Taller de Sistemas de Información 2

Enterprise Java Beans


Instituto de Computación


Facultad de Ingeniería


Universidad de la República de Uruguay

Enterprise Java Beans

- Son componentes server side
- Encapsulan lógica procedural
  - Típicamente representan procesos y/o reglas del negocio
- Integran un stack que resuelve entre otras cosas:
  - Mensajería, acceso remoto, web services, inyección de dependencias, ciclo de vida, transaccionalidad, manejo de excepciones, seguridad, etc.
- En la versión actual, aspectos que antes eran propietarios, ahora son estándar
  - Nombrado JNDI

Tipos de EJB

- Existen dos grandes tipos de componentes
- Componentes que encapsulan lógica sincrónica y componentes basados en mensajería
- Los primeros, son los Session Beans
  - Tienen funciones que encapsulan la lógica de negocio
- Los segundos, son los Message Driven Beans
  - Estos son componentes que actúan como consumidores de mensajes JMS
  - Permiten implementar lógica de negocio asíncrona y desacoplada

Session Beans

- Stateless
  - No mantienen estado conversacional
  - No presentan problemas de concurrencia
- Stateful
  - Mantienen estado conversacional
  - No presentan problemas de concurrencia
- Singleton
  - Se mantiene una sola instancia del bean, compartiendo el estado
  - Implementan efectivamente el patrón singleton
  - Pueden presentar problemas de concurrencia

Session Beans

- Los tres tipos de componentes tienen características en común, como ser el modelo de programación
- En algunos casos, estos pueden actuar como endpoints de servicios web
  - Servicios SOAP
  - Servicios REST / Recursos

Servicios del container

- Comunicación remota
- Inyección de dependencias
- Manejo del estado
- Pooling
- Ciclo de vida del componente
- Mensajería
- Gestión de transacciones
- Seguridad
- Soporte para concurrencia
- Interceptores

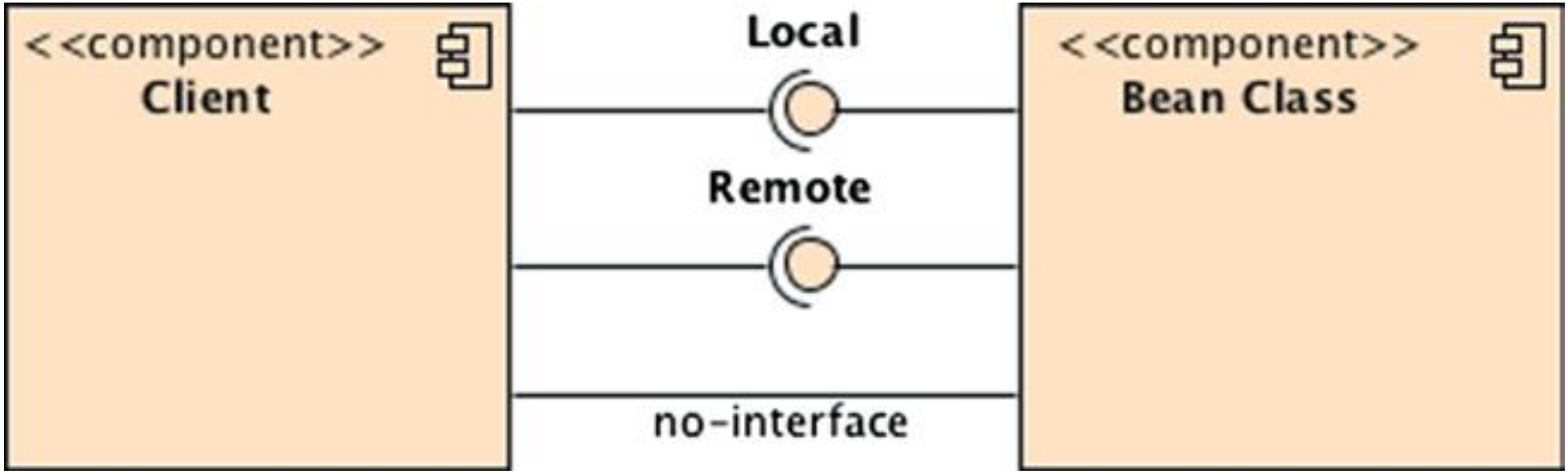
Ejemplo

```
@Stateless
public class BookEJB {

    @PersistenceContext(unitName = "UNIT")
    private EntityManager em;

    public Book findBookById(Long id) {
        return em.find(Book.class, id);
    }

    public Book createBook(Book book) {
        em.persist(book);
        return book;
    }
}
```

Anatomía de un Session Bean

- Un EJB suele estar compuesto por una clase de implementación (clase del bean) y cero o mas interfaces de negocio
- Clase de implementación
  - Contiene las implementaciones de los métodos de negocio
  - Puede implementar cero o mas interfaces de negocio
  - Debe esta anotada con: @Stateless, @Stateful o @Singleton
- Interfaces de negocio
  - Contiene las declaraciones de los métodos de negocio visibles al cliente
  - Son implementadas por el bean

Anatomía de un Session Bean



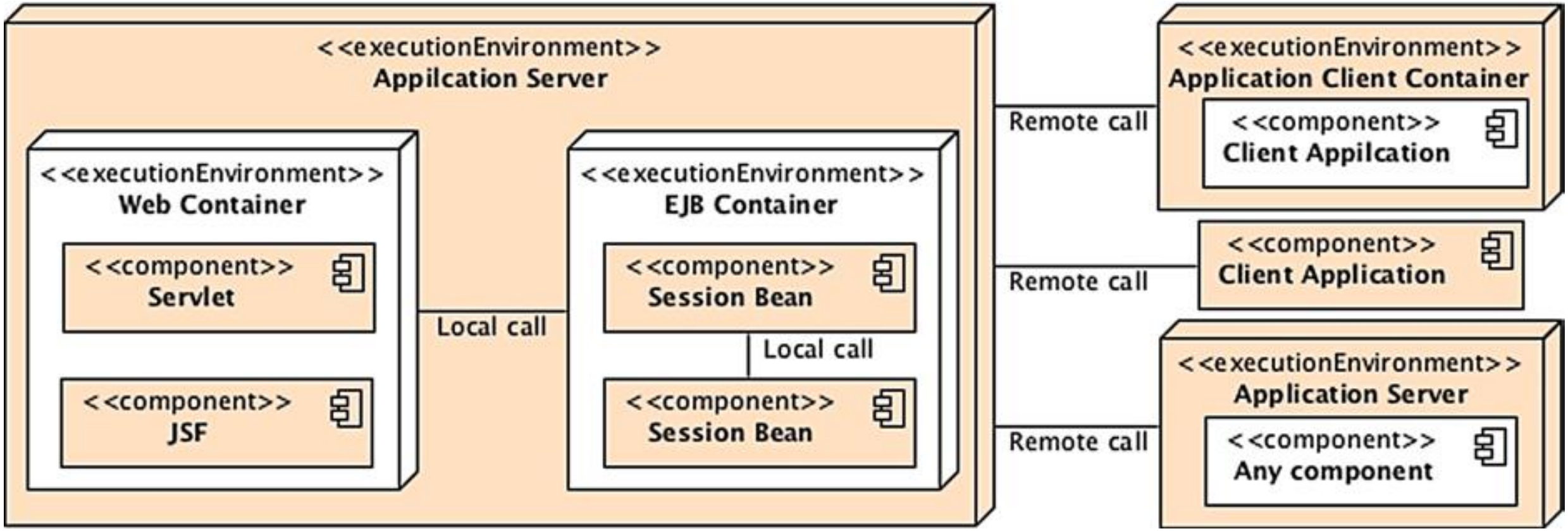
Clase de implementación

- Es cualquier clase Java que implementa la lógica del negocio
- Debe estar anotada con: @Stateless, @Stateful o @Singleton
- Debe implementar los métodos de su(s) interfaces
- Debe ser publica, no puede ser final o abstract
- No debe definir el método finalize()
- Los métodos no deben comenzar con “ejb” ni ser final o abstract
- Los argumentos y el tipo de retorno deben ser tipos validos RMI

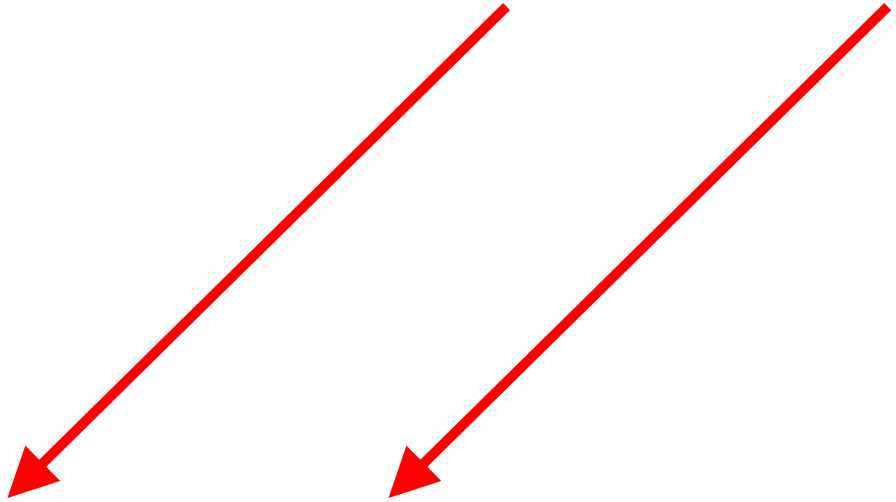
Interfaces

- Un session bean puede implementar cero o mas interfaces
  - Las interfaces pueden estar anotadas con alguna de estas anotaciones:
  - @Remote: Indica una interfaz que será accedida remotamente
  - @Local: Indica una interfaz que será accedida localmente (dentro de la JVM)
  - @WebService: Indica que la interfaz será accedida por un web service SOAP
  - @Path: Indica que la interfaz será accedida por un servicio REST (recurso)
- Una interfaz no puede estar marcada por mas de una anotación de acceso a la vez
- En caso de no usar una interfaz, se asume que se accede localmente, y a todos los métodos públicos del bean

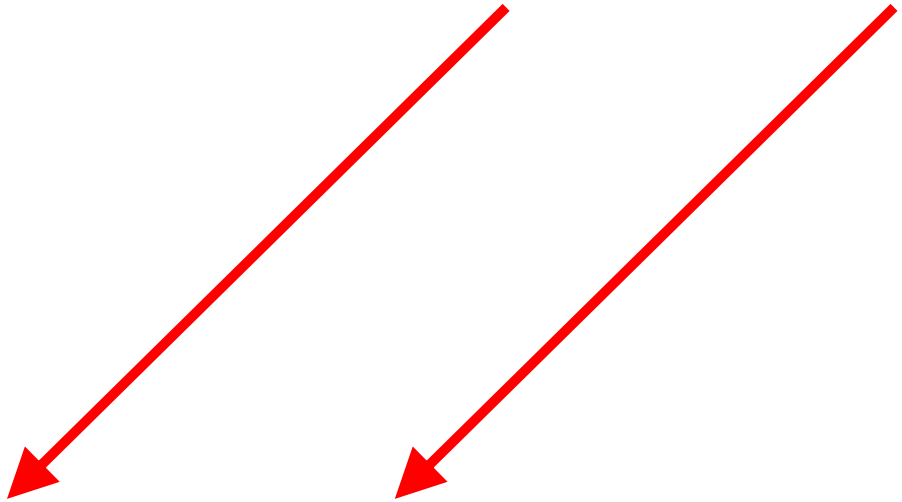
Interfaces


<<executionEnvironment>>
Appilcation Server
<<executionEnvironment>>
Web Container
<<component>>
Servlet
<<component>>
JSF
Local call
<<executionEnvironment>>
EJB Container
<<component>>
Session Bean
Local call
<<component>>
Session Bean
Remote call
Remote call
Remote call
<<executionEnvironment>>
Application Client Container
<<component>>
Client Appilcation
<<component>>
Client Application
<<executionEnvironment>>
Application Server
<<component>>
Any component

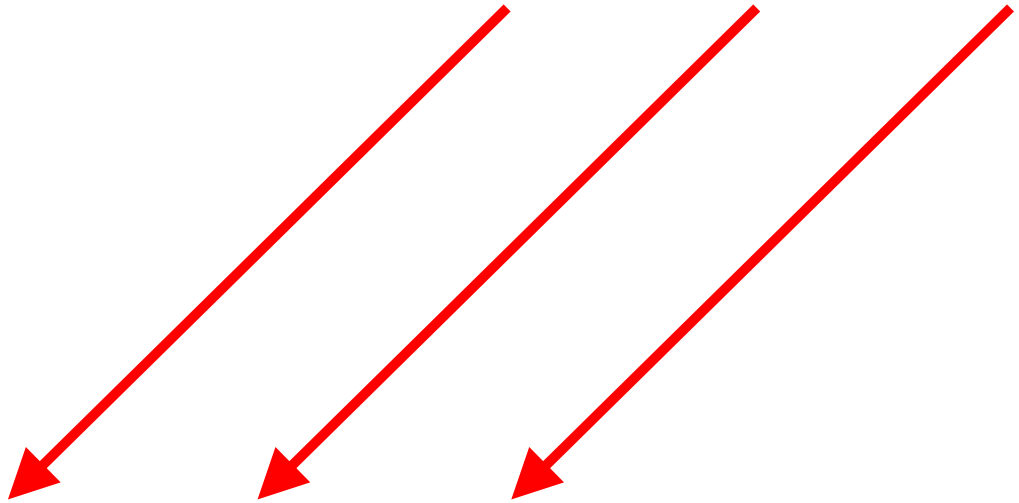
```java
@Local
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}

@Remote
public interface ItemRemote {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Stateless
public class ItemEJB implements ItemLocal, ItemRemote {
  // ...
```



```
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}

public interface ItemRemote {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {
  // ...
```



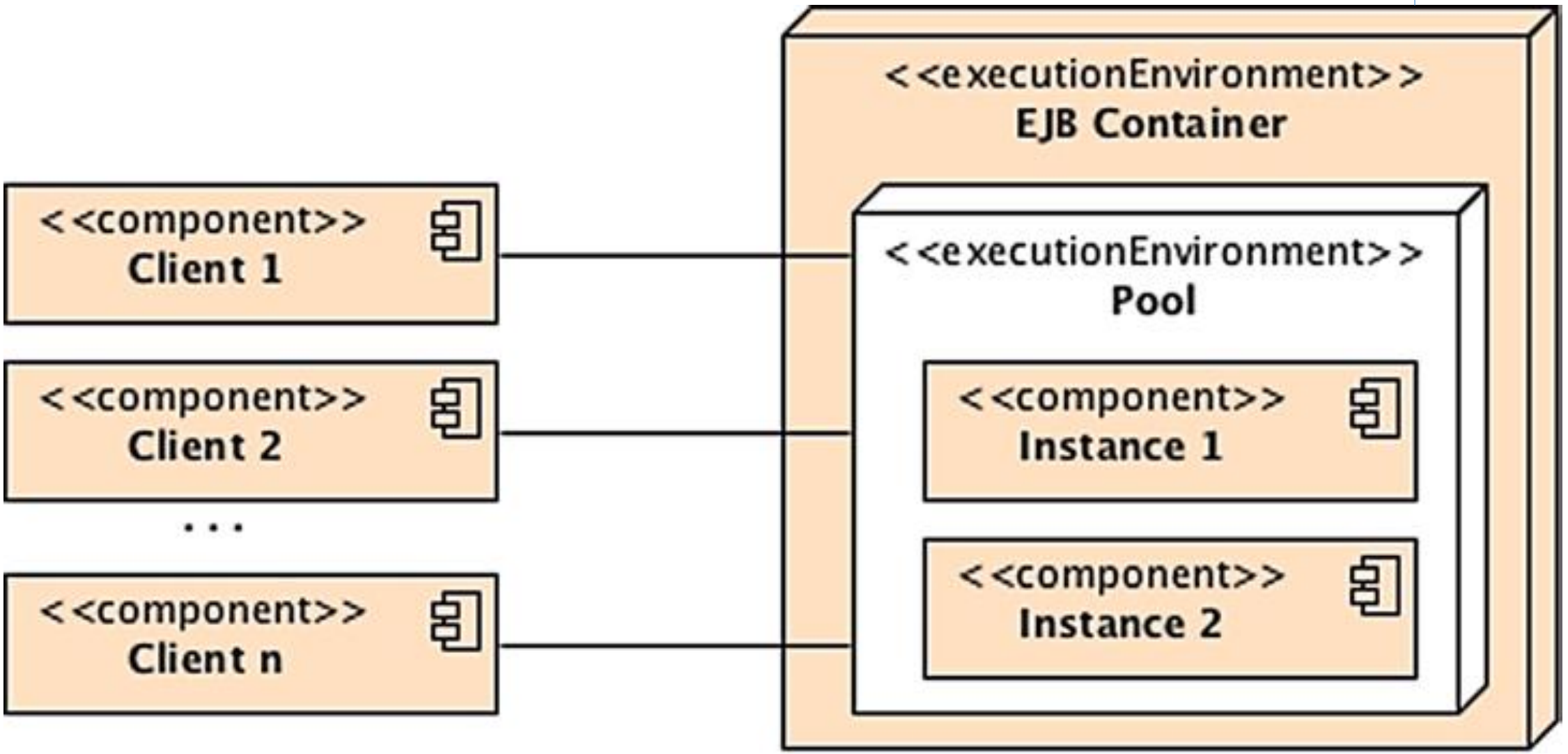
```java
@Local
public interface ItemLocal {
  List<Book> findBooks();
  List<CD> findCDs();
}

@WebService
public interface ItemSOAP {
  List<Book> findBooks();
  List<CD> findCDs();
  Book createBook(Book book);
  CD createCD(CD cd);
}

@Path(/items)
public interface ItemRest {
    List<Book> findBooks();
}

@Stateless
public class ItemEJB implements ItemLocal, ItemSOAP, ItemRest {
  // ...
}
```



Stateless beans

- Son el tipo de componente mas popular
- No manejan estado, lo que implica que (si nos interesa el estado), la operación debe ser completada en una sola llamada

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription(
   "Science fiction comedy series " +
   "created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);

statelessService.persistToDatabase(book);
```


<<executionEnvironment>>
EJB Container
<<component>>
Client 1
<<component>>
Client 2
. . .
<<component>>
Client n
<<executionEnvironment>>
Pool
<<component>>
Instance 1
<<component>>
Instance 2

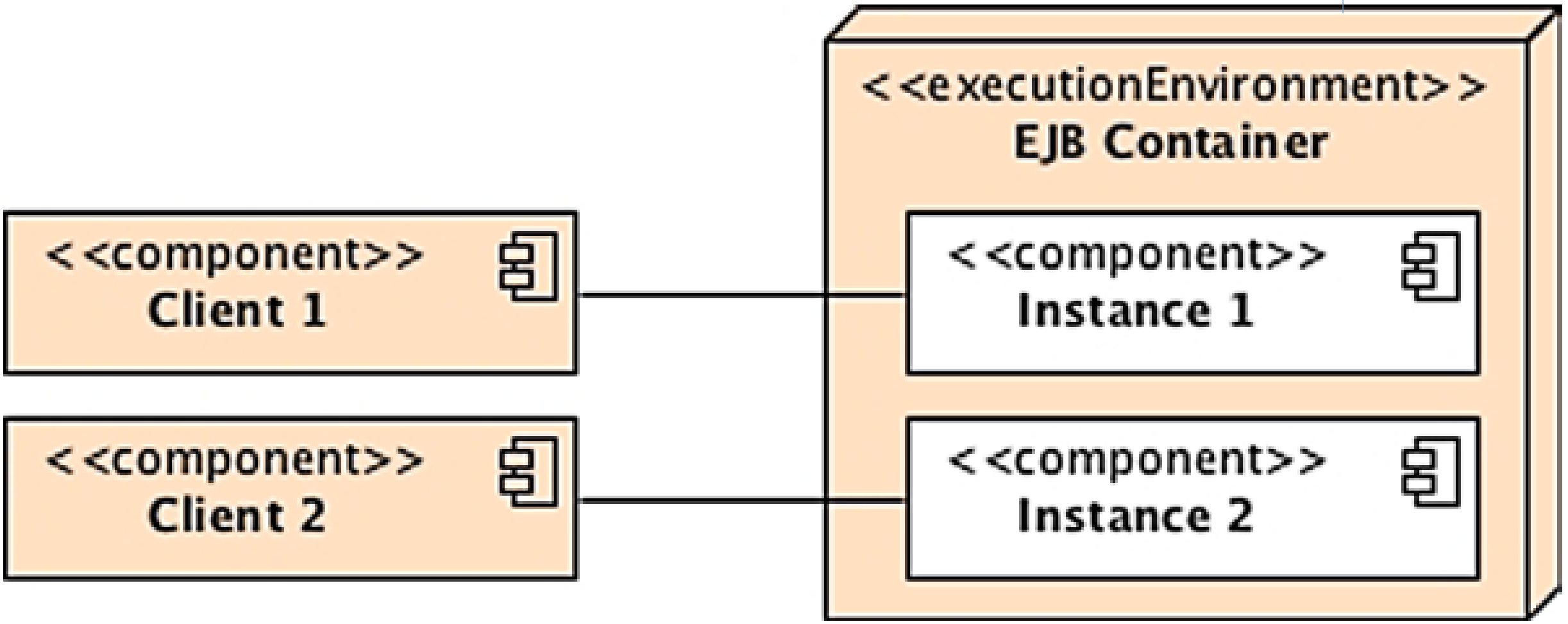
```
@Stateless
public class ItemEJB {
   @PersistenceContext(unitName = "chapter07PU")
   private EntityManager em;

   public List<Book> findBooks() {
      TypedQuery<Book> query =
            em.createNamedQuery(Book.FIND_ALL, Book.class);
      return query.getResultList();
   }
   public List<CD> findCDs() {
      TypedQuery<CD> query = em.createNamedQuery(
                                       CD.FIND_ALL, CD.class);
      return query.getResultList();
   }
   public Book createBook(Book book) {
      em.persist(book);
      return book;
   }
   public CD createCD(CD cd) {
      em.persist(cd);
      return cd;
   }
```

Stateful beans

- Permiten preservar el estado conversacional
- Son utiles para tareas que deben ser llevadas a cabo en varios pasos
- Por ejemplo, para implementar wizards
- Cada paso depende del estado generado por los pasos anteriores
- Un ejemplo típico puede ser un carrito de compras

Stateful beans

```
Book book = new Book();
book.setTitle("The Hitchhiker's Guide to the Galaxy");
book.setPrice(12.5F);
book.setDescription("Science fiction comedy series created by Douglas Adams.");
book.setIsbn("1-84023-742-2");
book.setNbOfPage(354);
statefulComponent.addBookToShoppingCart(book);
book.setTitle("The Robots of Dawn");
book.setPrice(18.25F);
book.setDescription("Isaac Asimov's Robot Series");
book.setIsbn("0-553-29949-2");
book.setNbOfPage(276);
statefulComponent.addBookToShoppingCart(book);
statefulComponent.checkOutShoppingCart();
```


<<executionEnvironment>>
EJB Container
<<component>>
Client 1
<<component>>
Client 2
<<component>>
Instance 1
<<component>>
Instance 2

Stateful beans

- A diferencia del caso anterior (stateless), en este caso tenemos un potencial problema de performance
- Un millón de clientes, se traducen en un millón de instancias para cada bean
- Para evitar estos problemas, el container hace uso de los conceptos de pasivado y activado

```
@Stateful
@StatefulTimeout(value = 20, unit = TimeUnit.SECONDS)
public class ShoppingCartEJB {
        private List<Item> cartItems = new ArrayList<>();
        public void addItem(Item item) {
                if (!cartItems.contains(item))
                        cartItems.add(item);
        }
        public void removeItem(Item item) {
                if (cartItems.contains(item))
                        cartItems.remove(item);
        }
        public Float getTotal() {
                if (cartItems == null || cartItems.isEmpty())
                        return 0f;
                Float total = 0f;
                for (Item cartItem : cartItems) {
                        total += (cartItem.getPrice());
                }
                return total;
        }
        @Remove
        public void checkout() {
                cartItems.clear();
        }
```

@javax.ejb.StatefulTimeout

- Esta anotación asigna un timeout
- Es el máximo tiempo en el cual el bean puede estar ocioso
- Una vez superado el tiempo, el bean es removido del container
- La unidad se expresa según la anotación:
  - java.util.concurrent.TimeUnit
  - El valor por defecto es MINUTE

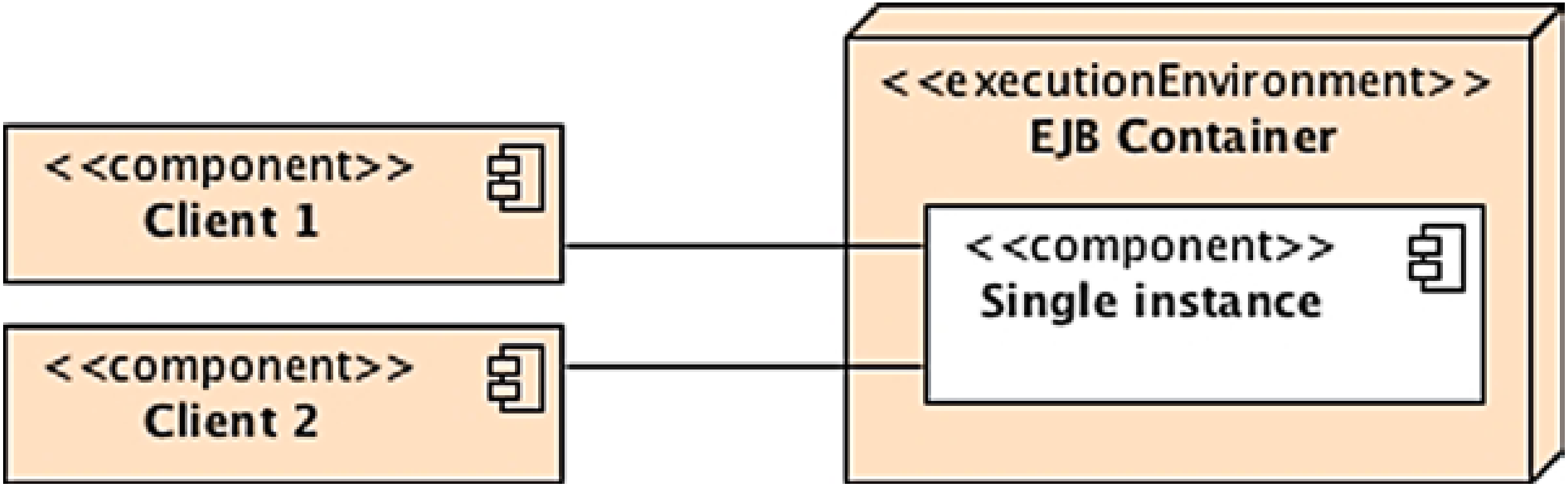
@javax.ejb.Remove

- Esta anotación se aplica sobre el método checkout
- Una vez ejecutado este método, la instancia del bean es removida del container
- Si no especificamos esta anotación, aun podemos contar con que el Garbage Collector mande eliminar esta instancia

Singleton beans

- Es un bean que se instancia una única vez por aplicación
- Es una implementación del patrón Singleton, pero basada en session beans, aprovechando los servicios brindados para el contenedor para estos

Singleton beans



```
@Singleton
public class CacheEJB {
    private Map<Long, Object> cache = new HashMap<>();

    public void addToCache(Long id, Object object) {
        if (!cache.containsKey(id))
            cache.put(id, object);
    }
    public void removeFromCache(Long id) {
        if (cache.containsKey(id))
            cache.remove(id);
    }
    public Object getFromCache(Long id) {
        if (cache.containsKey(id))
            return cache.get(id);
        else
            return null;
    }
}
```

Inicialización de Singletons

- Existen situaciones en las cuales la inicialización de un singleton es costosa
- Por ejemplo, si el CacheEJB anterior debiera poblarse con valores de una base de datos, esto puede ser costoso de realizar
- No es una tarea para realizar en cada acceso al componente
- Podemos decirle al container que inicialice un Singleton al inicializar con la anotación @Startup

Inicialización de Singletons

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

Encadenamiento de Singletons

- Existen situaciones en donde los datos de un singleton deben provenir de otro singleton
- En este caso, es importante que el primero sea inicializado antes que el segundo
- Para esto usamos la anotación @javax.ejb.DependsOn

Encadenamiento de Singletons

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

@DependsOn("CountryCodeEJB")
@Singleton
public class CacheEJB {...}
```

@DependsOn

- Utiliza una serie de strings como parámetros, donde cada una define de que EJBs Singletons se depende

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

@Singleton
public class ZipCodeEJB {...}
@DependsOn("CountryCodeEJB", "ZipCodeEJB")
@Startup
@Singleton
public class CacheEJB {...}
```

@DependsOn

- En caso de que los EJBs se encuentren en módulos diferentes, y necesitemos realizar una referencia explicita a esto, podemos hacer…

```
@DependsOn("business.jar#CountryCodeEJB")
@Singleton
public class CacheEJB {...}
```

Concurrencia

- La única instancia compartida puede sufrir acceso concurrente por parte de los cliente
- La anotación @ConcurrencyManagement controla este acceso concurrente por parte de los cliente
- Tenemos dos opciones:
  - Container-managed concurrency (CMC)
  - Bean-managed concurrency (BMC)

Container-Managed Concurrency

- Permite controlar el acceso concurrente
- Usamos la anotación @Lock, que puede tomar dos valores, READ (Shared) o WRITE (Exclusive)
- @Lock(LockType.WRITE)
  - No permite accesos concurrentes mientras la ejecución este en curso
- @Lock(LockType.READ)
  - Solo permite otros accesos concurrentes, pero con el mismo tipo de lock (READ)

Container-Managed Concurrency

- @Lock puede ser especificada en la clase, en el método o en ambos
- Si no especificamos el tipo de acceso, se asume @Lock(LockType.WRITE)
- @AccessTimeout puede ser usado para determinar cuanto tiempo esperar como máximo ante un lock que no puede ser obtenido

```
@Singleton
@Lock(LockType.WRITE)
@AccessTimeout(value = 20, unit = TimeUnit.SECONDS)
public class CacheEJB {

  private Map<Long, Object> cache = new HashMap<>();

  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }

  public void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }

  @Lock(LockType.READ)
  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

Container-Managed Concurrency

- Si un addToCache() se bloquea mas de 20 segundos, el cliente recibirá una excepción de tipo ConcurrentAccessTimeoutException
- La unidad del valor especificado, se indica con el enumerado TimeUnit
- En el caso anterior, esta es de tipo SECONDS

Bean-Managed Concurrency

- En este caso el container permite el acceso concurrente
- El programador es el encargado de controlar dicho acceso usando la palabra clave synchronized
- Puede estar aplicada a nivel de método o como parte del código usando un bloque synchronized

```
@Singleton
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class CacheEJB {

  private Map<Long, Object> cache = new HashMap<>();

  public synchronized void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }

  public synchronized void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }

  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
}
```

Concurrency Not Allowed

- Existen situaciones en donde no queremos permitir acceso concurrente al bean
- Podemos usar @AccessTimeout para regular la espera por un lock
- Un valor de -1, indica que la espera debe ser indefinida
- Un valor de 0, indica que el acceso concurrente no esta permitido
  - Puede generar condiciones de carrera

```
@Singleton
public class CacheEJB {

  private Map<Long, Object> cache = new HashMap<>();

  @AccessTimeout(0)
  public void addToCache(Long id, Object object) {
    if (!cache.containsKey(id))
      cache.put(id, object);
  }

  public void removeFromCache(Long id) {
    if (cache.containsKey(id))
      cache.remove(id);
  }
```

Inyección de dependencias

- Los EJB soportan inyección de dependencias
- Si el proyecto en el que se encuentran, incluye soporte CDI, entonces las inyecciones pueden resolverse con las anotaciones de CDI (como ser @Inject)
- En caso de no contar con un proyecto CDI, se proveen anotaciones para inyectar los elementos mas comunes dentro de la plataforma

Inyección de dependencias

- @EJB
  - Inyecta una referencia a un EJB especificado a través de la interfaz o de la clase del bean
- @PersistenceContext y @PersistenceUnit
  - Permite inyectar un EntityManager o una EntityManagerFactory, respectivamente
- @WebServiceRef
  - Inyecta una referencia a un servicio web
- @Resource
  - Permite inyectar recursos, como datasources JDBC, connection factories de JMS, el Timer Service, etc.

```
@Stateless
public class ItemEJB {

  @PersistenceContext
  private EntityManager em;

  @EJB
  private CustomerEJB customerEJB;

  @Inject
  private NumberGenerator generator;

  @WebServiceRef
  private ArtistWebService artistWebService;

  private SessionContext ctx;

  @Resource
  public void setCtx(SessionContext ctx) {
    this.ctx = ctx;
  }
  //...
```

Session Context

- A veces es necesario que el componente acceda a los servicios que provee el container
  - Por ejemplo, validar manualmente un usuario
  - Iniciar y/o commitear manualmente una transacción de negocio
- Para esto podemos inyectar una instancia de javax.ejb.SessionContext
- Podemos hacerlo con @Inject o @Resource

javax.ejb.SessionContext

Method	Description
`getCallerPrincipal`	Returns the `java.security.Principal` associated with the invocation.
`getRollbackOnly`	Tests whether the current transaction has been marked for rollback.
`getTimerService`	Returns the `javax.ejb.TimerService` interface. Only stateless beans and singletons can use this method. Stateful session beans cannot be timed objects.
`getUserTransaction`	Returns the `javax.transaction.UserTransaction` interface to demarcate transactions. Only session beans with bean-managed transaction (BMT) can use this method.
`isCallerInRole`	Tests whether the caller has a given security role.
`lookup`	Enables the session bean to look up its environment entries in the JNDI naming context.
`setRollbackOnly`	Allows the bean to mark the current transaction for rollback.
`wasCancelCalled`	Checks whether a client invoked the `cancel()` method on the client `Future` object corresponding to the currently executing asynchronous business method.

```
@Stateless
public class ItemEJB {
   @PersistenceContext(unitName = "UNIT_NAME")
   private EntityManager em;

   @Resource
   private SessionContext context;

   public Book createBook(Book book) {
      if (!context.isCallerInRole("admin"))
         throw new SecurityException("Only admins can execute");

      em.persist(book);

      if (inventoryLevel(book) == TOO_MANY_BOOKS)
         context.setRollbackOnly();
      return book;
   }
```

JNDI

- Java Naming and Directory Interface
- Es un API que permite acceder a componentes EJB
- Es estándar, independiente del servidor de aplicaciones en donde los componentes se encuentran instalados:
- **java:<scope>[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]**

JNDI

- scope
  - **global**: Permite que un componente externo acceda al namespace global
  - **app**: Permite que un componente en una aplicación, acceda al namespace de la aplicación
  - **module**: Permite que un componente en un modulo acceda al namespace de la aplicación
  - **comp**: Es un namespace especifico para la aplicación, inaccesible desde el exterior

JNDI

- app-name
  - Solo se requiere si el componente esta dentro de un WAR o EAR
  - El valor por defecto es el nombre del EAR o WAR sin la extensión
- module-name
  - Es el nombre del modulo en el cual el session bean se encuentra instalado
  - Puede ser un JAR o un WAR
  - El valor por defecto es el archivo sin la extensión

JNDI

- bean-name
  - Es el nombre del session bean
  - Por defecto es el nombre de la clase
- fully-qualified-interface-name
  - Es el nombre de la interfaz (nombre completo con package) por la cual se accede al componente
  - En el caso de no tener interfaces, este nombre puede ser el nombre completo de la clase

```
package org.javaee;

@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB
      implements ItemLocal, ItemRemote {

   ...
}
```

JNDI

- Dentro de un WAR

```
java:global/cdbookstore/ItemEJB!org.javaee.ItemRemote
java:global/cdbookstore/ItemEJB!org.javaee.ItemLocal
java:global/cdbookstore/ItemEJB!org.javaee.ItemEJB
```

- Dentro de un EAR

```
java:global/myapplication/cdbookstore/ItemEJB!org.javaee.ItemRemote
java:global/myapplication/cdbookstore/ItemEJB!org.javaee.ItemLocal
java:global/myapplication/cdbookstore/ItemEJB!org.javaee.ItemEJB
```

Deployment descriptor

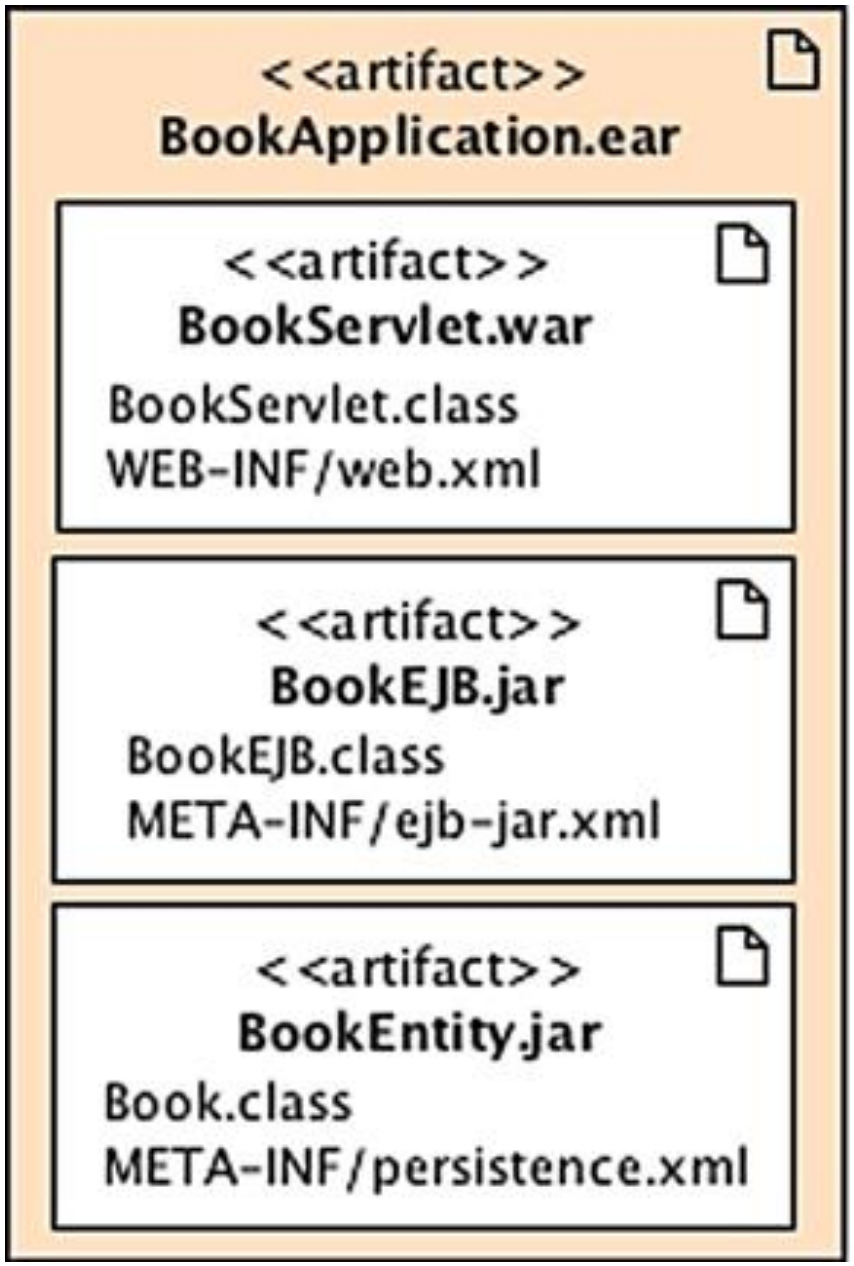
- Los metadatos especificados en código (con anotaciones), pueden ser configurados a través de archivos XML
  - No son necesarios en la versión actual
- Esto se realiza en el archivo ejb-jar.xml
- Este debe ubicarse en la carpeta META-INF del modulo EJB que estamos instalando

Packaging

- Los EJBs deben ser empaquetados antes de ser instalados en el servidor de aplicaciones
- En un modulo EJB generalmente encontramos
  - La clase del bean
  - Sus interfaces
  - Cualquier clase auxiliar referenciada
  - Excepciones que sean utilizadas
  - El deployment descriptor (opcional)

Packaging

- Cuando colocamos estos artefactos en un modulo (JAR), podemos colocar este dentro del servidor de aplicaciones
- Opcionalmente podemos colocar este modulo dentro de un archivo EAR (el cual se coloca dentro del servidor de aplicaciones)
- El modulo que contiene los EJBs, se denomina modulo EJB
- También podemos colocar los EJBs directamente en un modulo WEB, sin necesidad de separarlos en un ensamblado independiente


<<artifact>>
BookApplication.ear
<<artifact>>
BookServlet.war
BookServlet.class
WEB-INF/web.xml
<<artifact>>
BookEJB.jar
BookEJB.class
META-INF/ejb-jar.xml
<<artifact>>
BookEntity.jar
Book.class
META-INF/persistence.xml


<<artifact>>
BookApplication.war
WEB-INF/classes/BookEJB.class
WEB-INF/classes/Book.class
WEB-INF/classes/BookServlet.class
WEB-INF/ejb-jar.xml
WEB-INF/persistence.xml
WEB-INF/web.xml

Invocando EJBs

- Cuando un componente no define interfaces de acceso

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

// Client code injecting a reference to the EJB
@EJB ItemEJB itemEJB;
```

Invocando EJBs

- Si tenemos una interfaz de negocio implementada, debemos especificar como accederemos

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

// Client code injecting several references to the EJB or interfaces
@EJB ItemEJB itemEJB;
@EJB ItemLocal itemEJBLocal;
@EJB ItemRemote itemEJBRemote;
```

Invocando EJBs

- Si bien en este caso no es tan cómodo, podemos incluso buscar con el nombre JNDI portable usando la anotación @EJB

```
@EJB(lookup = "java:global/classes/ItemEJB")
ItemRemote itemEJBRemote;
```

Invocando EJBs

```
@Stateless
@Remote(ItemRemote.class)
@Local(ItemLocal.class)
@LocalBean
public class ItemEJB implements ItemLocal, ItemRemote {...}

@Inject ItemEJB itemEJB;
@Inject ItemLocal itemEJBLocal;
@Inject ItemRemote itemEJBRemote;
```

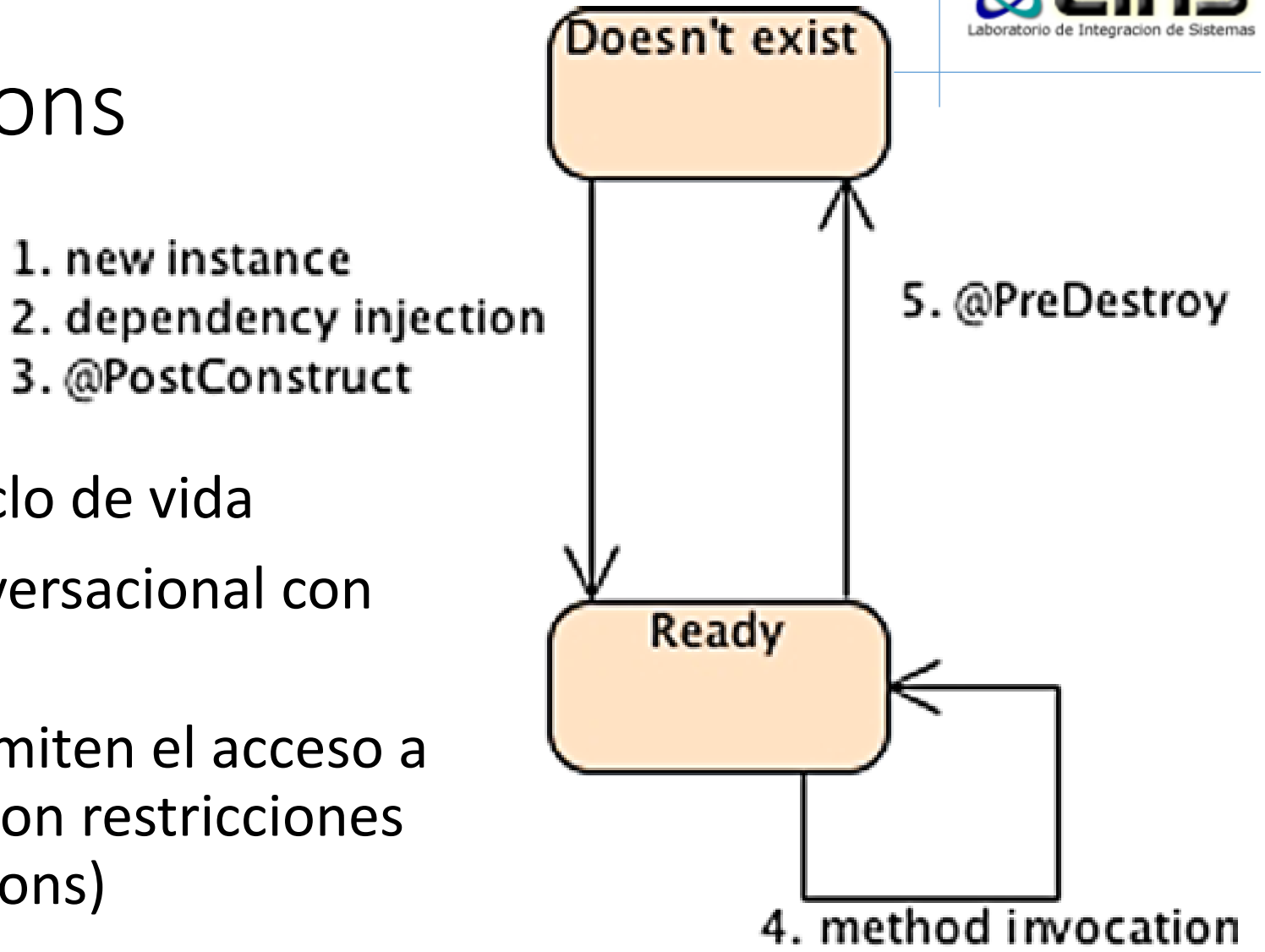
- En el caso de invocar a través de una búsqueda JNDI, podemos utilizar @Produces

```
// Code producing a remote EJB
@Produces @EJB(lookup = "java:global/classes/ItemEJB")
ItemRemote itemEJBRemote;

// Client code injecting the produced remote EJB
@Inject ItemRemote itemEJBRemote;
```

Callbacks

- Como en el caso de los beans CDI, los componentes EJB tienen un ciclo de vida asociado
- En dicho ciclo de vida, encontramos eventos, ante los cuales podemos responder
- En el caso de que el bean sea implementado como un bean CDI, podemos atender dichos eventos como fue visto en CDI

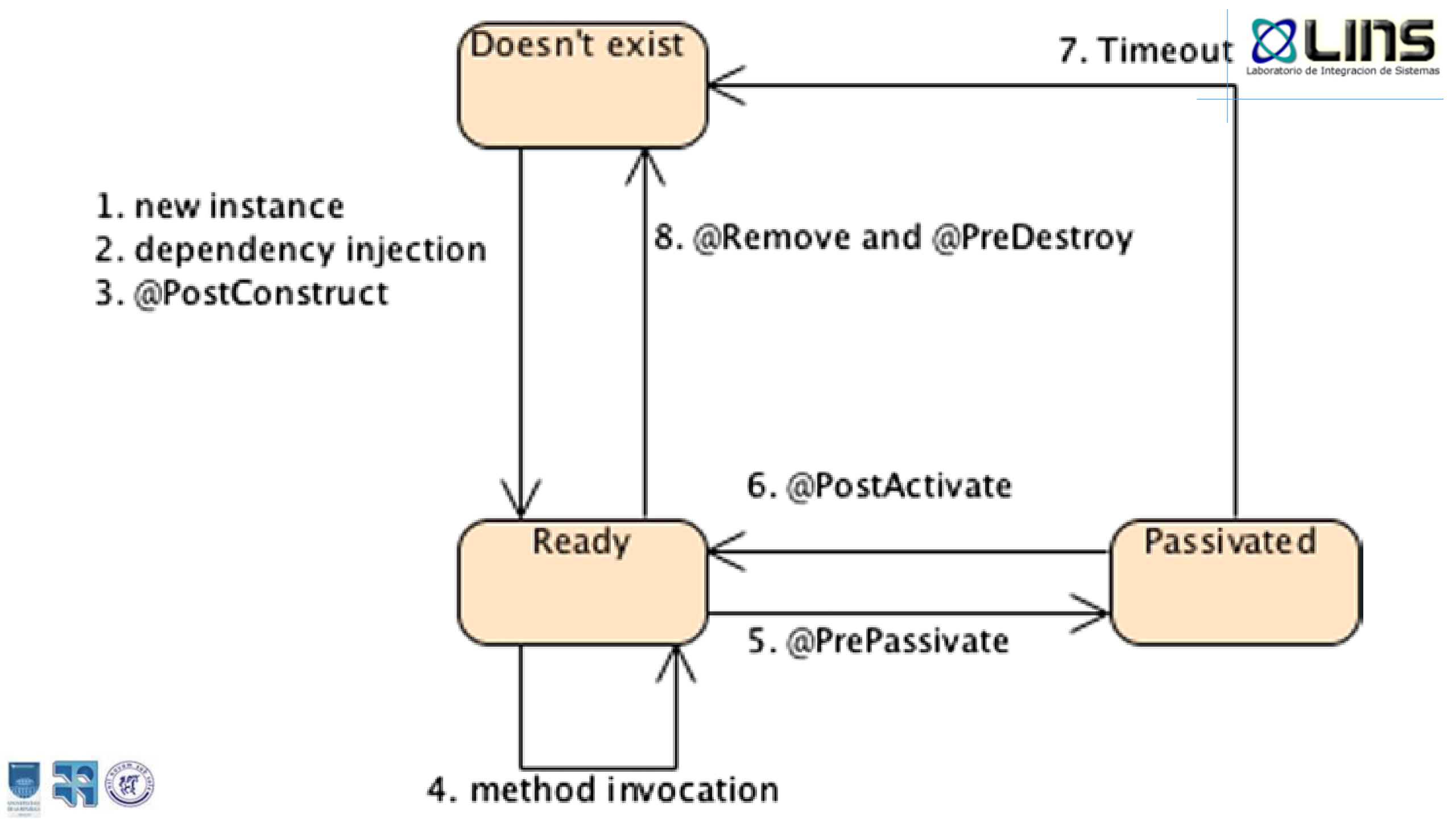
Stateless y Singletons



- Ambos tienen el mismo ciclo de vida
- No mantienen estado conversacional con un cliente dedicado
- Ambos tipos de beans permiten el acceso a cualquier tipo de cliente (con restricciones de concurrencia en Singletons)

Stateful

- Son similares a los anteriores, pero con la diferencia que se mantiene estado conversacional con un cliente dedicado
  - 1000 usuarios simultáneos, generan 1000 instancias de Stateful
- El contenedor puede sacar una instancia de memoria si no es usada por una cantidad de tiempo determinada


Doesn't exist
7. Timeout
1. new instance
2. dependency injection
3. @PostConstruct
8. @Remove and @PreDestroy
6. @PostActivate
Ready
Passivated
5. @PrePassivate
4. method invocation

Stateful

- En algunos casos, los componentes abren conexiones JDBC o conexiones de red
- Estos recursos no pueden ser mantenidos abiertos para cada instancia de Stateful
- Debemos abrirlos y cerrarlos luego de activar y antes de pasivar un componente
- También podemos desactivar el pasivado / activado con la siguiente anotación:
  - @Stateful(passivationCapable=false)

Callbacks disponibles

- Las siguientes anotaciones pueden ser aplicadas sobre métodos de callback
  - @PostConstruct
  - @PreDestroy
  - @PrePassivate
  - @PostActivate

Reglas para un callback

- El método no debe devolver valores ni recibir parámetros
- No puede propagar checked exceptions, pero si puede propagar runtime exceptions
- El método no puede ser static o final
- El método puede tener múltiples anotaciones, pero no podemos tener dos métodos diferentes con la misma anotación
- Un método de callback puede acceder las environment entries del bean

```
@Singleton
public class CacheEJB {

  private Map<Long, Object> cache = new HashMap<>();

  @PostConstruct
  private void initCache() {
    cache.put(1L, "First item in the cache");
    cache.put(2L, "Second item in the cache");
  }

  public Object getFromCache(Long id) {
    if (cache.containsKey(id))
      return cache.get(id);
    else
      return null;
  }
```

```
@Stateful
public class ShoppingCartEJB {

  @Resource(lookup = "java:comp/defaultDataSource")
  private DataSource ds;
  private Connection connection;

  private List<Item> cartItems = new ArrayList<>();

  @PostConstruct
  @PostActivate
  private void init() {
    connection = ds.getConnection();
  }

  @PreDestroy
  @PrePassivate
  private void close() {
    connection.close();
  }
```

Llamadas asíncronas

- Por defecto las invocaciones a los métodos del bean son síncronas
- Podemos transformar la llamada a un método en asíncrona, usando la anotación: @javax.ejb.Asynchronous
- Es útil cuando tenemos métodos que sabemos de antemano que son consumidores de tiempo
- Puede ser aplicada a nivel de clase o a nivel de métodos

```
@Stateless
public class OrderEJB {

    @Asynchronous
    public void sendEmailOrderComplete(
                                        Order order,
                                        Customer customer) {
        // Very Long task
    }

    @Asynchronous
    public void printOrder(Order order) {
        // Very Long task
    }
}
```

Llamadas asíncronas

- Cuando un cliente invoca uno de estos métodos, el control se devuelve al llamador
- La ejecución del método invocado continua en un thread aparte
- Existen sin embargo situaciones en las que debemos obtener un valor resultado
- Usamos la anotación @java.util.concurrent.Future<V>, donde V representa el valor devuelto

```
@Stateless
@Asynchronous
public class OrderEJB {

  @Resource
  SessionContext ctx;

  public Future<Integer> sendOrderToWorkflow(Order order) {
    Integer status = 0;

    // processing
    status = 1;

    if (ctx.wasCancelCalled()) {
      return new AsyncResult<>(2);
    }

    // processing

    return new AsyncResult<>(status);
  }
}
```

Future<V>

- En el caso anterior, se devuelve un valor de este tipo
- El cliente puede recuperar el valor usando Future.get()
- Si por algún motivo se debe cancelar la operación, se puede invocar Future.cancel()
  - El container intenta cancelar, solo si la llamada asíncrona no ha iniciado aun

Future<V>

- El método invocado puede invocar el método SessionContext.wasCancelCalled()
- Esto permite determinar si se solicito cancelar la innovación del método asíncrono
- javax.ejb.AsyncResult<V> es una implementación de Future<V>, que permite devolver el valor al container

```
@Stateless
@Asynchronous
public class OrderEJB {

  @Resource
  SessionContext ctx;

  public Future<Integer> sendOrderToWorkflow(Order order) {
    Integer status = 0;

    // processing
    status = 1;

    if (ctx.wasCancelCalled()) {
      return new AsyncResult<>(2);
    }

    // processing

    return new AsyncResult<>(status);
  }
}
```

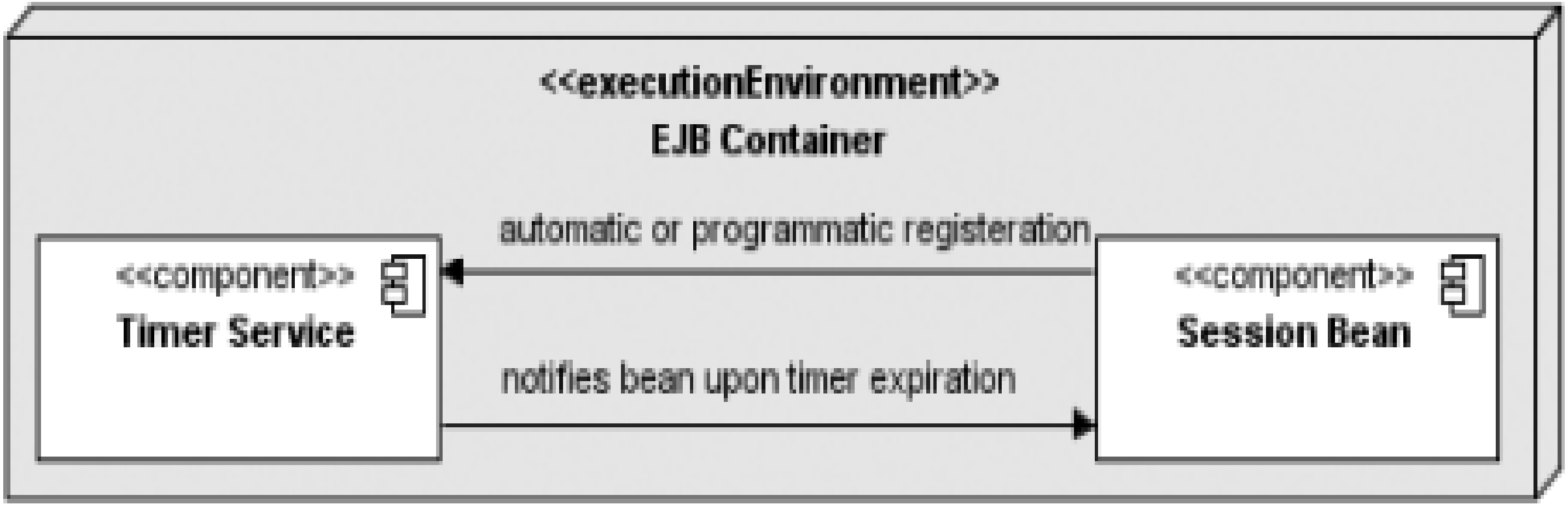
Timer Service

- Permite implementar notificaciones temporales
- Este servicio permite registrar notificaciones temporales para todo tipo de beans, salvo para Stateful Beans
- Podemos notificar en base a:
  - Una planificación basada en calendarios (calendar schedule)
  - En un momento especifico
  - Luego de un intervalo de tiempo
  - O en intervalos de tiempo predefinidos
- Los timers pueden ser programáticos o automaticos

Timer Service

- Programmatic Timers
  - Son establecidos explícitamente invocando alguno de los métodos de la interfaz TimerService
- Automatic Timers
  - Son creados automáticamente luego de un deploy exitoso
  - Se crean en base a métodos que tengan las anotaciones @Schedule o @Schedules

Timer Service



Calendar-Based Timer Expressions

- Ambos tipos de timers pueden ser establecidos en base a una planificación basada en calendarios
- Se expresa en forma similar a las expresiones CRON o QUARTZ
- Existen multiples atributos que forman una de estas expresiones
- Al combinarlos se forma una calendar based timer expression
  - dayOfMonth="Last“
  - year="2011“
  - month="July"

Attribute	Description	Default Value	Allowable Values and Examples
`second`	One or more seconds within a minute	0	0 to 59. For example: `second="30"`.
`minute`	One or more minutes within an hour	0	0 to 59. For example: `minute="15"`.
`hour`	One or more hours within a day	0	0 to 23. For example: `hour="13"`.
`dayOfWeek`	One or more days within a week	*	0 to 7 (both 0 and 7 refer to Sunday). For example: `dayOfWeek="3"`. `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`. For example: `dayOfWeek="Mon"`.
`dayOfMonth`	One or more days within a month	*	1 to 31. For example: `dayOfMonth="15"`. −7 to −1 (a negative number means the *n*th day or days before the end of the month). For example: `dayOfMonth="-3"`. `Last`. For example: `dayOfMonth="Last"`. [`1st`, `2nd`, `3rd`, `4th`, `5th`, `Last`] [`Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`]. For example: `dayOfMonth="2nd Fri"`.
`month`	One or more months within a year	*	1 to 12. For example: `month="7"`. `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`, `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec`. For example: `month="July"`.
`year`	A particular calendar year	*	A four–digit calendar year. For example: `year="2011"`.

Calendar-Based Timer Expressions

- Podemos usar wildcards
  - minute=”*”
  - dayOfWeek=”*”
- También podemos especificar listas de valores
  - Los duplicados se ignoran
  - dayOfWeek=”Tue, Thu”
  - hour=”4,9–17,22”

Calendar-Based Timer Expressions

- Rangos de valores
  - Usamos el carácter “–”
  - Los miembros no pueden ser wildcards, listas u otros rangos
  - Un rango de la forma x–x es equivalente al valor x
  - Un rango de la forma x–y, donde x es menor que y, incluye todos los valores desde x a y
  - Un rango de la forma x–y, donde y es mayor que x, incluye los valores desde x al máximo valor, y luego desde el mínimo valor al y
- Ejemplos
  - hour=“9–17“, dayOfWeek=“5–1”, dayOfMonth=”25–5”, dayOfMonth=“25–Last,1–5”

Calendar-Based Timer Expressions

- Intervalos
  - Usamos el carácter “/” para definir un intervalo
  - Restringe un atributo a un valor inicial seguido de un intervalo regular comenzando en dicho valor inicial
  - Para la expresión x/y, x representa el valor inicial, y representa el intervalo
  - En lugar de x podemos usar *, en este caso, x es igual a 0
  - Los intervalos solo pueden definirse para horas, minutos y segundos
- Ejemplos
  - minute=“*/10” equivale a minute= “0,10,20,30,40,50”
  - hour=“12/2”

Programmatic Timers

- Cuando un timer programático expira, el container invoca el método anotado con @Timeout, localizado en la clase de implementación del bean
- El método @Timeout contiene la lógica de negocio que maneja el evento temporizado
- Estos métodos …
  - Deben retornar **void**
  - No deben recibir parámetros o recibir un parámetro de tipo javax.ejb.Timer
  - No pueden propagar Application Exceptions

Programmatic Timers

```
@Timeout
public void timeout(Timer timer) {
    System.out.println("TimerBean: timeout occurred");
}
```

Programmatic Timers

- Para crear un timer, se invoca uno de los métodos “create” de la interfaz TimerService
- Estos métodos permiten crear timers basados en calendarios, para intervalos fijos, o para un único punto de tiempo
- Para timers basados en intervalos o en puntos fijos de tiempo, la expiración del timer se expresa en milisegundos
- Podemos pasar los milisegundos o podemos crear un objeto java.util.Date para indicar la fecha de expiración del timer

Programmatic Timers

Expira en un minuto

```
long duration = 6000;
Timer timer =
    timerService.createSingleActionTimer(duration, new TimerConfig());
```

Expira en la fecha dada

```
SimpleDateFormatter formatter =
    new SimpleDateFormatter("MM/dd/yyyy 'at' HH:mm");
Date date = formatter.parse("05/01/2010 at 12:05");
Timer timer = timerService.createSingleActionTimer(date, new TimerConfig());
```

Programmatic Timers

- Para las expresiones basadas en calendarios, debemos pasar al método de creación del timer, una instancia de **javax.ejb.ScheduleExpression**
- Usamos el método TimerService.createCalendarTimer

```
ScheduleExpression schedule = new ScheduleExpression();
schedule.dayOfWeek("Mon");
schedule.hour("12-17, 23");
Timer timer = timerService.createCalendarTimer(schedule);
```

Programmatic Timers

- Los timers son persistentes por defecto
- Cuando el servidor se apaga o se cae, al reiniciarlo los timers se reactivaran
- Si el timer expira mientras el servidor esta apagado, cuando el mismo reinicie se invocara inmediatamente al método @Timeout
- Para crear un timer no persistente, debemos crear una instancia de TimerConfig, pasarla como parámetro al timer, pero debemos invocar
- TimerConfig.setPersistent(false)

Programmatic Timers

- Es importante tener presente que los timers, si bien tienen precisión de milisegundos, la llamada al método de callback no tiene porque ocurrir con la precisión del milisegundo
- Esto es porque los timers no están pensados para aplicaciones de tiempo real
- Es típicamente usado para aplicaciones de negocio, en las que la precisión se mide en días, horas, meses, etc.

Automatic Timers

- Cuando se realiza un deploy exitoso, en caso de existir métodos anotados con @Schedule o @Schedules, el container crea los timers automáticos
- Se pueden tener múltiples métodos anotados con @Schedule/s, a diferencia de @Timeout
- El container marca el método como un método de timeout, de acuerdo a la expiración indicada por la anotación @Schedule

```
@Schedule(dayOfWeek="Sun", hour="0")
public void cleanupWeekData() { ... }
```

Automatic Timers

- @Schedule soporta atributos para representar
  - **calendar expressions**: utilizando los atributos antes vistos
  - **persistent**: Indica si el timer automático debe sobrevivir caídas del servidor. Por defecto, este tipo de timers son persistentes
  - **info**: Permite establecer información descriptiva del timer
  - **timezone**: Permite indicar que el timer esta asociado con una determinada zona de tiempo. Esto permite alterar las expresiones de tiempo, determinando contra que timezone ejecutan. Si no se especifica, se utiliza la del servidor de aplicaciones

Automatic Timers

- @Schedules permite registrar múltiples expresiones de calendario relacionadas con un método especifico

```
@Schedules ({
    @Schedule(dayOfMonth="Last"),
    @Schedule(dayOfWeek="Fri", hour="23")
})
public void doPeriodicCleanup() { ... }
```

Cancelando timers

- Los timers se cancelan por alguno de estos motivos
  - Cuando un timer asociado a un evento de única vez (en una fecha especifica) expira. Se invoca el método de timeout y se cancela el timer
  - Cuando se invoca el método cancel sobre la instancia del Timer. Los timers pueden ser localizados a través de los métodos de búsqueda en la interfaz TimerService
- Si cancelamos un timer ya cancelado, se propaga una excepción de tipo **javax.ejb.NoSuchObjectLocalException**

Salvando timers

- Si queremos salvar un timer para utilizarlo en el futuro, podemos usar el TimerHandle asociado
- Sobre un objeto Timer, invocamos el método getHandle, para obtener un objeto TimerHandler
  - Este es serializable, por lo que puede ser salvado en una base de datos (por ejemplo)
- Para obtener el Timer nuevamente, sobre el TimerHandle invocamos el método getTimer
- Los TimerHandle solo pueden ser usados localmente
  - No pueden ser pasados remotamente ni a través de web services

Información del timer

- Sobre una instancia de Timer, podemos además invocar estos métodos
  - public long getTimeRemaining();
  - public java.util.Date getNextTimeout();
  - public java.io.Serializable getInfo();
- getInfo() devuelve el objeto pasado como parámetro en la creación del Timer
- Para obtener la colección de timers asociados con un bean, invocamos sobre un TimerService inyectado, el método getTimers()

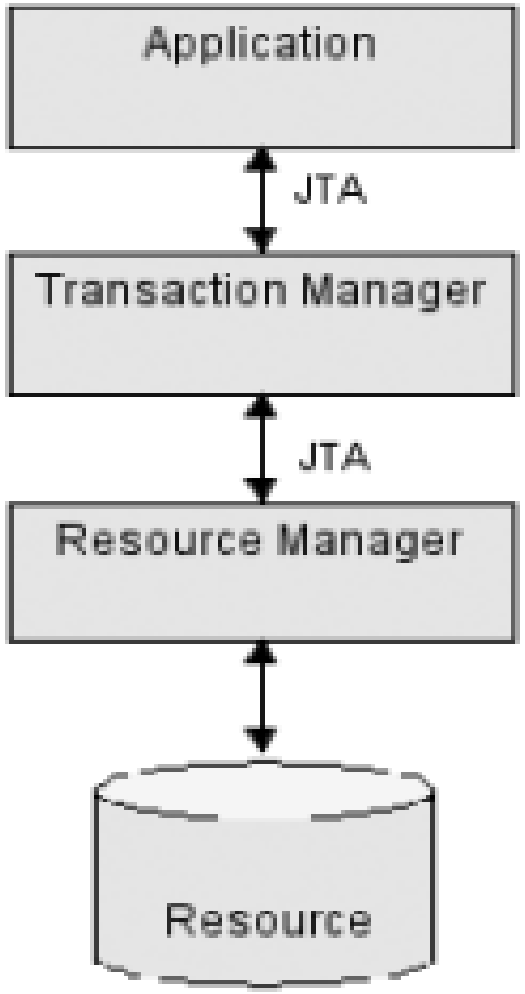
Transacciones y Timers

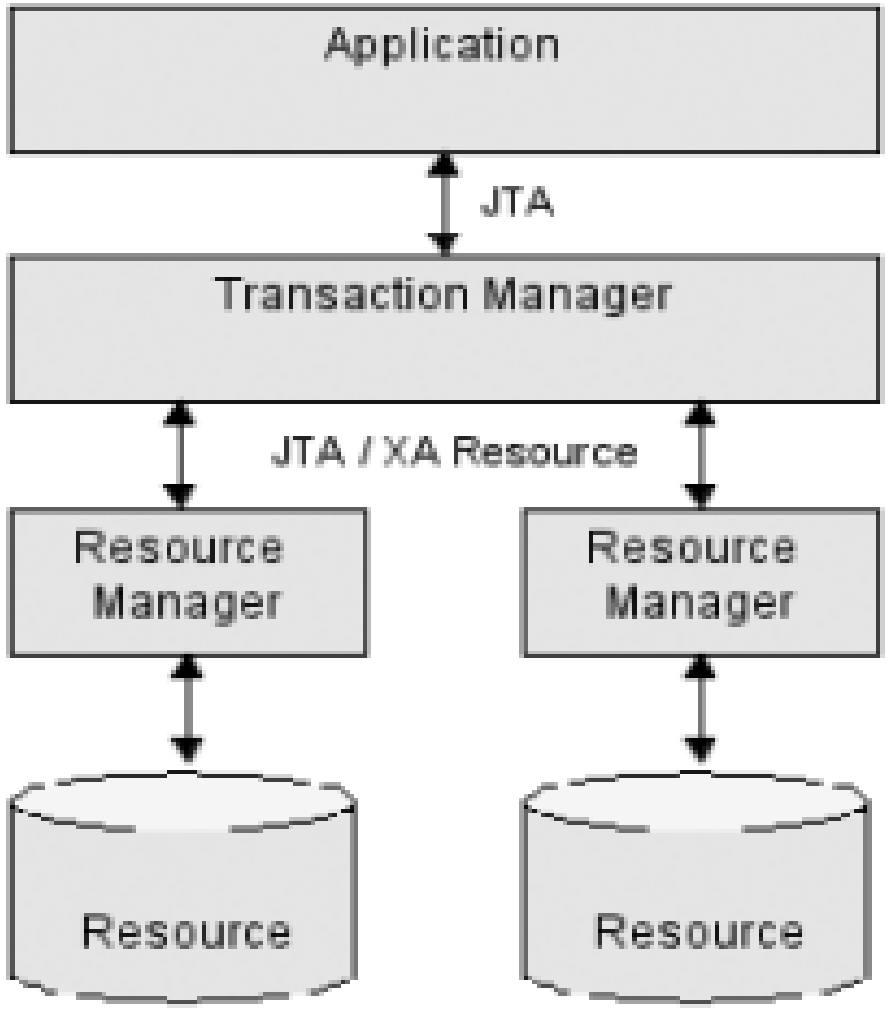
- Si los timers son creados dentro de una transacción, y esta transacción hace rollback, entonces la creación del timer también se deshace
- También si un timer es cancelado dentro de una transacción, y la misma hace rollback, entonces la cancelación también se deshace
- La duración del timer no se ve afectada, como si el timer nunca hubiese sido cancelado

Transacciones

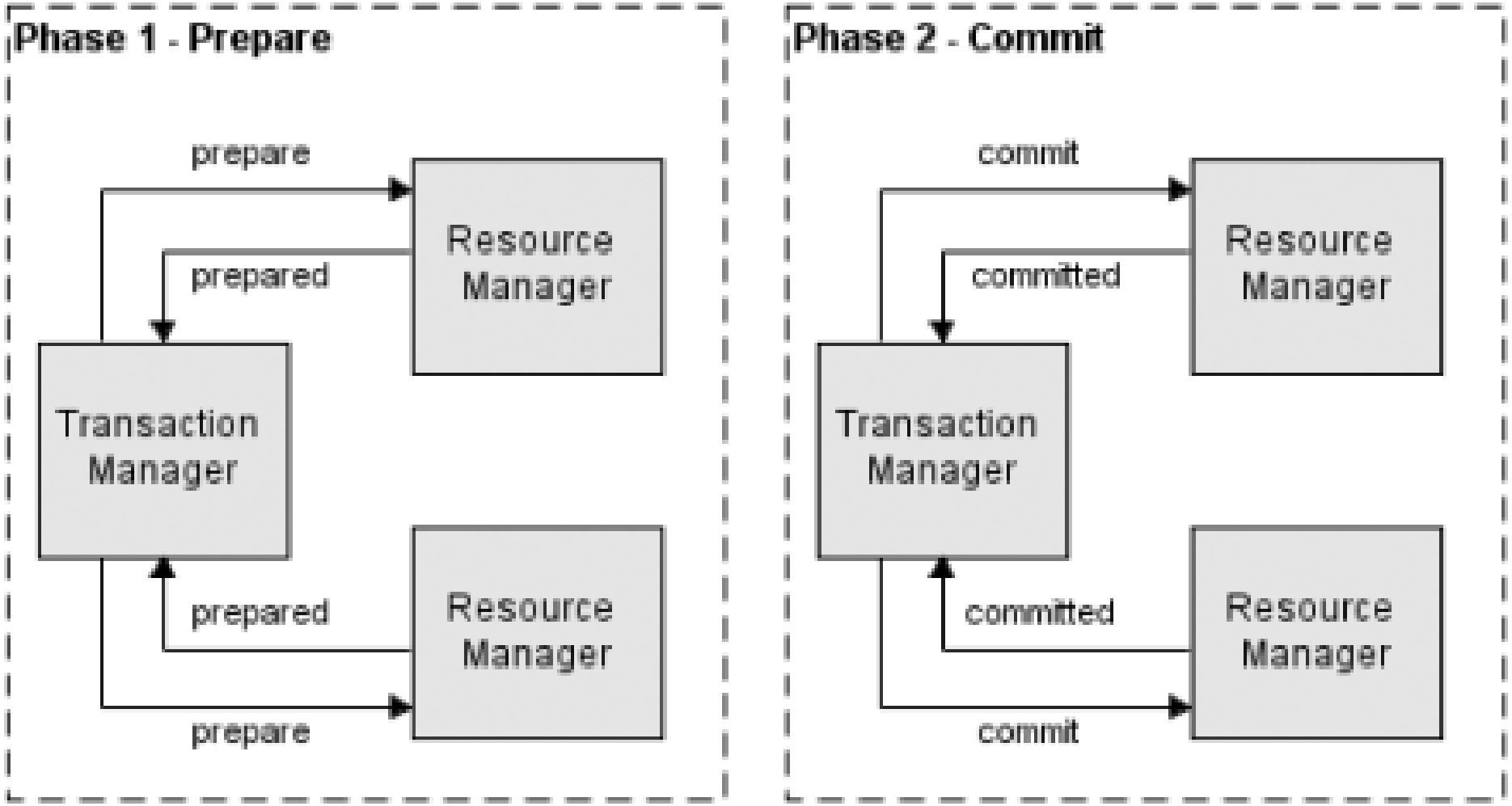
- Una transacción es utilizada para asegurar que los datos manejados por una aplicación, se mantienen en estado consistente
- Representa un grupo de operaciones, que deben ser realizadas como una unidad
- Comúnmente se le denominan Unit Of Work
- Las transacciones siguen las propiedades denominadas ACID

Local vs Distributed Transactions





Two Phase Commit



Transacciones en EJB

- Como muchos otros servicios en los EJBs, cuando trabajamos con un EJB, no debemos preocuparnos del transaction manager o de los resource managers
- El container se encarga de todo esto por nosotros
- Las APIs dentro del container, abstraen la problemática del manejo transaccional
  - Java Transaction Service y Java Transaction API

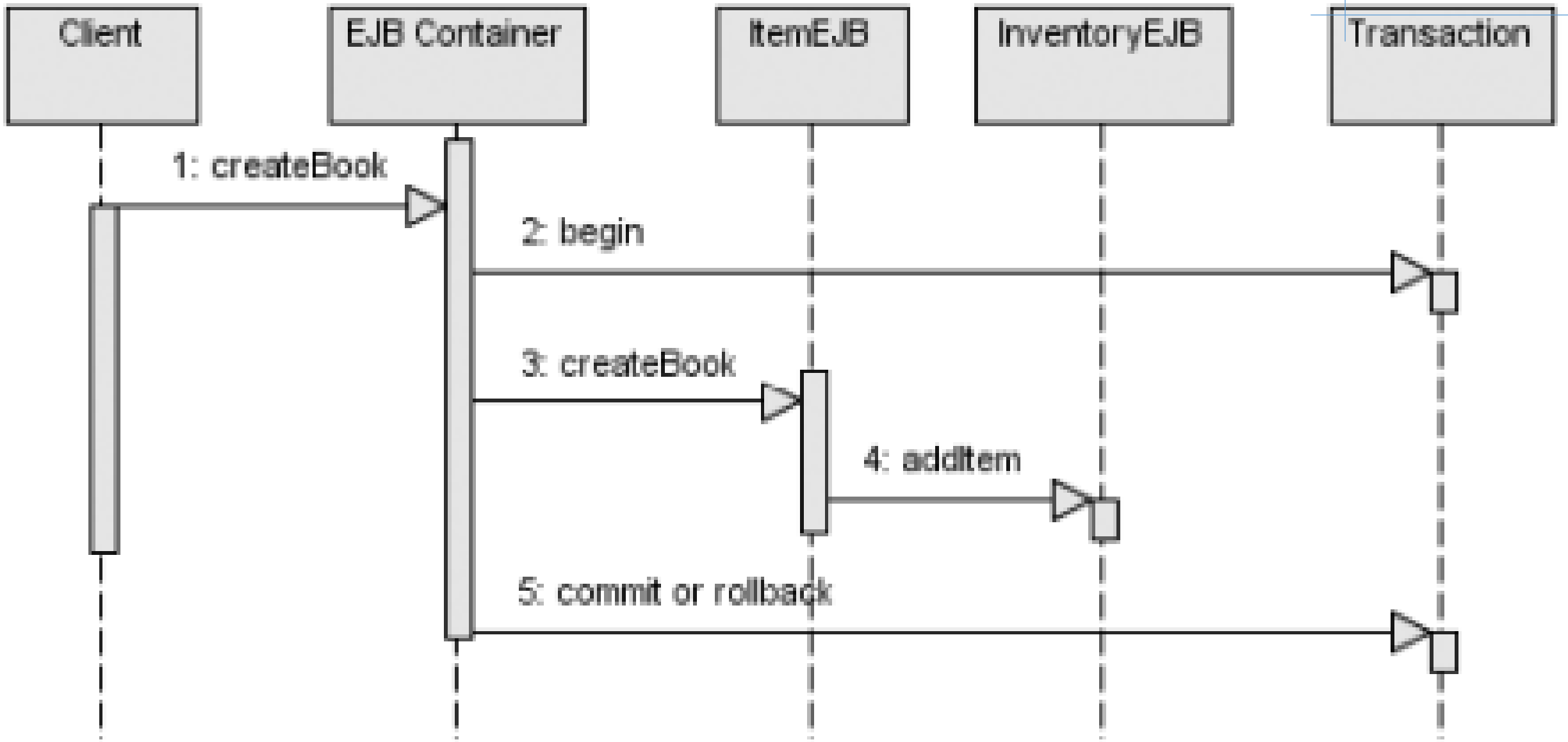
Transacciones en EJB

- Uno de los aspectos mas importantes del manejo de transacciones en los EJB, tiene que ver con el demarcado de la transacción
  - Esto es, indicar el punto del programa donde comienza la transacción y donde termina
- Asimismo, otro de los aspectos importantes a manejar, es indicar como se propaga una transacción cuando invocamos uno u otro método de un EJB
- Tenemos dos tipos de demarcado transaccional
  - Container Managed Transactions
  - Bean Managed Transactions

Container Managed Transactions

- Este manejo de las transacciones es totalmente declarativo
- En base a metadatos (anotaciones) en el código, podemos indicarle al container cuando este debe arrancar y terminar una transacción
- El container provee este servicio a Session Beans (de cualquier tipo) y a los Message Driven Beans
- El contexto transaccional del cliente no se propaga con llamadas a métodos asíncronas

```
@Stateless
public class ItemEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;
        @EJB
        private InventoryEJB inventory;
        public List<Book> findBooks() {
            >>    Query query = em.createNamedQuery("findAllBooks");
                  return query.getResultList();
        }
        public Book createBook(Book book) {
            >>    em.persist(book);
            >>    inventory.addItem(book);
                  return book;
        }
}
```

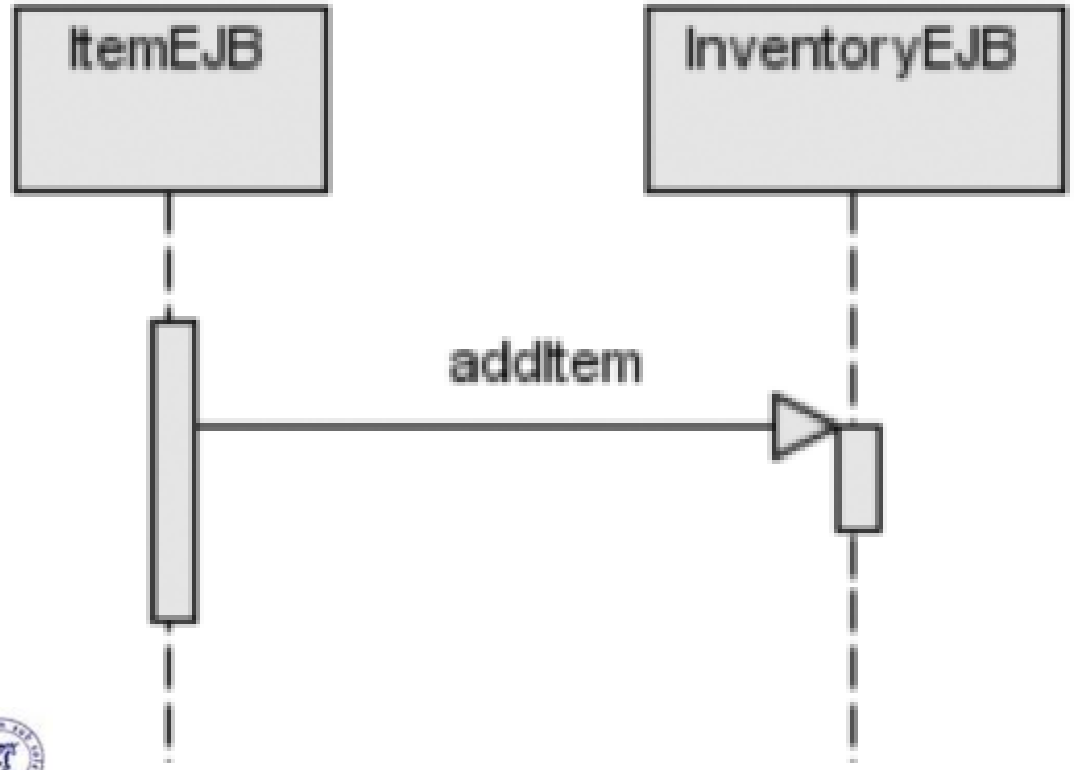

Client
EJB Container
ItemEJB
InventoryEJB
Transaction
1: createBook
2: begin
3: createBook
4: addItem
5: commit or rollback

Container Managed Transactions

- En el ejemplo anterior, no se necesita ninguna anotación especial para indicar que debemos manejar transacciones en el código
- Al utilizar configuración por excepción, se aplican los valores por defecto para el manejo transaccional
- Esto es:
  - Usar CMT
  - Aplicar el nivel REQUIRED a cada método

Container Managed Transactions

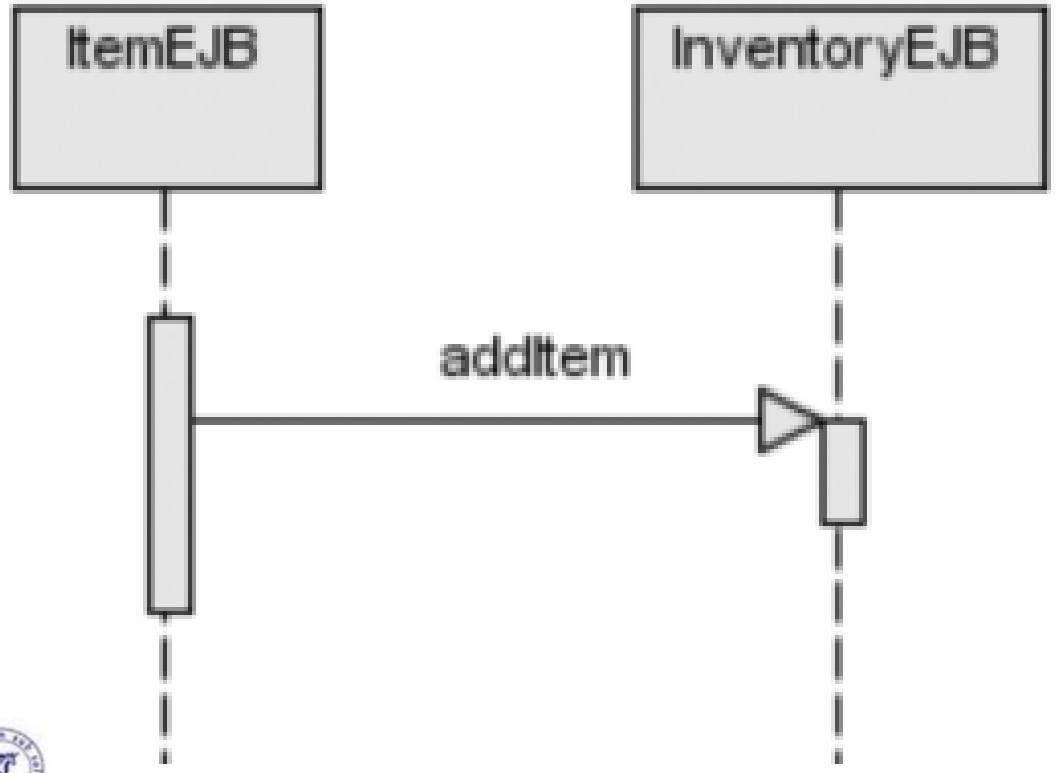
- Por ejemplo, si el método createBook() NO ESTA dentro de una transacción



REQUIRED	New transaction
REQUIRES_NEW	New transaction
SUPPORTS	No transaction
MANDATORY	Exception
NOT_SUPPORTED	No transaction
NEVER	No transaction

Container Managed Transactions

- Por ejemplo, si el método createBook() esta dentro de una transacción



REQUIRED	Caller's transaction
REQUIRES_NEW	New transaction
SUPPORTS	Caller's transaction
MANDATORY	Caller's transaction
NOT_SUPPORTED	No transaction
NEVER	Exception

@TransactionAttribute

- REQUIRED
  - Indica que el método siempre debe ser invocado dentro de una transacción
  - Si la invocación viene de un cliente no transaccional, el container inicia una nueva transacción
  - Sino, ejecuta en la transacción del cliente
- REQUIRES_NEW
  - El container siempre crea una nueva transacción al invocar un método
  - El éxito o no de esta nueva transacción, no afecta a la posible transacción iniciada por el cliente

@TransactionAttribute

- SUPPORTS
  - Si el cliente viene con una transacción, se utiliza esta
  - En caso contrario, se ejecuta el método sin transacciones
- MANDATORY
  - Indica que el método siempre debe ser invocado desde un cliente que haya iniciado una transacción
  - Si esto no es asi, se propaga una excepción de tipo javax.ejb.EJBTransactionRequiredException

@TransactionAttribute

- NOT_SUPPORTED
  - El método del EJB no puede ser invocado desde un contexto transaccional
  - Si el cliente viene con una transacción, se pone en pausa esta y luego se invoca el método
- NEVER
  - El método no debe ser invocado desde un cliente transaccional
  - Si el cliente invoca desde un contexto transaccional, entonces se propaga la siguiente excepción javax.ejb.EJBException

```
@Stateless
@TransactionAttribute(TransactionAttributeType.SUPPORTS)   <<
public class ItemEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;
        @EJB
        private InventoryEJB inventory;
        public List<Book> findBooks() {
                Query query = em.createNamedQuery("findAllBooks");
                return query.getResultList();
        }
    >>  @TransactionAttribute(TransactionAttributeType.REQUIRED)
        public Book createBook(Book book) {
                em.persist(book);
                inventory.addItem(book);
                return book;
        }
```

CMT y Rollbacks

- En el caso de CMT, no podemos hacer rollback explícitamente, ya que la demarcación es automática, al comienzo y fin del método
- Sin embargo, podemos decirle al container que a la transacción actual se le debe hacer un rollback
- Usamos el método setRollbackOnly(), del SessionContext, el cual puede ser inyectado a traves de @Resource

```
@Stateless
public class InventoryEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;

        @Resource
        private SessionContext ctx;

        public void oneItemSold(Item item) {
                em.merge(item);
                item.decreaseAvailableStock();
                sendShippingMessage();
                if (inventoryLevel(item) == 0)
                        ctx.setRollbackOnly();
        }
```

```
@Stateless
public class InventoryEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;

        @Resource
        private SessionContext ctx;

        public void oneItemSold(Item item) {
                em.merge(item);
                item.decreaseAvailableStock();
                sendShippingMessage();
                if (inventoryLevel(item) == 0)
                        ctx.setRollbackOnly();
        }
```

CMT y Rollbacks

- De la misma forma, debido que la transacción no se puede deshacer inmediatamente, tenemos en el SessionContext, el método **getRollbackOnly()**
- Este devuelve un booleano que indica si a la transacción actual se le aplicara un rollback
- De esta forma, podemos evitar realizar tareas innecesarias, costosas que serán deshechas al terminar el método

Excepciones

- Propagar una excepción dentro de EJB que utiliza transacciones, no siempre deshace la transacción
- Esto dependerá del tipo de excepción y/o los metadatos asociados a dicha excepción
- Tenemos 2 tipos de excepciones
  - Application Exceptions
  - System Exceptions

Excepciones

- Application Exceptions
  - Son excepciones que representan problemas de la lógica de negocio
  - Por ejemplo, “Inventario es escaso” o “No hay crédito”
  - Por si solas, este tipo de excepciones no hace el rollback de la transacción
  - Estas excepciones deben extender a java.lang.Exception

Excepciones

- System Exceptions
  - Son excepciones causas por problemas a nivel del sistema, pero no de la aplicación o el negocio
  - Por ejemplo, errores en búsquedas JNDI, errores al acceder a la base de datos, errores en la JVM, Null Pointer Exceptions, etc.
  - Debe ser una subclase de RuntimeException o java.rmi.RemoteException
  - Propagar este tipo de excepciones, hace que la transacción sea marcada para rollback

```
@Stateless
public class InventoryEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;
        public void oneItemSold(Item item)

        throws InventoryLevelTooLowException{
                em.merge(item);
                item.decreaseAvailableStock();
                sendShippingMessage();
                if (inventoryLevel(item) == 0)
    >>                  throw new InventoryLevelTooLowException();
        }
}
```

@ApplicationException

- Podemos usar esta anotación para indicar si la Application Exception debe o no deshacer la transacción actual

```
@ApplicationException(rollback = true)
public class InventoryLevelTooLowException extends Exception {
        public InventoryLevelTooLowException() {}
        public InventoryLevelTooLowException(String message) {
                super(message);
        }
}
```

Bean Managed Transactions

- Debemos usar la anotación TransactionManagement aplicada a la clase del bean

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

Bean Managed Transactions

- La interfaz principal para manipular la transacción en el bean es UserTransaction
- Esta puede ser inyectada a través de @Resource en la clase del bean
- Provee los métodos:
  - begin, commit, rollback
  - setRollbackOnly
  - getStatus, setTransactionTimeout
- Un transacción nunca es propagada dentro de un método en un bean con este tipo de manejo transaccional

```
@Stateless
public class InventoryEJB {
        @PersistenceContext(unitName = "LIBRARYDS")
        private EntityManager em;
        @Resource
    >>  private UserTransaction ut;
        public void oneItemSold(Item item) {
                try {
    >>                  ut.begin();
                        em.merge(item);
                        item.decreaseAvailableStock();
                        sendShippingMessage();
                        if (inventoryLevel(item) == 0)
    >>                          ut.rollback();
                        else
    >>                          ut.commit();
                } catch (Exception e) {
    >>                  ut.rollback();
                }
                sendInventoryAlert();
        }
```