguagem Java e explora assuntos detalhados nas especificações e documentações oficiais sobre o tema, utilizadas como principais fontes. A autoria deste texto é de inteira responsabilidade do seu autor, que o escreveu independentemente com finalidade educativa e não tem qualquer relação com a Oracle.

O código-fonte relacionado aos tópicos abordados neste material estão em:

`github.com/helderdarocha/javaee7-course`
`github.com/helderdarocha/CursoJavaEE_Exercicios`
`github.com/helderdarocha/ExercicioMinicursoJMS`
`github.com/helderdarocha/JavaEE7SecurityExamples`

www.argonavis.com.br

R672p Rocha, Helder Lima Santos da, 1968-

Programação de aplicações Java EE usando Glassfish e WildFly.

360p. 21cm x 29.7cm. PDF.

Documento criado em 16 de outubro de 2013.

Atualizado e ampliado entre setembro e dezembro de 2016.

Volumes (independentes): *1: Introdução, 2: Servlets, 3: CDI, 4: JPA, 5: EJB, 6: SOAP, 7: REST, 8: JSF, 9: JMS, 10: Segurança, 11: Exercícios*.

1. Java (*Linguagem de programação de computadores*). 2. Java EE (*Linguagem de programação de computadores*). 3. Computação distribuída (*Ciência da Computação*). I. Título.

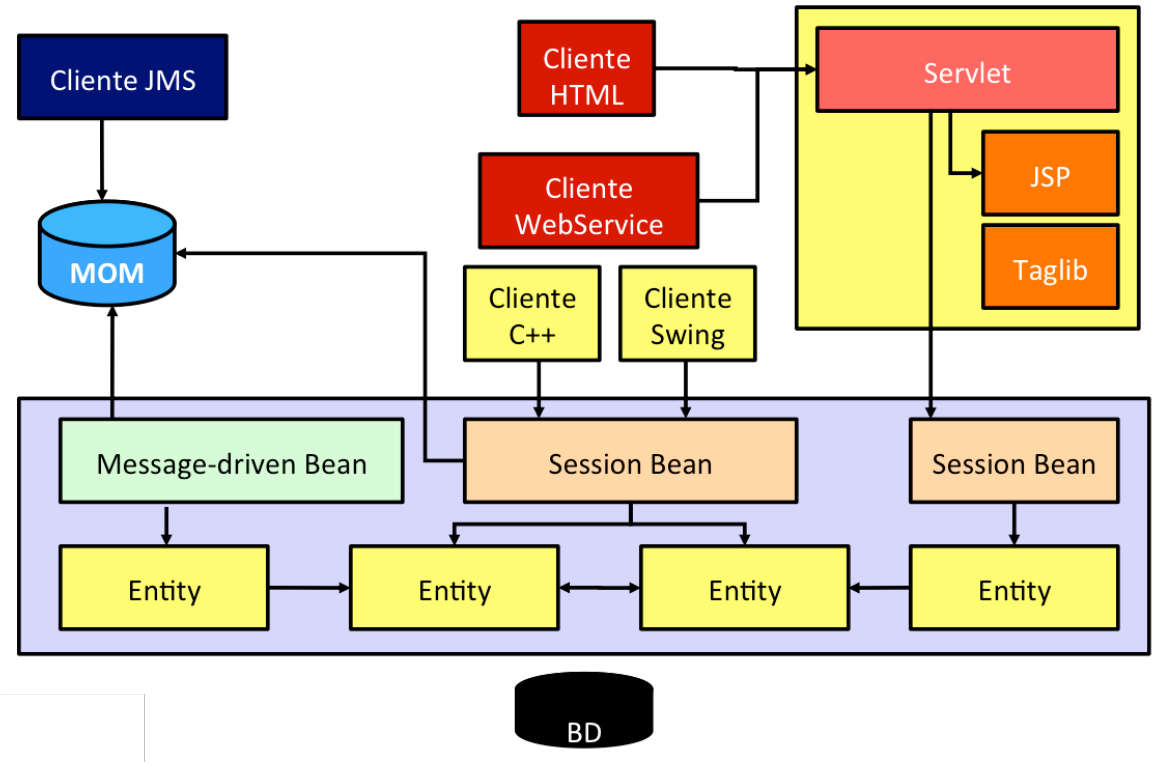
CDD 005.13'3

Capítulo 5: Enterprise JavaBeans (EJB)

1 Enterprise JavaBeans

Enterprise JavaBeans são componentes que encapsulam a lógica de negócios de uma aplicação. Implementam serviços síncronos e assíncronos, podem exportar uma interface para clientes locais, mas também para clientes remotos usando protocolos de objetos remotos (IIOP) ou Web Services (SOAP). Instâncias de EJBs têm o ciclo de vida controlado em tempo de execução pelo container no qual ele está instalado, que também disponibiliza o acesso a serviços de segurança, transações, agendamento, concorrência, sincronização de estado, distribuição e outros de forma transparente.

O desenho abaixo situa os Enterprise JavaBeans (session beans e message-driven beans) dentre os outros componentes de uma aplicação Java EE



Enterprise JavaBeans são *JavaBeans*, ou POJOs – um objeto Java qualquer contendo um construtor default (sem argumentos), atributos encapsulados (*private*) e acessíveis através de métodos get/set.

Existem dois tipos de EJBs:

- Session beans
- Message-driven beans

2 Session Beans

Session beans são componentes de negócios que exportam uma interface de serviços. Eles representam uma sessão de comunicação com um cliente (que não necessariamente corresponde a uma sessão Web).

Um session bean deve ser criado quando houver interesse em fornecer uma *interface de serviços*. Esta interface pode ser local (acessível dentro da mesma aplicação) ou remota (acessível de outras aplicações ou mesmo de outras máquinas). É possível ainda exportar uma interface para clientes SOAP. A interface é definida por uma coleção de métodos.

Existem diversas formas de declarar uma interface para um session bean:

- *Interface local* usando *no-interface view*: o bean é acessível localmente, apenas para clientes da mesma aplicação. A interface de serviços consiste de todos os métodos do bean (inclusive os herdados de suas superclasses).
- *Interface local* declarada usando *@Local*: o bean implementa uma ou mais interfaces Java que contém todos os métodos de sua interface de serviços que podem ser acessadas localmente por clientes da mesma aplicação.
- *Interface remota* (para clientes IIOP) declarada usando *@Remote*: o bean implementa uma ou mais interfaces que contém todos os métodos de sua interface de serviços que acessíveis por clientes locais e clientes em outras aplicações, servidores ou máquinas.
- *Interface remota* (para clientes SOAP) declarada usando *@WebService*: o bean implementa uma interface Java que declara os métodos exportados ou marca-os com *@WebMethod* no próprio bean.

2.1 Tipos de session beans

Existem três tipos de session beans:

- *Stateful Session Beans* – usado para comunicação que precisa preservar o estado do cliente entre chamadas de métodos do bean.
- *Stateless Session Beans* – usado para comunicação que não precisa manter o estado do cliente.
- *Singleton Session Beans* – usado para comunicação que precisa compartilhar o estado entre diversos clientes.

2.1.1 Stateful Session Beans

Stateful session beans modelam diálogos que consistem de uma ou mais requisições, onde certas requisições podem depender do estado de requisições anteriores. Esse estado é guardado nas variáveis de instância do bean.

A forma mais simples de criar um stateful session bean é usando o estilo *no-interface view*, anotando a classe com @Stateful:

```
@Stateful
public class CestaDeCompras { ... }
```

Isto ira exportar uma interface local que consiste de todos os métodos de instância da classe (inclusive os que foram herdados de superclasses). Qualquer outro componente da aplicação que injetar o bean poderá chamar qualquer um desses métodos.

Como em stateful session beans o estado do cliente é mantido por varias chamadas, poderá chegar o momento em que o cliente decida que o diálogo foi finalizado. Ele poderá então desejar remover o bean e encerrar sua sessão. Para isto é necessário anotar um método da interface com *@Remove*:

```
@Remove public void remover() {}
```

O método não precisa conter nenhuma instrução. Se o bean também participa de um escopo CDI, ele só poderá ser removido pelo cliente se o seu escopo for *@Dependent* (default). Caso ele tenha outro escopo, ele deve deixar que o container remova o bean quando o escopo terminar.

O container removerá o bean depois que o método anotado com @Remove terminar.

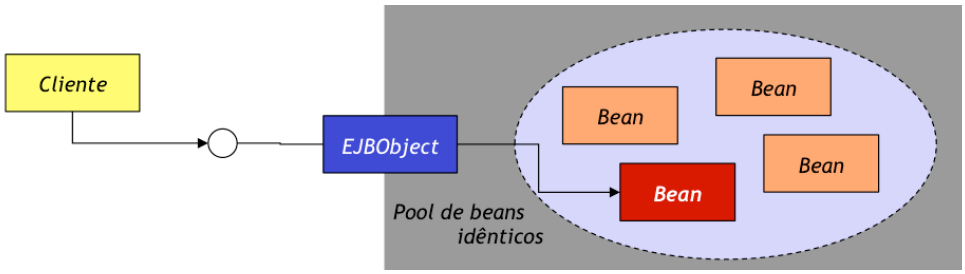
Stateful session beans mantém estado para cada cliente, portanto o container não pode fazer o mesmo tipo de pooling que faz com stateless session beans. Os beans não são considerados equivalentes. Se há mais clientes que beans no pool, mas grande parte permanece inativa por certos períodos, o container usa um mecanismo de ativação e passivação para otimizar o uso de recursos. Ele armazena o estado (não transiente) de beans menos acessados liberando a instância para reuso por um novo cliente. Se o cliente antigo faz uma nova chamada no bean passivo, ele é automaticamente ativado. Essa estratégia aumenta a capacidade de atender clientes.

Por exemplo, se num pool de 5 beans, todos estão ativos e em comunicação com seus clientes, aparecer um sexto cliente, o estado do bean que foi usado há mais tempo é serializado e gravado em meio persistente. Se houver um método anotado com *@PrePassivate* ele e chamado antes da passivação.

Se esse cliente está continuando um diálogo iniciado anteriormente, o estado anterior do bean que ele usou é recuperado. Se houver um método anotado como *@PostActivate* ele é chamado após a ativação.

2.1.2 Stateless Session Beans

Stateless session beans modelam diálogos que consistem de apenas uma requisição. Como não mantém estado do diálogo, todas as instâncias da mesma classe são *equivalentes e indistinguíveis*. Qualquer instância disponível de um session bean pode servir a qualquer cliente. Session Beans podem ser guardados em pool, reutilizados e passados de um cliente para outro em cada chamada.



A forma mais simples de criar um stateful session bean é usando o estilo *no-interface view*, anotando a classe com *@Stateless*:

```
@Stateless
public class VitrineVirtual { ... }
```

2.1.3 Singleton Session Beans

Singleton session beans foram projetados para compartilhar informação entre beans. São instanciados uma única vez por aplicação, e têm escopo de aplicação. Por terem estado compartilhado, precisam lidar com questões relacionadas ao acesso concorrente. Só pode haver uma instância do bean por aplicação (mas se uma aplicação for distribuída por várias máquinas virtuais, haverá uma instância em cada JVM).

A forma mais simples de criar um stateful session bean é usando o estilo *no-interface view*, anotando a classe com *@Singleton*:

```
@Singleton
public class ServicosGlobais { ... }
```

É comum instruir o container a carregar o bean durante a inicialização da aplicação anotando-o com @Startup:

```
@Startup
@Singleton
public class ServicosGlobais { ... }
```

Às vezes um singleton depende da inicialização prévia de outros singletons. Isto pode ser configurado usando *@DependsOn*:

```
@DependsOn("ServicosGlobais")
@Singleton
public class EstatisticasDeAcesso { ... }
```

O acesso concorrente a um Singleton session bean é controlado pelo container (default) e todos os seus métodos tem (por default) acesso exclusivo. Isto significa que apenas um cliente de cada vez pode ter acesso ao método. Esse comportamento pode ser alterado para maior eficiência (ex: permitir que métodos que não alteram o estado sejam acessados simultaneamente.)

2.2 Interfaces e acesso

2.2.1 Interfaces para clientes locais

A forma mais simples de exportar a interface de um SessioBean é usar a configuração default (*no-interface view*), que considera *todos* os métodos do bean (inclusive os herdados) como parte da interface.

A forma mais simples (e recomendada) de obter acesso a este bean em uma classe cliente da mesma aplicação é usando injeção de dependências (ou CDI):

```
@EJB
BibliotecaBean bean;
```

A estratégia no-interface view expõe *todos os métodos*, o que pode não ser desejável se a classe contém métodos que não devem ser expostos. Pode-se ter mais controle criando uma interface Java anotada com @Local e contendo apenas os métodos que deverão ser exportados.

```
@Local
public interface Biblioteca {
    void emprestar(Livro livro);
    void devolver(Livro livro);
}
```

O bean só precisa implementar a interface:

```
@Stateless
public class BibliotecaBean implements Biblioteca { ... }
```

Os clientes de um bean @Local devem pertencer à mesma aplicação e podem usar CDI ou injeção de dependências (com @EJB) para injetar a *interface* (não o bean):

```
@SessionScoped
public class Administrador {
```

```
    @EJB Biblioteca biblioteca;

    public void recebidos(List<Livro>) {
        for(Livro m : livros) {
            biblioteca.devolver(m);
        }
    }
...
}
```

Ou fazer o lookup JNDI (também pela *interface*):

```
@SessionScoped
public class Administrador {
    Biblioteca biblioteca;

    @PostConstruct
    public void init() {
        Context ctx = new InicialContext();
         biblioteca = (Biblioteca)ctx.lookup("java:app/Biblioteca");
    }
    ...
}
```

Se a interface usada não puder ser anotada como @Local, ainda é possível declará-la como tal se ela for implementada pelo bean e passada como parâmetro de uma anotaçao @Local na classe do bean:

```
@Local(Biblioteca.class)
public class BibliotecaBean implements Biblioteca {}
```

2.2.2 Interfaces para clientes remotos

Clientes remotos precisam ser declarados através de uma interface de serviços. Se a interface de serviços for anotada com *@Remote*, o bean só precisa implementá-la.

```
@Remote
public interface BibliotecaRemote {
    void consultar(Livro livro);
    void reservar(Livro livro);
}

@Stateless
public class BibliotecaBean implements BibliotecaRemote { ... }
```

Se a interface *não* for anotada com *@Remote*, o bean ainda pode usá-la passando seu nome como argumento para *@Remote*:

```
@Remote(BibliotecaRemote.class)
public class BibliotecaBean implements BibliotecaRemote, OutraInterface {}
```

Dentro de um container EJB é possível obter a referência para um bean remoto via injeção de dependências (chamando pela interface, sempre):

```
@EJB BibliotecaRemote biblioteca;
```

Dependendo da configuração e do servidor, às vezes é possível usar injeção de dependências em containers localizados em máquinas diferentes, mas nesses casos o mais garantido é usar JNDI:

```
Context ctx = new InicialContext();
BibliotecaRemote biblioteca =
                (BibliotecaRemote)ctx.lookup("java:global/libraryapp/Biblioteca");
```

Em clientes RMI/IIOP que não estão usando um container ainda é possível obter acesso a um EJB via acesso ao JNDI global proprietário do servidor. Em alguns servidores (ex: WebSphere 7) o proxy obtido via JNDI é um objeto CORBA que precisa ser convertido. Nesse caso, é recomendado usar o método *PortableRemoteObject.narrow()* para converter o proxy do objeto:

```
java.lang.Object stub = ctx.lookup("java:app/BibliotecaRemote ");
BibliotecaRemote biblioteca = (BibliotecaRemote)
    javax.rmi.PortableRemoteObject.narrow(stub, BibliotecaRemote.class);
```

Chamadas remotas tem uma natureza diferente de chamadas locais. Os parâmetros dos métodos e os tipos de retorno são sempre passados por valor (são proxies de rede), portanto não é possível fazer operações que alteram propriedades de um bean remoto por referência (é necessário alterar o bean localmente e depois sincronizar).

Proxies de beans remotos e seus parâmetros precisam ser transferidos pela rede. É geralmente mais eficiente transferir objetos maiores em poucas chamadas, portanto interfaces remotas devem usar o padrão Data-Transfer Objects (DTOs) e transferir objetos de baixa granularidade, para reduzir o número de chamadas.

3 Message Driven Beans (MDB)

Message-driven beans (MDB) são processadores assíncronos. São usados para modelar eventos, notificações e outras tarefas assíncronas. Um MDB não possui uma interface e não pode ser chamado diretamente por um cliente. Mas um MDB pode agir como cliente e chamar outros beans, iniciar contextos transacionais e injetar e usar serviços disponíveis à aplicação.

MDBs são listeners de mensageria. São geralmente usados para processar mensagens JMS e implementam a interface *MessageListener*. São anotados com *@MessageDriven* informando o nome da fila à qual o bean está registrado como listener. A menos que seja

configurado com um filtro de mensagens, todas as mensagens recebidas na fila serão recebidas pelo metodo *onMessage()* do bean, que poderá processá-las.

```
@MessageDriven(mappedName="pagamentos")
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
             // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```

O atributo *mappedName* é uma forma *portável* de informar o nome da fila, mas de acordo com a especificação é de *implementação opcional*. Funciona na implementação de referência (Glassfish) mas pode não funcionar em outros servidores. Nesses casos a fila precisa ser informada usando propriedades de ativação (que podem variar entre fabricantes).

A propriedade *activationConfig* de *@MessageListener* recebe uma lista propriedades de configuração em anotações *@ActivationConfigProperty*. No WildFly/JBoss e WebSphere é obrigatório usar a propriedade *destination*, que informa a fila ao qual o bean deve ser registrado:

```
@MessageDriven(
  activationConfig={
    @ActivationConfigProperty(propertyName="destination",
                              propertyValue="pagamentos")}
)
public class ProcessadorCartaoDeCredito implements MessageListener {
    @Override
    public void onMessage(Message message) {
        try {
             // processa a mensagem
        } catch (JMSException ex) { ... }
    }
}
```

Existem outras propriedades que podem ser configuradas. Uma delas é *messageSelector* que permite filtrar as mensagens que serão recebidas pelo bean. O filtro atua sobre os cabeçalhos e propriedades da mensagem. Cabeçalhos são gerados automaticamente pelo sistema, e incluem dados como data de validade, id da mensagem, etc. Propriedades são definidas pelo remetente da mensagem, que pode gravar uma propriedade em uma mensagem JMS usando:

```
mensagem.setIntProperty("quantidade", 10);
```

O MDB pode, por exemplo, ser configurado para não receber mensagens que não tem essa propriedade ou que tem com valor menor, usando:

```
@MessageDriven(
  activationConfig={ ...,
    @ActivationConfigProperty(
        propertyName="messageSelector",
        propertyValue="quantidade is not null and quantidade >= 10")}
)
public class ProcessadorCartaoDeCredito implements MessageListener {…}
```

3.1 Produtor JMS

Produtores de mensagens são clientes que enviam mensagens a um destino. Um produtor JMS pode ser qualquer componente Java EE ou até mesmo aplicações cliente standalone que tenham como obter acesso a conexões e destinos no servidor. Se estiver dentro do container, o produtor pode injetar a conexão e destino (fila) para onde enviará mensagens:

```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;
@Resource(mappedName="jms/Queue")
private static Queue queue;
```

De clientes externos, ele pode obter proxies para esses objetos através de JNDI:

```
Context ctx = new InitialContext();
ConnectionFactory factory = (ConnectionFactory) ctx.lookup("jms/ConnectionFactory");
Queue queue = (Queue) ctx.lookup("jms/Queue");
```

MDBs também podem injetar Session Beans usando @EJB, outros serviços usando @Resource, e qualquer objeto ou serviço injetável (usando @Inject) via CDI.

Obtidos os objetos, mensagens podem ser enviadas usando os métodos do JMS:

```
JMSContext ctx = connectionFactory.createContext();
context.createProducer().send(queue, "Texto contido no corpo da mensagem");
```

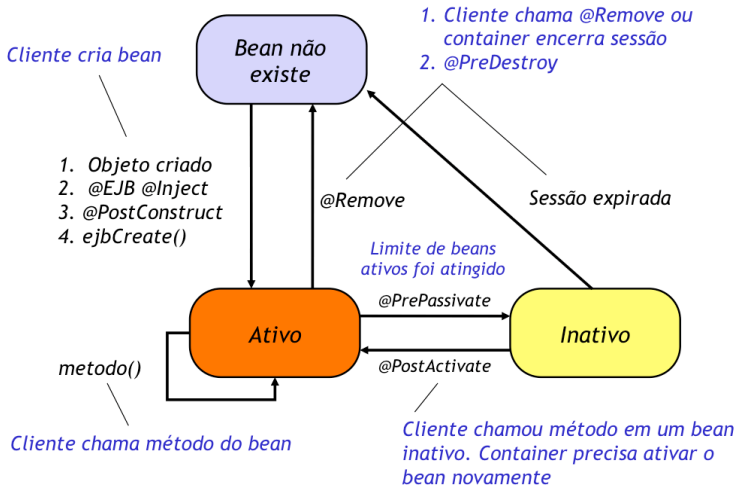
Usando CDI é possivel injetar o contexto JMS em vez de obter um ConnectionFactory:

```
@Inject @JMSConnectionFactory("jms/ConnectionFactory")
private JMSContext ctx2;
```

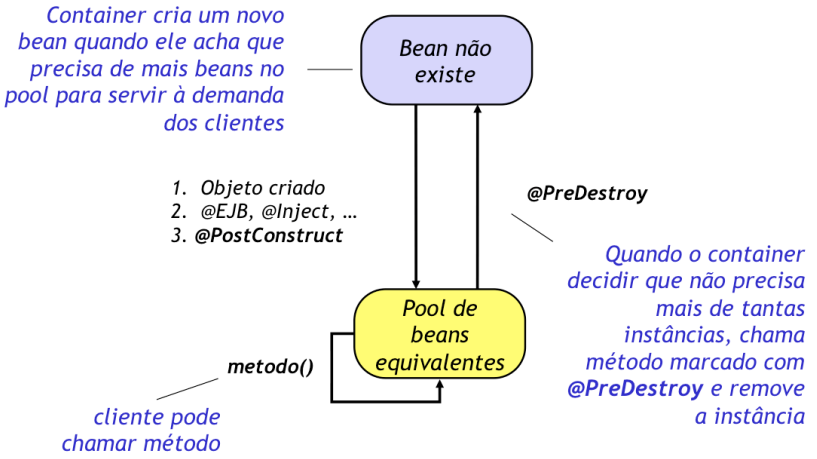
4 Ciclo de vida

Os diagramas abaixo ilustram os ciclos de vida de cada tipo de Enterprise JavaBean.

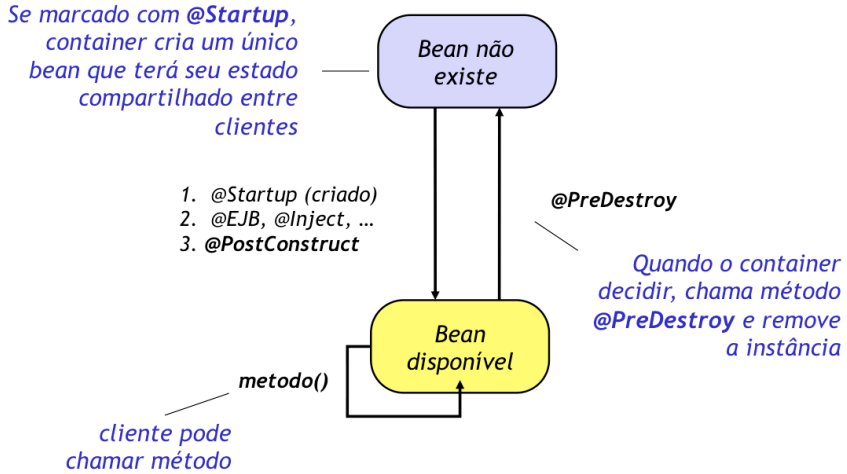
4.1 Stateful Session Bean



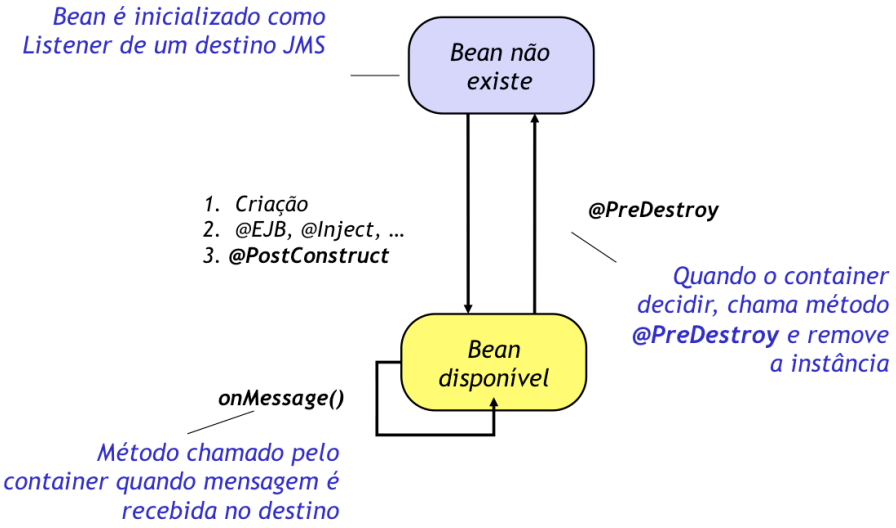
4.2 Stateless Session Bean



4.3 Singleton Session Bean



4.4 Message-driven Bean



4.5 Callbacks

Existem cinco anotações EJB para marcar métodos de callback, que são chamados automaticamente em eventos do ciclo de vida de um bean. Quatro delas são usadas em session beans:

@PostConstruct é usada para anotar um método que deve ser chamado após a execução do construtor default do bean e injeção das dependências, e antes que qualquer método da interface do bean seja chamado:

```
@Stateless
public class Biblioteca {

    @Inject ServicoIsbn servico;

    @PostConstruct
    private void inicializar() {
        servico.login();
    }
    ...
}
```

@PreDestroy é usada para anotar um método que deve ser chamado antes do bean ser removido. O bean pode ser removido explicitamente pelo cliente (ex: através de um método anotado com @Remove em stateful session beans) ou pelo container (a qualquer momento para stateless beans, e na finalização da aplicação para singletons).

```
@Stateless
public class Biblioteca {
    ...
    @PreDestroy
```

```
    private void finalizar() {
        servico.logout();
    }
    ...
}
```

@PrePassivate é usada apenas em stateful session beans e chamado antes que o bean entre no estado passivo.

```
@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PrePassivate
    private void passivar(InvocationContext ctx) {
        cliente.liberarCodigoAcesso();
    }
    ...
}
```

@PostActivate é usado apenas em stateful session beans e chamado depois que o bean é reativado do seu estado passivo.

```
@Stateful
public class CestaDeCompras implements Serializable {
    private transient String codigoAcesso;
    @Inject private Cliente cliente;

    @PostActivate
    private void ativar(InvocationContext ctx) {
        codigoAcesso = cliente.obterNovoCodigoAcesso();
    }
    ...
}
```

@PostConstruct e *@PreDestroy* também podem ser usados em *MessageDrivenBeans*. A quinta anotação, *@AroundConstruct*, é rara e usada apenas em interceptadores (tratados em outro capítulo). Métodos marcados com *@PrePassivate* ou *@PostActivate* em beans que não são stateful session beans são ignorados.

5 Acesso via JNDI

Componentes geralmente são injetados pelos clientes que os acessam de dentro de uma aplicação. Mas, se for necessário, eles também podem ser carregados diretamente através de JNDI. O acesso JNDI é necessário para acesso a componentes remotos entre aplicações. Servidores diferentes podem estipular formas diferentes para acessar serviços via JNDI, mas a

especificação Java EE define três formas padrão de acesso JNDI que podem ser usadas independente do servidor: *global*, no contexto de uma *aplicação* (se o bean estiver em modulo empacotado em um EAR), e no contexto de um *módulo*.

Um nome JNDI global tem a forma mínima:

```
java:global/aplicacao/bean
```

para beans armazenados em EJB JARs ou WARs. A aplicação é (por default) o nome do arquivo sem a extensão (aplicacao.war ou aplicacao.jar)

Se o WAR ou EJB JAR estiver dentro de um arquivo EAR, ele é considerado um módulo da aplicação que é representada pelo EAR. Nesse caso, é preciso incluir o EAR no caminho global:

java:global/*aplicacao*/*modulo*/*bean*

Onde o nome da aplicação (geralmente) é o arquivo EAR sem a extensão (ex: aplicacao.ear) e os nomes dos módulos, se não forem o nome do arquivo (default), estão definidos no /META-INF/application.xml do EAR.

Se o bean tiver mais de uma interface e houver necessidade de identificar especificamente uma delas, o nome (qualificado) da interface pode ser incluído depois do nome do bean, separado por uma exclamação:

java:global/*aplicacao*/*modulo*/*bean*!*pacote.Interface*

Se o acesso ao bean ocorrer entre módulos instaladas na mesma aplicação (EAR), pode-se usar o acesso no escopo do container com prefixo java:app:

java:app/*modulo*/*bean*

Se a comunicação ocorrer no contexto do módulo, ou se a aplicação não estiver empacotada em um EAR, pode-se usar qualquer uma das duas formas:

java:app/*bean*
java:module/*bean*

Por exemplo, se um bean @Remote chamado LivroBean é empacotado em um WAR biblioteca.war. Ele pode ser chamado localmente (ex: por um managed bean ou servlet no mesmo WAR) usando:

```
public class LivroClientServlet extends HttpServlet {
    public void init() {
        try {
            Context ctx = new InitialContext(); // inicialização do JNDI
            LivroBean bean = (LivroBean)ctx.lookup("java:app/LivroBean");
              bean.inicializar();
            ...
```

E por um cliente remoto usando:

```
public class LivroRemoteTest extends TestCase {
    @Test
    public void testLookup() {
        Context ctx = new InitialContext();
        LivroBean bean = (LivroBean)ctx.lookup("java:global/biblioteca/LivroBean");
            ...
```

Mas para o primeiro caso, que executa dentro do container, talvez seja mais prático simplesmente injetá-lo usando @EJB (ou @Inject, se configurado com CDI):

```
public class LivroClientServlet extends HttpServlet {
    @EJB LivroBean bean;
    public void init() {
        try {
            bean.inicializar();
            ...
```

6 Concorrência, chamadas assíncronas e agendamento

A especificação EJB explicitamente proíbe beans de criar threads (além de não recomendar que EJBs não abram sockets ou streams de I/O). Embora a proibição esteja apenas na especificação (e não seja verificada pelo container), e uma má idéia criar threads porque eles podem interferir com o controle do ciclo de vida dos componentes realizado pelo container. Também não é recomendado o uso de travas e sincronização, que podem interferir no controle de transações e causar deadlocks.

Mas existem várias aplicações de threads, como notificações e agendamento que podem ser realizadas em EJB usando recursos como MDB, timers e métodos assíncronos. A sincronização de dados compartilhados é automática em singletons, mas existem várias opções de configuração.

6.1 Acesso concorrente em singletons

Por default, todo singleton bean tem controle de acesso concorrente, que impede acesso simultâneo a seus métodos. Esse acesso pode ser configurado através da anotação *@ConcurrencyManagement* a nível de classe e com travas de leitura/gravação em cada método usando a anotação @Lock.

6.1.1 Concorrência gerenciada belo bean

Por default, a gerência de concorrência e feita pelo container. Mas anotando-se a classe do Singleton com:

`@ConcurrencyManagement(ConcurrencyManagementType.BEAN)`

os métodos do singleton não serão mais thread-safe e o controle de concorrência ficará a cargo do bean, que pode usar modificadores da linguagem Java como *synchronized* ou *volatile*, ou travas do pacote *java.util.concurrent.locks*, como *Condition*, *Lock* e *ReadWriteLock*.

É uma solução flexível, porém mais complexa e deve ser usada raramente.

6.1.2 Concorrência gerenciada pelo container

Singleton beans anotados com

`@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)`

têm a concorrência gerenciada pelo container. Essa anotação não é necessária porque este é o default.

O container permite configurar o acesso concorrente de cada método com a anotação @Lock, que recebe como parâmetro um dos valores do *enum LockType*: *READ* ou *WRITE*. A opção READ permite acesso simultâneo, e com WRITE o acesso e exclusivo. Por default, todos os métodos de um Singleton se comportam como se estivessem anotados com *@Lock(LockType.WRITE)*.

Impedir o acesso simultâneo a todos os métodos pode introduzir um gargalo de performance em um singleton que seja acessado com muito mais frequência para ler (e não alterar) seu estado. Portanto, é uma boa prática anotar métodos que não alteram o estado do singleton com *@Lock(LockType.READ)*:

```
@Singleton
public class ExampleSingletonBean {

    private String state;

    @Lock(LockType.READ)
    public String getState() {
        return state;
    }

    @Lock(LockType.WRITE)
    public void setState(String newState) {
        state = newState;
    }
}
```

Se um método é marcado com *WRITE*, apenas um cliente poderá acessa-lo de cada vez. Os outros precisarão esperar o método terminar. Pode-se estabelecer um timeout para isto usando a anotação *@AccessTimeout* (na classe, para todos os métodos, ou em cada método

individualmente) informando quantos milissegundos o cliente pode esperar. Quando o timeout acabar, o container lança uma exceção (*ConcurrentAccessTimeoutException*):

```
@Singleton
@AccessTimeout(value=60000)
public class MySingletonBean {
    ...
}
```

6.2 Chamadas assíncronas

Uma das maneiras de executar uma operação assíncrona usando session beans é implementá-lo como cliente JMS. O método deve criar uma mensagem, empacotar os dados necessários para que a operação assíncrona seja executada, e enviar a mensagem para uma fila JMS mapeada por um MDB. Ao receber a mensagem, o MDB desempacota os parâmetros e chama a operação.

Embora esta possa ser uma solução eficiente, ela requer a criação de bastante código extra, além da necessidade de empacotar e desempacotar dados em uma mensagem. Uma outra forma de realizar operações assíncronas é usando métodos assíncronos.

Métodos assíncronos podem ser usados em qualquer session bean e precisam ou retornar void ou *Future<V>*. A primeira solução já é uma substituição adequada para a solução com MDB, que também não retorna valor. A segunda utiliza um objeto Future, do pacote *java.util.concurrent*, que permite monitorar chamadas assíncronas. É preciso também anotar o método com *@Asynchronous* (que também pode ser usado na classe, se todos os métodos da classe forem assíncronos).

O exemplo abaixo ilustra um bean contendo um método assíncrono:

```
@Stateless
public class Tradutor {
    @Asynchronous
    public Future<String> traduzir(String texto) {
        StringBuilder textoTraduzido;
        // ... realiza a tradução (demorada)
        return new AsyncResult<String>(textoTraduzido.toString());
    }
}
```

O cliente pode chamar o método e receber de volta um objeto Future. O resultado pode ser extraído quando estiver pronto usando o método get(). O método isDone() pode ser usado para monitorar o estado do objeto e descobrir quando o resultado pode ser extraído:

```
@SessionScoped @Named
public class ManagedBean {
```

```
    @EJB
    private Tradutor tradutor;

    private Future<String> future;

    public void solicitarTraducao(String texto) {
       future =  tradutor.traduzir(texto);
    }
    public String receberTraducao() {
       while (!future.isDone()){
           Thread.sleep(1000);
           // fazer outras coisas
       }
       return (String)future.get();
    }
}
```

6.3 Timers

Timers são serviços de agendamento que podem ser configurados em stateless e singleton session beans. Permitem que métodos sejam chamados periodicamente, ou uma única vez em data e hora marcada, ou depois de um intervalo especificado. Timers automáticos são configurados anotando métodos com @Schedule (ou @Schedules para múltiplos timers), que recebe como parâmetros as regras de agendamento. Os parâmetros são *hour*, *minute*, *second*, *dayOfWeek*, *dayOfMonth*, *timezone*, *year*, que podem receber expressões (para timers periódicos) ou valores fixos (para timers que executam apenas uma vez):

Exemplo

```
@Stateless
public class MyTimer {
   @Schedule(hour="*", minute="*", second="*/10")
   public void printTime() { ... }
...
}
```

A seguir alguns exemplos de expressões:

- hour="7,19,23", dayOfWeek="1-5" – executa as 7, 19 e 23 horas, de segunda a sexta
- minute="30", hour="5" - executa uma vez às 5 horas e 30 minutos
- hour="*", minute="*", second="*/10" - toda hora e minuto a cada 10 segundos

Timers também podem ser criados programaticamente usando os métodos de *TimerService* em beans que implementam a interface *TimedObject* e seu método *ejbTimeout(Timer).* Neste caso o timer geralmente é configurado na inicialização do bean:

```
@Singleton
@Startup
public class MyTimer implements TimedObject {

    @Resource TimerService timerService;

    @PostConstruct
    public void initTimer() {
        if (timerService.getTimers() != null) {
            for (Timer timer : timerService.getTimers()) {
                timer.cancel();
            }
        }
        timerService.createCalendarTimer(
            new ScheduleExpression().hour("*").minute("*").second("*/10"),
            new TimerConfig("myTimer", true)
        );
    }

    @Override
    public void ejbTimeout(Timer timer) {
        //código a executar quando o timer disparar
    }
}
```

Além do *createCalendarTimer()*, que cria um Timer para agendamento periódico, o *TimerService* também permite criar timers que disparam uma única vez com *createSingleActionTimer()*.

Se o bean não implementa a interface *TimedObject*, ele também pode definir um timeout usando um método anotado com *@Timeout*. O método precisa retornar void, e não precisa receber parâmetros (se receber deve ser um objeto Timer):

```
@Timeout
public void timeout(Timer timer) {
    //codigo a executar quando  timer disparar
}
```

6.4 Utilitários de concorrência do Java EE 7

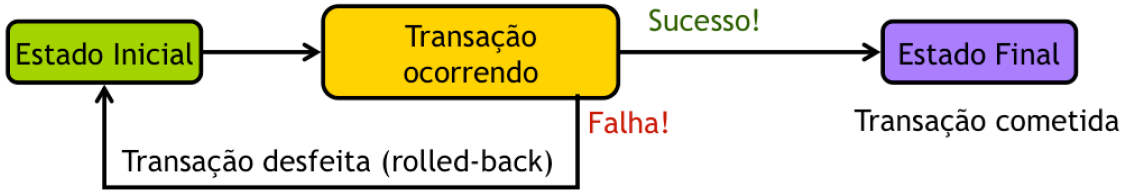
Existem vários utilitários de concorrência que devem ser usados caso haja a necessidade de trabalhar com threads dentro de aplicações Java EE. Elas estão declaradas no pacote *javax.enterprise.concurrent* e são similares a classes do pacote Java SE *java.util.concurrent*, porém gerenciadas pelo container e adequadas ao uso em EJBs e servlets. As mais importantes são:

- *ManagedExecutorService*
- *ManagedScheduledExecutorService*

- *ManagedThreadFactory*
- *ContextFactory*

7 Transações

Transações representam operações atômicas, indivisíveis. Um mecanismo de transações visa garantir que um procedimento ou termine com sucesso ou que todas as suas etapas sejam completamente desfeitas.



A especificação EJB não suporta transações aninhadas. Se uma transação começa quando já existe um contexto transacional, ela pode:

- Continuar o contexto transacional existente
- Interromper o contexto transacional existente
- Iniciar um novo contexto transacional

O controle de transações em EJB resume-se a *demarcação de transações*, ou seja, determinar quando ela será *iniciada* e quando será *concluída* (ou desfeita).

Estão disponíveis duas maneiras de demarcar transações:

- Explícita, ou programática (Bean-Managed Transactions – BMT)
- Implícita, ou declarativa (Container-Managed Transactions- CMT) - default

7.1 Transações gerenciadas pelo bean (BMT)

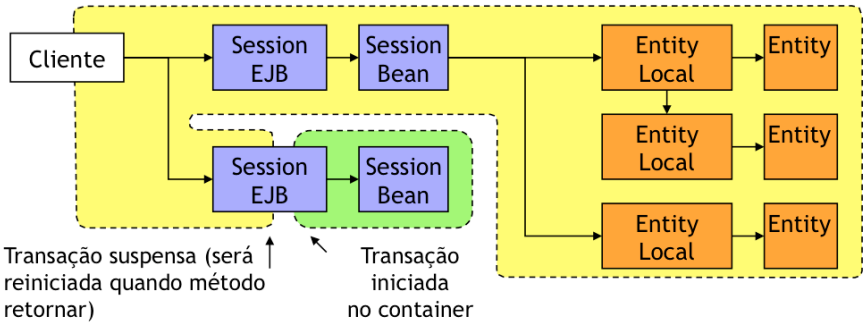
Transações gerenciadas pelo bean (BMT) envolvem o controle de transações através do uso direto de APIs para demarcação de transações em Java. Isto inclui métodos de controle transacional de *java.sql.Connection* (JDBC), *javax.jms.Session* (JMS), *EntityTransaction* em JPA e *UserTransaction* em JTA. Esses métodos são proibidos em CMT.

Para permitir que um bean utilize qualquer uma dessas APIs, é necessário configurá-lo usando: *@TransactionManagement*:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Fachada { ... }
```

7.1.1 Propagação de transações

Transações terminam no mesmo lugar onde começaram. O contexto da transação será propagado para todos os métodos chamados (se eles não iniciarem nova transação). Se métodos chamados iniciarem nova transação, a transação principal será suspensa até que o método termine.



O bean abaixo usa transações do JDBC, portanto precisa ser declarado como BMT:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Bean {

  @Resource DataSource ds;

  public void processarPedido (String id, int quantidade) {
     Connection con = ds.getConnection();
     try {
         con.setAutoCommit(false);
        atualizarEstoque(id, quantidade);
         con.commit();
     } catch (Exception e) {
       try {
            con.rollback();
       } catch (SQLException sqx) {}
     }
  }
}
```

7.1.2 UserTransaction

Containers JavaEE implementam a interface *javax.transaction.UserTransaction*, que pode ser usada em qualquer componente EJB, servlets e clientes standalone para demarcar transações em um container. As transações são distribuídas e usam a técnica two-phase commit. *UserTransaction* é usada transparentemente em transações CMT, mas também pode ser usada explicitamente em transações BMT.

Os principais métodos de *UserTransaction* usados para demarcar transações via BMT são:

- *begin()*: marca o início
- *commit():* marca o término
- *rollback():* condena a transação

O exemplo abaixo usa JTA para controlar uma transação:

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class Bean {
    @Resource EJBContext ctx;
    private double saldo;

    public void sacar(double quantia) {
         UserTransaction ut = ctx.getUserTransaction();
        try {
           double temp = saldo;
             ut.begin();
           saldo -= quantia;
           atualizar(saldo); // afetado pela transaçao
             ut.commit();
        } catch (Exception ex) {
           try {
                ut.rollback();
              saldo = temp;
           } catch (SystemException e2) {}
        }
    }
}
```

Clientes remotos podem obter uma referência para *UserTransaction* através de JNDI. O container deve disponibilizar o JTA na localidade *java:comp/UserTransaction*.

7.2 Transações gerenciadas pelo container (CMT)

O uso de Container-Managed Transactions (CMT) garante um controle de transações totalmente gerenciado pelo container. É mais simples de usar que BMT, e recomendado e default (mas pode ser especificado explicitamente via anotações ou configuração XML):

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class Fachada {…}
```

Em CMT não é possível demarcar blocos avulsos de código. A granularidade mínima é o método. As transações são gerenciadas por default de acordo com a politica REQUIRED, que garante que o método estará sempre dentro de um contexto transacional (continua contexto existente, ou cria um novo). CMT não permite o uso de nenhuma API transacional dentro do

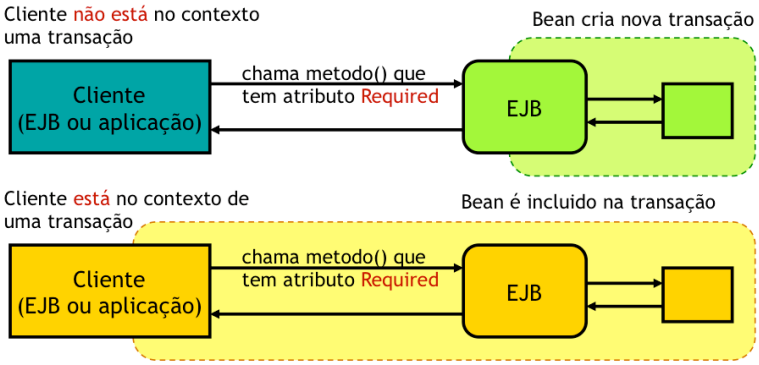
código (isto inclui métodos da API de transações de *java.sql*, *javax.jms*, *EntityTransaction* e *UserTransaction*).

7.2.1 Atributos (políticas de propagação)

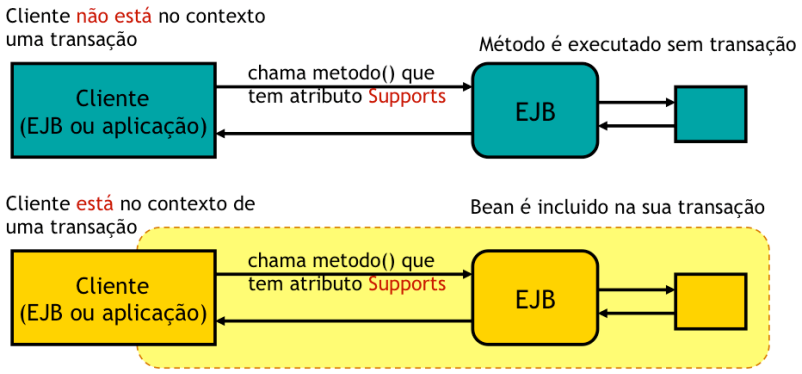
As politicas transacionais determinam como cada método irá reagir quando for chamado por um cliente dentro de um contexto transacional existente ou não. Pode-se definir uma política default para toda a classe, ou especificar individualmente em cada método com *@TransactionAttribute* e um dos tipos definidos no enum *TransactionAttributeType*:

- MANDATORY
- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- NEVER

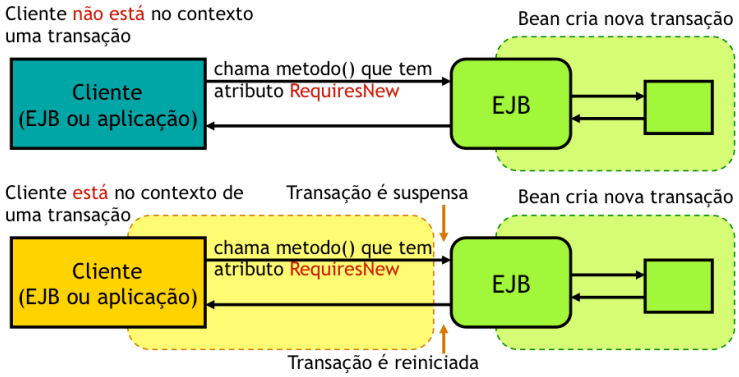
TransactionAttributeType.REQUIRED - Indica que o método precisa ser chamado dentro do escopo de uma transação. Se não existe transação, uma transação nova é criada e dura até que o método termine (é propagada para toda a cadeia de métodos chamados). Se já existe uma transação iniciada pelo cliente, o bean é incluído no seu escopo durante a chamada do método.



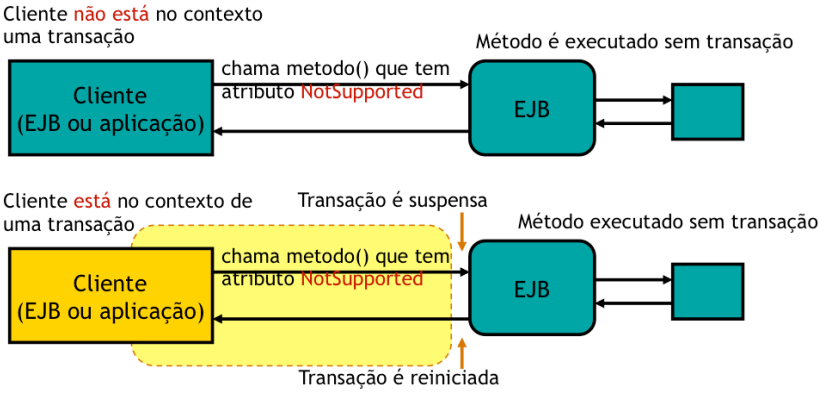
TransactionAttributeType.SUPPORTS - Indica que o método suporta transações, ou seja, ele será incluído no escopo da transação iniciada pelo cliente se ela existir. Se ele for chamado fora do escopo de uma transação ele realizará suas tarefa sem criar transações e poderá chamar objetos suportam ou não transações.



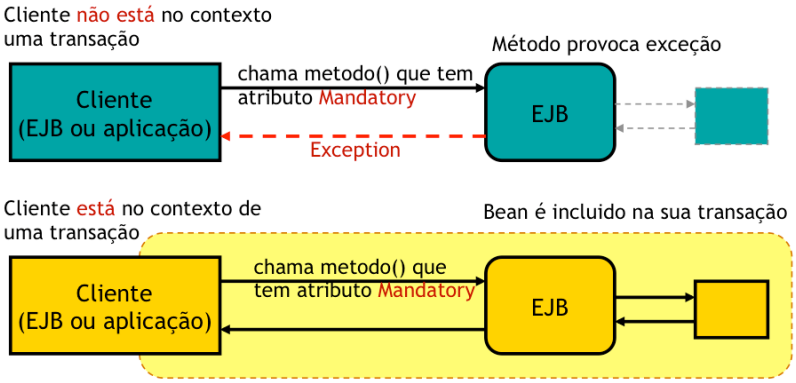
TransactionAttributeType.REQUIRES_NEW - Indica que uma nova transação, iniciada no escopo do bean, sempre será criada, estando ou não o cliente no escopo de uma transação. Se o cliente já tiver iniciado um contexto transacional, este será suspenso ate que a nova transação iniciada no método termine (ao final do método).



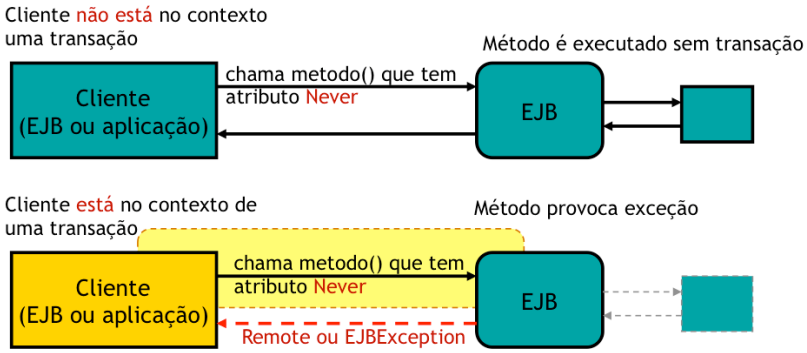
TransactionAttributeType.NOT_SUPPORTED - Indica que o método não suporta transações. Se o método for chamado pelo cliente no escopo de uma transação, ela será suspensa enquanto durar a chamada do método e não sera propagada. A transação do cliente será retomada quando o método terminar.



TransactionAttributeType.MANDATORY - Indica que a presença de um contexto transacional iniciado pelo cliente é obrigatória. Chamado fora do contexto transacional, causará javax.transaction.TransactionRequiredException.



TransactionAttributeType.NEVER - Indica que o método nunca pode ser chamado no escopo de uma transação. Se o cliente que chama o método iniciar ou propagar o contexto de uma transação, o bean causará uma RemoteException se for um bean remoto, ou EJBException se for local.



A declaração de *@TransactionAttribute no* nível da classe não afeta os métodos de callback do ciclo de vida (anotados com *@PostConstruct*, etc.) que têm um contexto transacional não especificado e dependente do tipo do bean.

7.2.2 Destino de uma transação em CMT

Apenas exceções do sistema (*Runtime*, *Remote*, *EJBException*) provocam rollback automático. O container não assume que outras exceções devam causar rollback. Capturar a exceção e chamar *rollback()* explicitamente não é uma alternativa legal pois o método não é

permitido em CMT. Portanto, para que uma exceção dispare automaticamente um rollback há duas alternativas:

- Anotar a exceção com *@ApplicationException*
- Capturar a exceção e condenar a transação CMT usando o método setRollbackOnly() de um EJBContext. O status da transação está disponível via getRollbackOnly():

```
try {
   return new ClientePK(clienteDAO.create(clienteDTO));
} catch (UsuarioJaExisteException e) {
   if (!ctx.getRollbackOnly())
      ctx.setRollbackOnly(); // doom this transaction
   throw e;
}
```

7.3 Transações em Message-driven beans

Como um cliente não chama um MDB diretamente, não é possível propagar um contexto transacional em um MDB, mas MDBs podem iniciar novos contextos transacionais. O escopo da transação deve iniciar e terminar dentro do método *onMessage()*.

É mais simples usar CMT, que considera a entrega da mensagem como parte da transação. Havendo rollback, o container poderá reenviar a mensagem. Se for usado BMT isto precisa ser feito explicitamente, lançando uma *EJBException* para evitar o *acknowledgement* e forçar o reenvio.

Para message-driven beans, apenas *NOT_SUPPORTED* e *REQUIRED* podem ser usados.

7.4 Sincronização de estado em Stateful Session Beans

Entities (JPA) mantêm seu estado no banco de dados. Quaisquer alterações em suas variáveis de instância, durante uma operação, serão revertidas em um rollback. Já o estado de Stateful Session Beans é mantido em variáveis de instância e não no banco de dados. O container não tem como recuperar o estado de suas variáveis em caso de rollback. É preciso guardar o estado anterior do bean para reverter caso a transação falhe.

Se o bean inicia a transação (BMT), podemos guardar o estado antes do begin e recuperar após o rollback. E se o container iniciar a transação (CMT)?

7.4.1 Interface SessionSynchronization

A interface SessionSynchronization pode ser implementada por Stateful Session Beans configurados com CMT para capturar eventos lançados nos pontos de demarcação. Ela consiste de três métodos:

- void **afterBegin**(): Chamado logo após o início da transação, deve guardar o estado do bean para recuperação em caso de falha.
- void **beforeCompletion**(): Chamado antes do commit() ou rollback(). Geralmente vazio, mas pode ser usado pelo bean para abortar a transação se desejar (usando setRollbackOnly())
- void **afterCompletion**(boolean state): Chamado depois do commit() ou rollback(). Se a transação terminou com sucesso, state é true; caso contrário, é false deve-se restaurar o estado do bean aos valores guardados em afterBegin()

8 Clientes EJB

Clientes EJB que rodam fora do container só podem acessar beans *@Remote* ou *@WebService*. Beans *@Remote* requerem proxy IIOP, obtido via lookup JNDI. Com a API *Embeddable EJB*, pode-se realizar a conexão sem depender de configurações proprietárias. Uma vez configurada, *EJBContainer.createEJBContainer()* obtém um contexto JNDI global para localizar o bean:

```
public void testEJB() throws NamingException {
    EJBContainer ejbC =
        EJBContainer.createEJBContainer(
           ResourceManager.getInitialEnvironment(new Properties()));
    Context ctx = ejbC.getContext();
    MyBean bean = (MyBean) ctx.lookup ("java:global/classes/org/sample/MyBean");
    // fazer alguma coisa com o bean
    ejbC.close();
}
```

O container deve ser fechado após o uso.

Para configurar, é preciso ter no classpath os JARs requeridos pelo container, que pode incluir JARs específicos do servidor usado. E necessário especificar as seguintes propriedades em um arquivo jndi.properties localizado no classpath:

- javax.ejb.embeddable.initial
- javax.ejb.embeddable.modules
- javax.ejb.embeddable.appName

O valor das propriedades é dependente do fabricante e do ambiente do servidor. O fabricante também poderá requerer propriedades adicionais.

9 Referências

[1] EJB 3.2 Expert Group. *JSR 345: Enterprise JavaBeans, Version 3.2*. Oracle. 2013. http://download.oracle.com/otndocs/jcp/ejb-3_2-fr-spec/index.html

[2] Eric Jendrock et al. *The Java EE 7 Tutorial*. Oracle. 2014. https://docs.oracle.com/javaee/7/JEETT.pdf

[3] Linda deMichiel and Bill Shannon. *JSR 342. The Java Enterprise Edition Specification. Version 7*. Oracle, 2013. http://download.oracle.com/otn-pub/jcp/java_ee-7-fr-spec/JavaEE_Platform_Spec.pdf

[4] Arun Gupta. *Java EE 7 Essentials*. O'Reilly and Associates. 2014.