Enterprise Java Beans 3.1

Summary of EJB specifications for EJB Certification

0 Contents

1 Client versus Client View

The client of a Session Bean may be:

- a Local client,
- a Remote client,
- a Web service client.

A **client** never directly accesses instances of the Session Bean's class, but uses the session bean's **client view**.

The client view of a session object is independent of the implementation of the session bean and the container.

1.1.1 Web Service Client

- The only EJB's that can expose a web service and can have web service clients are:
  - Stateless Session Bean (SLSB)
  - Singleton Session Bean (SSB)
- A web service client accesses a session bean through the web service client view
- A web service client is location independent and remotable
- A web service client may be a Java client and/or a non-Java client
- It is possible to provide a web service client view in addition to other client views for an enterprise bean
- A web service client view may be initially defined by a WSDL document and then mapped to a web service endpoint, or
- An existing session bean may be adapted to provide a web service client view.
- In the case of Java clients, the WS-endpoint is accessed as a JAX-WS (or deprecated JAX-RPC) service endpoint using the JAX-WS (or JAX-RPC) client view APIs.
- A reference to a JAX-WS-endpoint is done with the @WebServiceRef annotation
- A web service endpoint (JAX-RPC) gets the MessageContext interface by means of the SessionContext.*getMessageContext()* method.
- A web service endpoint (JAX-WS) should use the WebServiceContext.*getMessageContext()* method to obtain the MessageContext interface.

1.1.2 Local Client

- A local client is a client that is collocated in the same JVM.
- Access to an enterprise bean through the local client view is only required to be supported for local clients packaged within the same application as the enterprise bean that provides the local client view.
- A local client accesses a session bean through the bean's local business interface or through a no-interface client view representing all the public methods of the bean class.
- The arguments and results of the methods of the local business interface or local no-interface client view are **passed by reference**
- It gives lightweight access to an EJB: in other words **fine-grained** component access (e.g. more interactions between the client and bean)

1.1.3 Remote Client

- The remote client view of an enterprise bean is location independent. (but if you use it when the bean and the client are in the same application, you will pay the overhead penalty)
- The arguments and results of the methods of the remote business interface are **passed by value**.
- The objects that are passed as parameters on remote calls must be serializable.
- Potentially more expensive as it involves overhead: network latency, larger client and server software stacks, argument copying, etc.
- Typically used for relatively **coarse-grained** component access (e.g. few interactions between the client and bean)

2 Session Bean

2.1 General aspects

- A Session Bean, in general, does not survive a crash and restart of the container.
- A Session bean that does not make use of the Java Persistence API (JPA) must explicitly manage cached database data.
- A Session bean can be invoked synchronously or asynchronously.
- A Session bean declared in deployment descriptor elements override corresponding annotated elements
- Business interface methods should be public and not final or static.
- The business interface method name cannot start with “ejb” (because of previous EJB-versions)
- As a general rule, callback methods should not be exposed as business methods (not public)
- The same business interface cannot be both a local and a remote business interface of the bean.
- A Session bean implements zero (no-interface view) or more business interfaces.

2.1.1 Creation of bean

1. The container calls the bean class’s new Instance method
2. The container performs any dependency injection as specified by annotations on the bean class or by the deployment descriptor.
3. The container calls the PostConstruct lifecycle callback interceptor method for the bean

2.1.2 Business Interface

- The business interface is allowed to have superinterfaces.
- The interface can be annotated on with @Local or @Remote on the interface class, or annotated with @Local (<name>.class) or @Remote (<name>.class) on the bean class

```
@Local
public interface EchoBeanLocalInterface extends Echo {
   ...
}

@Stateless
public class EchoBean implements EchoBeanLocalInterface {
   ...
}
```

```
public interface EchoBeanLocalInterface extends Echo {
   ...
}

@Stateless
```

```
@Local(EchoBeanLocalInterface.class)
public class EchoBean implements EchoBeanLocalInterface {
   ...
}
```

- The business interface cannot be Local and Remote at the same time.
- A Session Bean is allowed to expose more than one interface.

```
@Stateless
@Local( {EchoBeanLocalInterface.class, AnotherLocalInterface.class} )
@Remote( {EchoBeanRemoteInterface.class, AnotherRemoteInterface.class} )
public class EchoBean implements EchoBeanLocalInterface,
      AnotherLocalInterface, EchoBeanRemoteInterface, AnotherRemoteInterface
{
   ...
}
```

- If a bean class has more than one interface, any business interface of the bean class must be explicitly designated as a business interface of the bean by means of @Local or @Remote.
- If the interface is a remote business interface, its methods must not expose local interface types, timers or timer handles.
- If the Session bean class implements a single interface, that interface is assumed to be the local business interface of the bean.
    - That interface is Local (if not specified elsewhere as being Remote)

```
// Although not specified, this Interface is the local business interface
public interface EchoBeanSingleInterface extends Echo {
   ...
}

@Stateless
public class EchoBean implements EchoBeanSingleInterface {
   ...
}
```

- If the interface class is defined on the bean class by the usage of @Local or @Remote, the Session bean does not have to implement all the methods of the interface.

```
public interface EchoBeanSingleInterface extends Echo {
   public String echo (String phraseToEcho);
   public String anotherEcho (String anotherPhraseToEcho);
}

@Stateless
@Remote(EchoBeanSingleInterface.class)
```

```
public class EchoBean {
    public String echo (String phraseToEcho) {
           return "echo: " + phraseToEcho;
    }
    // although the EchoBeanSingleInterface is annotated on the bean
    // class it doesn't have to implement the method anotherEcho()
    // this is because the class doesn't have an implements declaration
}
```

2.1.3 @LocalBean - No interface view

- If the bean does not expose any other client views (Local, Remote, No-Interface, Web Service) and its implements clause is empty, the bean defines a no-interface view.

```
@Stateless
public class NoInterfaceBean {
}
```

- If the bean exposes at least one other client view, the bean designates that it exposes a no-interface view by means of the @LocalBean annotation on the bean class or in the deployment descriptor.

```
@Stateless
@Remote(EchoBeanRemote.class)
@LocalBean
public class EchoBean implements EchoBeanRemote {
}
```

2.1.4 Obtaining access to a session bean

A client can obtain access to session bean (and use the methods defined in the business interface, or use the public methods by a no-interface view) through dependency injection or lookup in the JNDI namespace.

2.1.5 Example EchoBean

I have an ear (*echoear)* project containing a web project (*echoweb*) and an ejb project (*echobean*). This is the ear deployment descriptor *application.xml.*

```
<application>
  <display-name>echoear</display-name>
  <module>
    <ejb>echobean.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>echoweb.war</web-uri>
      <context-root>lab</context-root>
    </web>
```

```
  </module>
</application>
```

In the echobean project:

```
public interface Echo {
       public String echo (String phraseToEcho);
}
```

```
@Remote
public interface EchoBeanRemote extends Echo {
}
```

```
@Stateless
public class EchoBean implements EchoBeanRemote {
       public String echo (String phraseToEcho) {
              return "echo: " + phraseToEcho;
       }
}
```

```
package nl.notes.ejb;
// import statements left out
@Stateless
@LocalBean
public class WorkingBean {

@EJB (name="ejb/EchoBean")
EchoBeanRemote echo;
@Resource
SessionContext context;

public String talkViaDependencyInjection (String ja) {
   return echo.echo(ja);
}
public String talkViaSessionContextLookup (String ja) {
   EchoBeanRemote e = (EchoBeanRemote) context.lookup("ejb/EchoBean");
   return e.echo(ja);
}
public String talkViaInitialNamingContextLookup (String ja) {
   InitialContext icontext;
```

```
   String result="";
   try {
       icontext = new InitialContext();
       EchoBeanRemote i = (EchoBeanRemote) icontext.lookup
                                    ("java:global/echoear/echobean/EchoBean");
       result = i.echo(ja);
       } catch (NamingException e) { //do nothing for the sake of the example
       }
       return result;
    }
}
}
```

In the echoweb project:

```
package nl.notes.servlet;
// import statements left out
import nl.notes.ejb.WorkingBean;

@WebServlet("/EchoServlet")
public class EchoServlet extends HttpServlet {
   @EJB WorkingBean work;
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                            throws ServletException, IOException {

    ServletOutputStream out = response.getOutputStream();
    out.print("<html><body>");

    out.print(work.talkViaDependencyInjection(" DependencyInjection <br>" ));
    out.print(work.talkViaSessionContextLookup(" JNDI SessionContext <br>" ));
    out.print(work.talkViaInitialNamingContextLookup(" JNDI initialContext "));

    out.print("</body></html>");
   }
}
```

Executing the example after setting up the Application Server:

```
URL: http://localhost:8080/lab/EchoServlet
Output:
DependencyInjection
JNDI SessionContext
JNDI initialContext
```

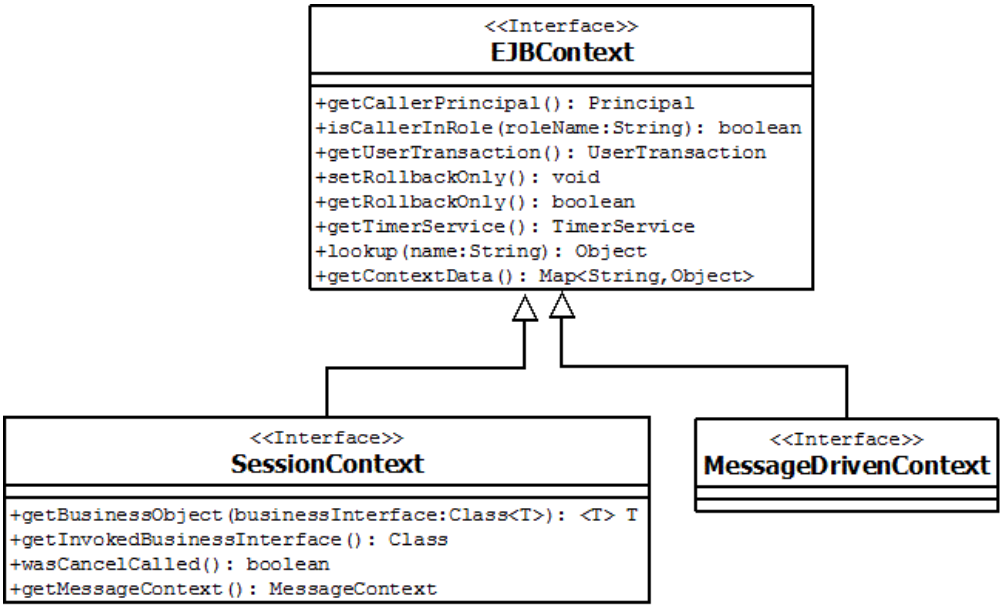
2.1.6 Concurrency

The container serializes calls to each stateful and stateless session bean instance. Most containers will support multiple instances of a session bean to deal with concurrent calls. However each instance sees only a serialized sequence of method calls.

A singleton session bean is intended to be shared and supports concurrent access. The developer decides whether the bean will use container managed or bean managed concurrency.

- Container managed concurrency is the default.
- A Singleton bean cannot use both container managed and bean managed.

2.1.7 SessionContext and EJBContext interfaces



2.1.7.1 EJBContext Interface

Return	Method	throws Exception
Principal	getCallerPrincipal() Principal that identifies the caller	IllegalStateException when invoked from MessageDriven Bean/Session Bean/Singleton Bean (in the PostConstruct, PreDestroy lifecycle callback methods)
boolean	isCallerInRole(String roleName) true if caller has a given security role	IllegalStateException MessageDriven Bean/Session Bean/Singleton Bean (in the PostConstruct, PreDestroy lifecycle callback methods)
UserTransaction	getUserTransaction() get transaction interface for BM-EJBs	IllegalStateException container managed
void	setRollbackOnly() mark current transaction for rollback	IllegalStateException bean managed, or transaction attribute method is SUPPORTS (and no transaction), or NOT_SUPPORTED, or NEVER

boolean	getRollbackOnly() true if transaction has been marked for rollback (false if transaction has been rolled back already)	IllegalStateException bean managed, or transaction attribute method is SUPPORTS (and no transaction), or NOT_SUPPORTED, or NEVER
TimerService	getTimerService() get access to the EJB Timer Service	IllegalStateException Stateful Session bean
Object	lookup(String name) lookup a resource within the "java:" namespace relative to "java:comp/env/"	IllegalArgumentException no matching entry
Map <String, Object>	getContextData() returns the context data associated with this invocation or lifecycle callback	

2.1.7.2 SessionContext Interface

Return	Method	throws Exception
<T> T	getBusinessObject(Class<T> businessInterface) obtain a reference to this object (business interface view or no-interface view). Equivalent to "this" in a pojo	IllegalStateException no business interface or no-interface view
Class	getInvokedBusinessInterface() obtain the business interface or no-interface view type through which the current business method invocation was made	IllegalStateException not invoked through a business interface or no-interface view
boolean	wasCancelCalled() true if a client invoked the *cancel*() method on the client Future object corresponding to the currently executing asynchronous business method.	IllegalStateException not invoked from within an asynchronous business method
MessageContext	getMessageContext() obtain a reference to the JAX-RPC MessageContext. in JAX-WS use WebServiceContext. *getMessageContext()*	IllegalStateException not invoked from web service endpoint

2.1.8 @Asynchronous

- The **@Asynchronous** annotation is used to designate which business methods are asynchronous.
- Only for Session Bean's (not Message Driven Bean) local business view, no-interface view and remote business view.
- If @Asynchronous is applied at the class level, all business methods declared on that specific class are asynchronous.
- The client transaction context does not propagate with an asynchronous method invocation.
- The client security context propagates with an asynchronous method invocation.

- If the client receives an EJBException immediately, the container has problems allocating the internal resources required to support the asynchronous method.
- The result value of an asynchronous invocation is a Future<V> object for which the get() methods return the result value.
- A concrete Future<V> implementation called AsyncResult<V> is provided by the container as a convenience. AsyncResult<V> has a constructor that takes the result value as a parameter.
- Example:

```
@Asynchronous
public Future<Integer> performAddition(Integer a, Integer b) {
   Integer result = a + b;
   return new AsyncResult<Integer>(result);
}
```

- A client can request that an asynchronous invocation should be cancelled by calling the Future<V>.cancel(boolean mayInterruptIfRunning) method (there is no guarantee)
- The boolean *mayInterruptIfRunning* indicates whether the implementer of the asynchronous method will be able to see on the SessionContext if *a cancel()* has been requested. If true, the implementer might use that info to short-circuit the method (see next bullet). Otherwise, the implementer won't be able to see a client's *cancel()* request.
- You can check whether the client has requested cancellation by calling the SessionContext.*wasCancelCalled()* method
- There are actually two ways an asynchronous call can be cancelled:

  1) If the method has not been invoked: by the container

  2) If the method has been invoked: by the implementer of the asynchronous method, short circuiting the method by checking the SessionContext.*wasCancelCalled()* method. This cancel request can be ignored by the implementer.

```
...
@Resource SessionContext myContext;

@Asynchronous
public Future<Integer> performAddition (Integer a, Integer b) {
   Integer result = null;
   // if the client didn't request to cancel this method
   if( myContext.wasCancelCalled()== false ) {
      result = a + b;
   }
   return new AsyncResult<Integer>(result);
}
...
```

- If the asynchronous method has return type void, then no exceptions will be delivered to the client. (so don't use methods that thrown an ApplicationException)

- If the asynchronous method has return type Future<V>, and an exception is thrown during the execution of the method (e.g. asyncMethod is the asynchronous annotated method)
    - asyncMethod.*get()* will result in an ExecutionException
    - ExecutionException.getCause() will return the original exception to the client

2.1.9 Predestroy not called

The following scenarios result in the PreDestroy lifecycle callback interceptor method(s) not being called for an instance:

1. A crash of the EJB container.
2. A system exception thrown from the instance's method to the container.
3. A timeout of client inactivity while the instance is in the passive state. (Stateful SB only)

The application using the session bean should therefore provide some clean up mechanism for the resources that were not cleaned up (or released) by the lack of the PreDestroy calls.

2.1.10 Superclass Session Bean

A session bean class can have a superclass that is a session bean.

- A client view (i.e. business interface or no-interface client view) exposed by a particular session bean is not inherited by a subclass

```
@Stateless
public class A implements Echo { ... }
@Stateless
public class B extends A implements Talk
```

Session bean A exposes local business interface Echo and Session bean B exposes local business interface Talk, but not Echo (unless explicitly put as an implementer)

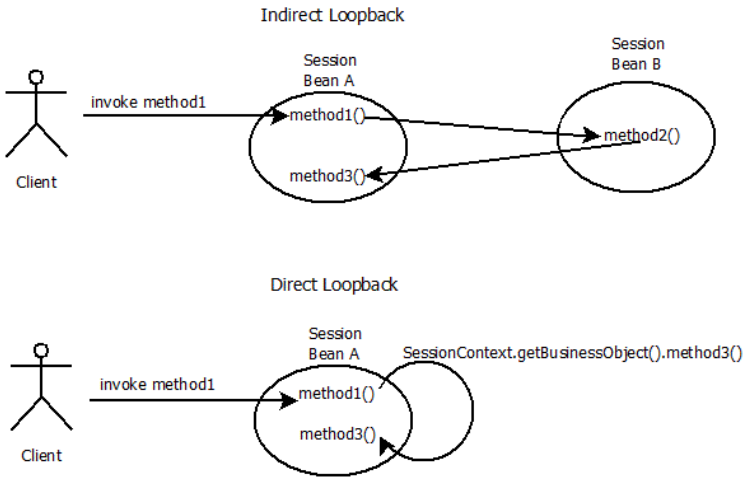
2.1.11 Threading

The container must ensure that only one thread can be executing a Stateless or Stateful Session Bean instance at any time.

By default there will be only one thread executing *the* Singleton Session Bean instance. However the threading can be altered (see 2.4.6 @Lock)

2.1.12 Loopback call (re-entry)

Re-entry is allowed for Singleton Session Beans, but not for Stateful Session Beans and Stateless Session Beans. A loopback call is either done *indirectly* (via another EJB) or *directly* via the getBusinessObject() method.



Note:

- For a Stateless Session Beans, the *getBusinessObject*() returns a reference to another instance, whereas for Stateful Session Beans it returns the reference to the current instance.
- For a Stateful Session Beans any method called on the object returned by getBusinessObject() will result in an IllegalLoopbackException.
- If method3() is called directly from method1() without the SessionContext.getBusinessObject(), there won't be an IllegalLoopbackException.
- We don't have worry when coding about the fact that the instance can be re-entered, because the container takes care of it:
  - for Stateless Session Beans it will grab another instance (avoiding reentrancy)
  - for Stateful Session Beans it will throw an IllegalLoopbackException (blocking reentrancy)

2.2 Stateless Session Bean (SLSB)

The session bean instances contain no conversational state between methods. Any instance can be used for any client.

2.2.1 Equals

All business object references of the same interface type for the same stateless session bean have the same object identity, which is assigned by the container.

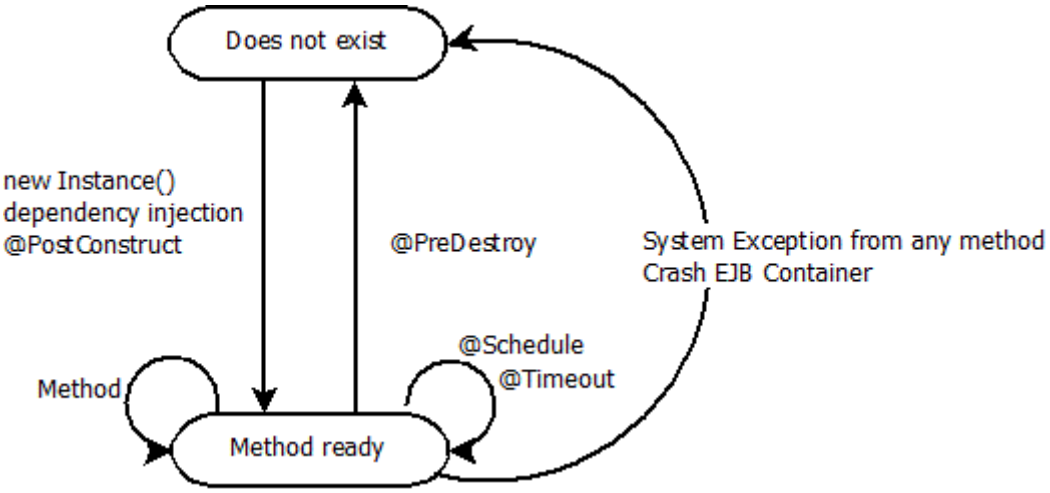
```
@EJB
Echo echo1;
@EJB
Echo echo2;

if (echo1.equals(echo1)) { // this test must return true
}

if (echo1.equals(echo2)) { // this test must return true

}
```

2.2.2 LifeCycle diagram



- N.B.: it is either the method annotated with @Timeout or the ejbTimeout() method when the bean is implementing the TimedObject interface

2.2.3 @PostConstruct, @PreDestroy

The PostConstruct and PreDestroy lifecycle callback interceptor methods execute in an unspecified security context and an unspecified transaction context.

The PostConstruct callback invocations occur before the first business method invocation on the bean.

The PostConstruct and PreDestroy method in a Stateless Session bean has only access to some of the SessionContext methods, JNDI access to java:comp/env, and access to the EntityManagerFactory.

No Access to:

- Another Enterprise bean (Glassfish allows it though, specs say behaviour is undefined)
- Resource manager
- EntityManager
- TimerService and Timer methods

An IllegalStateException is thrown for:

- Security related SessionContext methods
  - getCallerPrincipal(), isCallerInRole()
- Transaction related SessionContext methods
  - getRollbackOnly(), setRollbackOnly(),
- Asynchronous related method
  - wasCancelCalled()
- Client related SessionContext method
  - getInvokedBusinessInterface()
- All TimerService and Timer methods (except *SessionContext.getTimerService()*)

2.3 Stateful Session Bean (SFSB)

The session bean instances contain conversational state which must be retained across methods and transactions.

This conversational state describes the conversation represented by a specific client/session object pair, meaning:

- the instance's field values,
- its associated interceptors and their instance field values,
- the objects that can be reached from these instances' fields

Typically, a session object's conversational state is not written to the database.

The session bean instance conversational state is not transactional. It is not automatically rolled back to its initial state if the transaction in which the object has participated rolls back.

You should use the afterCompletion() notification to manually reset the conversational state in case of a rollback.

The optional SessionBean interface (needed in earlier EJB versions) defines four methods: setSessionContext(), ejbRemove(), ejbPassivate(), and ejbActivate(). If the Session bean implements the optional SessionBean interface and uses the annotations:

- @PreDestroy can only be applied on the *ejbRemove*() method;
- @PostActivate can only be applied on the *ejbActivate*() method;
- @PrePassivate can only be applied on the *ejbPassivate*() method.

2.3.1 Equals

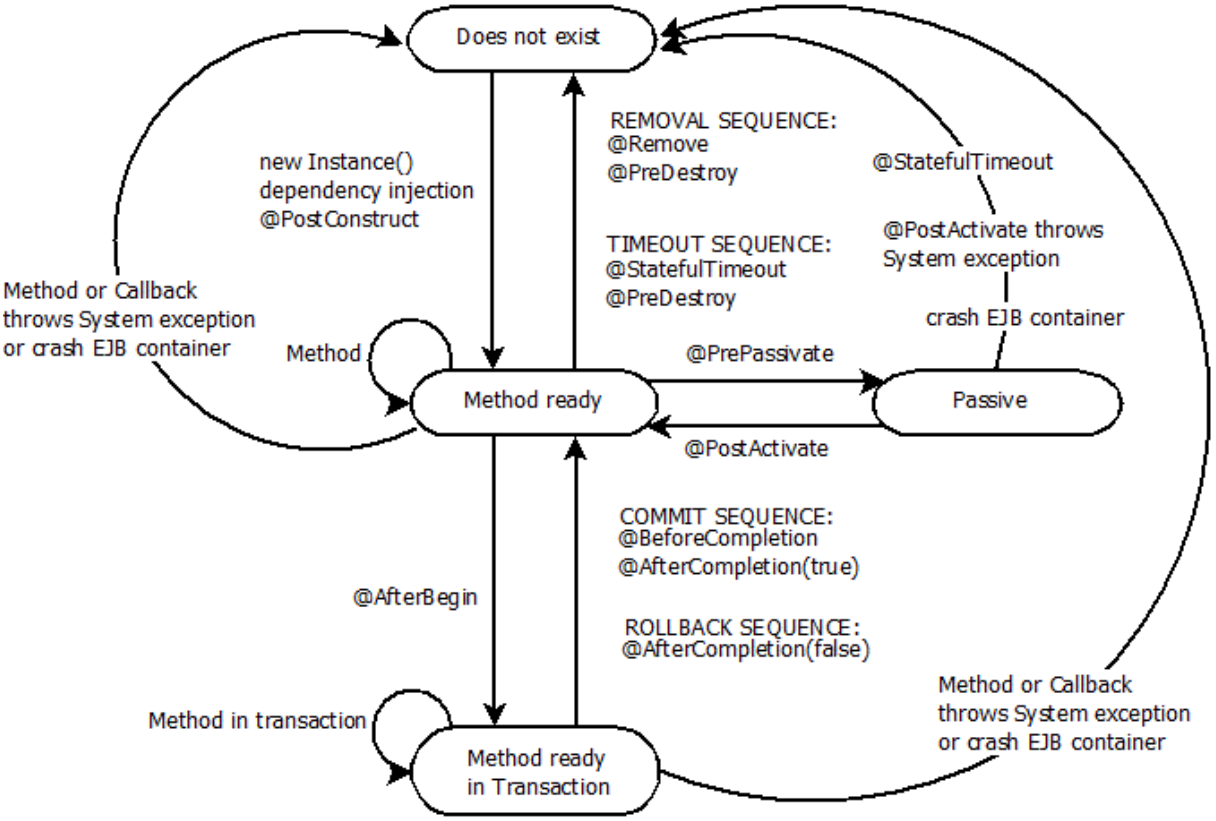
A stateful session object has a unique identity that is assigned by the container at the time the object is created. A client of the stateful session bean business interface can determine if two business interface or no-interface view references refer to the same session bean by use of the equals() method.

```
@EJB
Echo echo1;
@EJB
Echo echo2;

if (echo1.equals(echo1)) { // this test must return true
}

if (echo1.equals(echo2)) { // this test must return false
}
```

2.3.2 LifeCycle diagram



- A transition from one state to the other (not being “Does not exist”) is only successful when no system exception is thrown from the transitioning method(s).
- A system exception thrown from the @PostConstruct (or @AfterBegin) method results in the bean to be discarded before it gets to the “Method ready” (“Method ready in Transaction”) state
- In a transaction it is either the @AfterBegin annotated method or when the SessionSynchronization interface is implemented, the afterBegin() method.

2.3.3 @PostConstruct, @PreDestroy

The PostConstruct and PreDestroy lifecycle callback interceptor methods execute in an unspecified security context and an unspecified transaction context.

The PostConstruct callback invocations occur before the first business method invocation on the bean.

An IllegalStateException is thrown for:

- Transaction related SessionContext methods
  - getRollbackOnly(), setRollbackOnly(),
- Asynchronous related method
  - wasCancelCalled()
- Client related SessionContext method
  - getInvokedBusinessInterface()
- Timer related SessionContext method
  - getTimerService()
- all TimerService and Timer methods.

2.3.4 Passivation and Activation

After the PrePassivate method the instance fields and the fields of its associated interceptors are ready to be serialized by the container.

The following fields are serialized by the container:

- A reference to other EJBs (via business interface or no-interface view)
- A reference to the SessionContext object,
- A reference to the UserTransaction interface.
- A reference to a Timer object.
- A reference to the environment naming context (JNDI)
- A reference to a resource manager connection factory (@Resource).
- A reference to an EntityManager or EntityManagerFactory

2.3.5 @PrePassivate, @PostActivate

A session bean container may need to temporarily transfer the state of an idle stateful session bean instance (SLSB's and SSB's cannot be passivated) to some form of secondary storage. This is called passivation; the reverse mechanism is called activation.

A developer of a stateful session bean must close and open the resources in the PrePassivate() and PostActivate() lifecycle callback interceptor methods.

In general there are two solutions in dealing with non-serializable objects in instance fields:

- Set them to *null* in the PrePassivate method, and re-create them in the PostActivate method
- Mark those fields as *transient* so that they are skipped during serialization, and initialize the values of the transient fields in the PostActivate method (because the container is not required to default them like in standard Java serialization)

The PrePassivate and PostActivate lifecycle callback interceptor methods execute in an unspecified transaction and security context.

2.3.6 @Remove

The Remove method causes the removal of the stateful session bean after the remove method ends successfully.

If the Remove annotation specifies the value of **retainIfException** as **true**, and the Remove method throws an application exception, the instance is not removed

2.3.7 SessionSynchronization Interface

A stateful session bean class can optionally implement the javax.ejb.**SessionSynchronization** interface (methods *afterBegin()*, *afterCompletion()*, *beforeCompletion()*) **or annotate** methods using the individual @AfterBegin, @BeforeCompletion, and @AfterCompletion annotations (N.B. using both is not allowed)

- A session synchronization method can have public, private, protected, or package level access.
- A session synchronization method must not be declared as final or static.
- The return type must be void.
- The @AfterBegin and @BeforeCompletion methods take no arguments; the @AfterCompletion takes a boolean.

- The @AfterBegin occurs before any *@AroundInvoke* method invocation.
- The @BeforeCompletion occurs after all *@AroundInvoke* invocations are finished.

2.3.7.1 @AfterBegin

The afterBegin notification signals a session bean instance that a new transaction has begun. The container invokes this method before the first business method within a transaction (which is not necessarily at the beginning of the transaction). The afterBegin notification is invoked with the transaction context.

2.3.7.2 @BeforeCompletion

The beforeCompletion notification is issued when a session bean instance's client has completed work on its current transaction but prior to committing the resource managers used by the instance. At this time, the instance should write out any database updates it has cached. The instance can cause the transaction to rollback by invoking the *setRollbackOnly()* method on its session context (as this method always runs in a transaction context).

2.3.7.3 @AfterCompletion

The afterCompletion notification signals that the current transaction has completed. A completion status of true indicates that the transaction has committed. A status of false indicates that a rollback has occurred. Since a session bean instance's conversational state is not transactional, it may need to **manually reset** its state if a rollback occurred.

2.3.8 @StatefulTimeout (value=10 , unit = TimeUnit.SECONDS)

Specifies the amount of time a stateful session bean can be idle (not receive any client invocations) before it is eligible for removal by the container. A timeout can be specified using the @StatefulTimeout annotation on the bean class or by using the deployment descriptor element.

- A stateful session bean instance is not eligible for timeout while it is associated with a transaction or during a business method or a callback method.
- A timeout value of -1 indicates that the bean must not be removed due to timeout
- A timeout value of 0 indicates that the bean is immediately eligible for removal after becoming idle
- A timeout unit is optional and defaults to TimeUnit.MINUTES

2.4 Singleton Session Bean (SSB)

A single session bean instance is shared between clients and supports concurrent access.

- Any Singleton instance that successfully completes initialization is explicitly removed by the container during application shutdown.
- Errors occurring during Singleton initialization are considered fatal and will result in removal of the Singleton instance.
- Possible initialization errors:
    - an injection failure,
    - a system exception thrown from a PostConstruct method,
    - a commit failure (container-managed) of a PostConstruct method.
- System exceptions thrown from business methods or callbacks of a Singleton do not result in the removal of the Singleton instance.

- It is legal to store Java EE objects that do not support concurrent access (e.g. Entity Managers, Stateful Session Bean references) within Singleton bean instance state. You should then take care of concurrency issues yourself.
- The container will block concurrent access to the Singleton bean instance until its initialization, including the @PostConstruct lifecycle callback method, has finished.

2.4.1 Equals

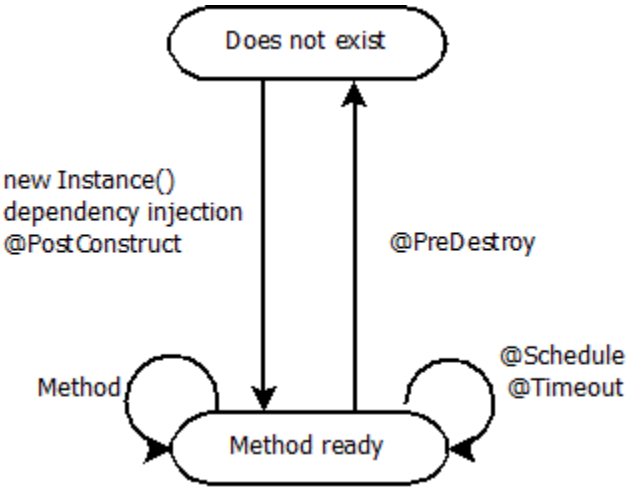
All business object references of the same interface type for the same singleton session bean have the same object identity, which is assigned by the container.

```
@EJB
EchoSingleton echo1;
@EJB
EchoSingleton echo2;

if (echo1.equals(echo1)) { // this test must return true
}

if (echo1.equals(echo2)) { // this test must return true
}
```

2.4.2 LifeCycle diagram



2.4.3 @PostConstruct, @PreDestroy

The PostConstruct and PreDestroy lifecycle callback interceptor methods execute in a transaction context determined by the bean's transaction management type and any applicable transaction attribute.

The PostConstruct callback invocations occur before the first business method invocation on the bean.

An IllegalStateException is thrown for:

- Security related SessionContext methods
  - getCallerPrincipal(), isCallerInRole()

- Asynchronous related method
  - wasCancelCalled()
- Client related SessionContext method
  - getInvokedBusinessInterface()
- All TimerService and Timer methods (except *SessionContext.getTimerService()*)

2.4.4 @Startup

If the Startup annotation appears on the Singleton bean the container must initialize the Singleton bean instance during the application startup sequence.

2.4.5 @DependsOn

The DependsOn annotation is used to express the dependencies on other Singletons.

Example:

enterprise application abc.ear contains b1.jar and b2.jar

packaged in b1.jar

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

packaged in b2.jar

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

@DependsOn({"b1.jar#B", "C"})
@Singleton
public class A { ... }
```

Note: A will start after B and C. The order of B and C is undefined

Circular dependencies within DependsOn metadata are not permitted. Circular dependencies are not required to be detected by the container but will probably result in a deployment error.

2.4.6 @Lock

With container managed concurrency, the container is responsible for controlling concurrent access to the bean instance based on method-level locking metadata. Each business method or timeout method is associated with either a read (shared) lock or write (exclusive) lock.

- By default, there is a Write (exclusive) lock associated with every method (because the write lock is by default declared on class level) of a container managed bean, and the concurrency lock attribute is therefore optional.
- When a Write lock is applied all other concurrent invocations are blocked, i.e. also those ones trying to access a method which has a Read-lock.
- A concurrency locking attribute may be specified on a method to override the one specified on the bean class.

2.4.6.1 Read Lock

@Lock(LockType.READ) - any number of other concurrent invocations on Read methods are allowed to access the bean instance.

2.4.6.2 Write Lock

@Lock(LockType.WRITE) - no other concurrent invocations will be allowed until the Write method's processing completes.

2.4.7 @ConcurrencyManagement

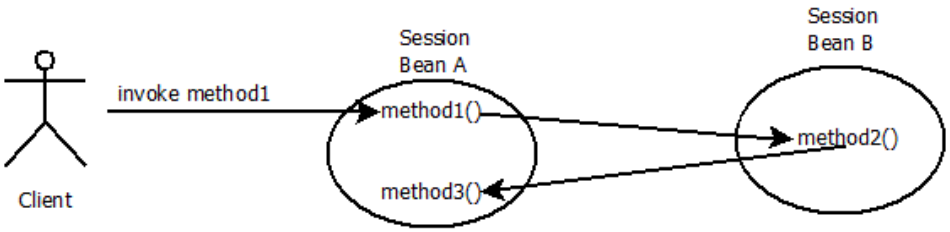
@ConcurrencyManagement (ConcurrencyManagementType.CONTAINER) - container managed

@ConcurrencyManagement (ConcurrencyManagementType.BEAN) - bean managed

2.4.8 @AccessTimeout

- Timeouts can be specified via metadata so that a blocked request can be rejected if a lock is not acquired within a certain amount of time (results in ConcurrentAccessTimeoutException)
- e.g. @AccessTimeout(value = 5, unit = TimeUnit.SECONDS), **default** is **TimeUnit.MILLISECONDS**
- A timeout value of -1 indicates that the client request will block indefinitely
- A timeout value of 0 indicates that concurrent access is not allowed (results in ConcurrentAccessException)

2.4.9 Loopback call



If a loopback call occurs on a Singleton that already holds a Write lock on the same thread:

- If the target of the loopback call is a Read method, the Read lock must always be granted immediately, without releasing the Write lock.
- If the target of the loopback call is a Write method, the call must proceed immediately, without releasing the Write lock.

If a loopback call occurs on a Singleton that holds a Read lock on the same thread (and doesn't also hold a Write lock on the same thread):

- If the target of the loopback call is a Read method, the call must proceed immediately, without releasing the original Read lock.
- If the target of the loopback call is a Write method, an IllegalLoopbackException must be thrown to the caller.

2.4.10 Superclass and locking

Example:

```
public interface SomeBusiness {

   public void firstMethod();
```

```
   public void secondMethod();
   public void thirdMethod();
}
@Lock(LockType.READ)
public class PoJo {
   public void firstMethod () { ... }
   public void secondMethod () { ... }        // Lock (Read) class level
}
@Singleton
public class SingleBean extends PoJo implements SomeBusiness {
   public void firstMethod () { ... }         // Lock (Write) by default
   @Lock(LockType.WRITE)
   public void thirdMethod () { ... }         // Lock (Write) method level
   ...                                  // secondMethod Lock (Read) is inherited
}
```

3 Message Driven Bean (MDB)

3.1 General Aspects

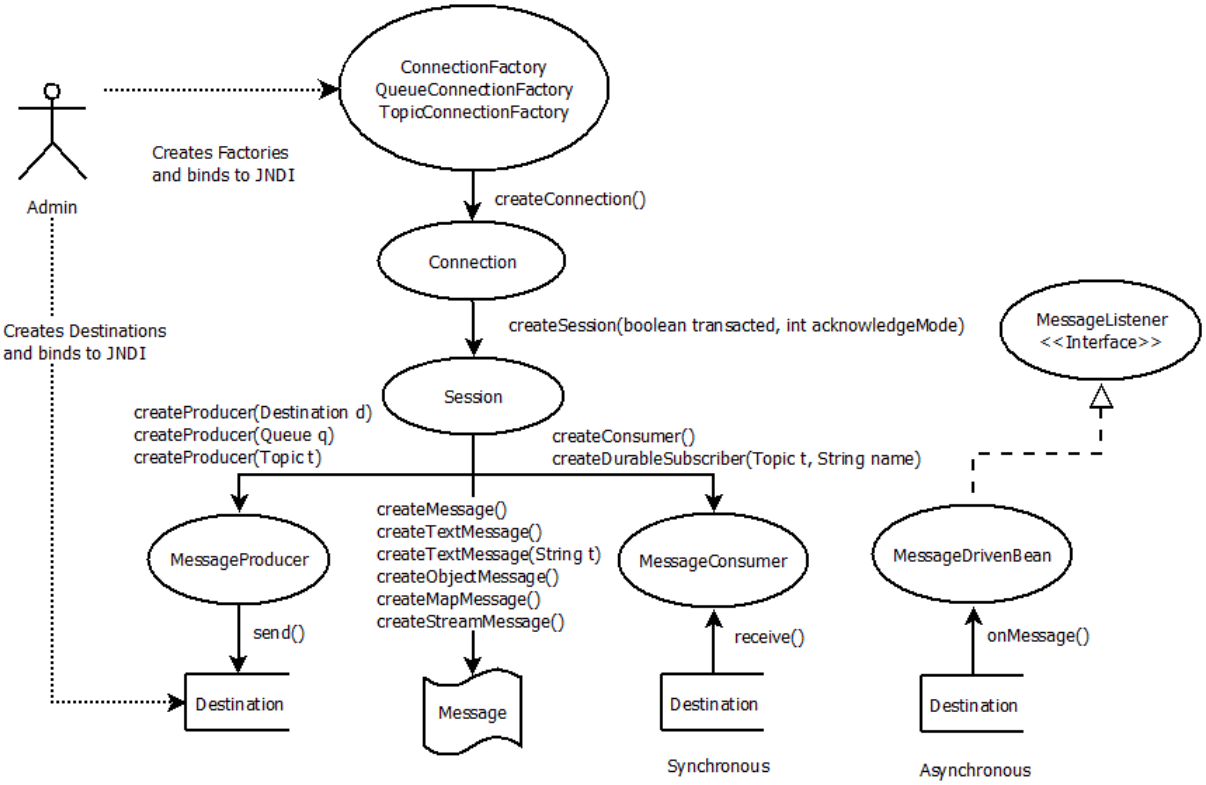
- A Message Driven bean is a message consumer.
- A Message Driven bean is asynchronous.
- A Message Driven bean is invoked by the container as a result of the arrival of a message.
- Message Driven beans have no client
- Message Driven beans have no conversational state
- A Message Driven bean instance variables can contain state across the handling of messages (e.g. database connection or reference to an enterprise bean).
- A Message Driven bean listens for messages on an endpoint or destination (to which a client can send messages)
- A client cannot get a reference to a message driven bean, but only to a message endpoint or destination
- The Message Driven bean has to implement a message listener interface for the messaging type that the Message Driven bean supports (javax.jms.MessageListener interface for a JMS bean)
- A Message Driven bean listener interface may define more than one message listener method. The resource adapter will determine which method will be invoked.
- Optional MessageDrivenBean interface (needed in earlier EJB versions) defines two methods: *setMessageDrivenContext*() and *ejbRemove*().
- More than one Message Driven bean can be associated with the same *JMS queue*, but JMS does not define how messages should be distributed between the queue receivers.
- If a Message Driven bean is *bean managed* and it throws a RuntimeException, the container should not acknowledge the message.

3.2 Java Message Service (JMS)

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages.

Key points:

- Loosely coupled: A component sends a message to a destination, and the recipient can retrieve the message from the destination. However, the sender and the receiver do not have to be available at the same time in order to communicate.
- Reliable: a message is delivered once and only once.
- Asynchronous: A JMS provider can deliver messages to a client as they arrive
- Synchronous: a JMS provider can also send and receive messages synchronously



3.2.1 A JMS application parts

1. A JMS provider is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform includes a JMS provider.
2. JMS clients are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.
3. Administered objects are preconfigured JMS objects created by an administrator for the use of clients. The JMS administered objects are Destination and ConnectionFactory.
4. Messages are the objects that communicate information between JMS clients.

3.2.2 JMS in Java EE

- Application clients, Enterprise JavaBeans (EJBs), and Servlets can send or synchronously receive a JMS message.
- A synchronous *receive()* ties up server resources, so use a timed synchronous receive (e.g. *receive(10)* – wait 10 milliseconds)
- Message Driven Beans, and Application clients can receive messages asynchronously.
- A JMS provider can optionally implement concurrent processing of messages by MDBs.
- Sending and receiving messages can be part of distributed transactions, which allow JMS operations and database accesses to take place within a single transaction.
- An application cannot both send a JMS message and receive a reply to it within the same transaction

3.2.3 Managing JMS Resources (Connections and Sessions) in Session Beans

- For the duration of a business method: close the resource in a finally block within the method.

- For the duration of the Session Bean: use a @PostConstruct callback method to create the resource and to use a @PreDestroy callback method to close the resource. If you use a stateful session bean, close the resource in a @PrePassivate callback method and set its value to null, and you must create it again in a @PostActivate callback method.

3.3 Obtaining access to an message endpoint or message destination

References to message destinations can be injected, or they can be looked up in the client's JNDI namespace.

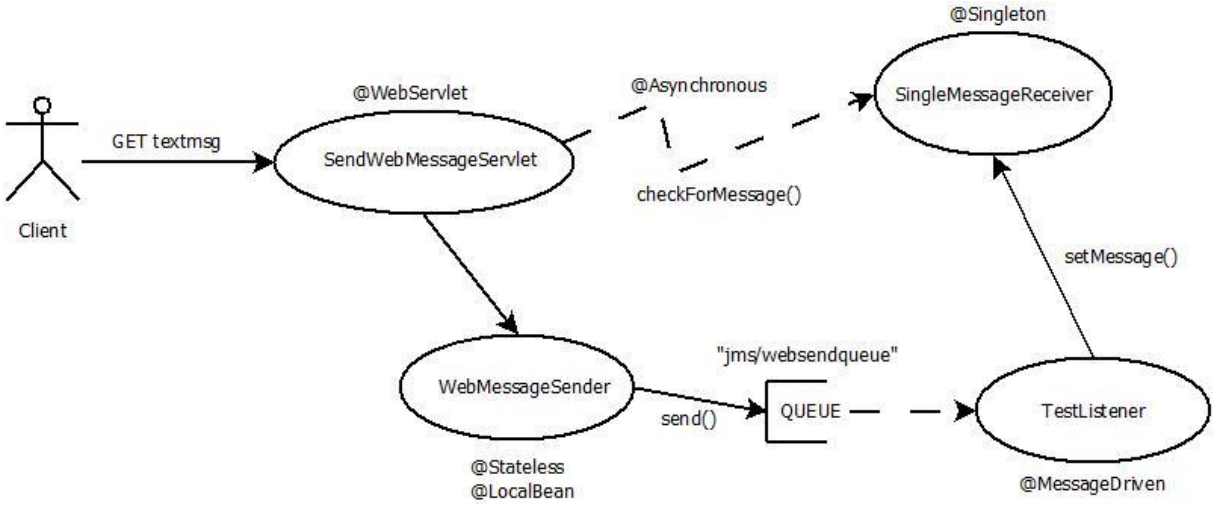
Injected:

```
@Resource (lookup = "jms/websendqueue")
Queue queue;
```

JNDI lookup:

```
Context initialContext = new InitialContext();
Queue queue = (Queue) initialContext.lookup ("java:comp/env/jms/websendqueue");
```

3.4 Example WebMessageSender



In this example we have the following scenario:

We type in a message (*textmsg*) on a browser, and we will send it via the Servlet (*SendWebMessageServlet)* that connects to the Stateless Session Bean (*WebMessageSender)* to the endpoint (QUEUE). Before sending the message we do an asynchronous call from our Servlet (*SendWebMessageServlet*) to see if the Singleton (*SingleMessageReceiver*) already contains a message.

Our Message Driven Bean (*TestListener*) will pick up the message from the queue (*QUEUE*) and will update the Singleton Session Bean (*SingleMessageReceiver)* if it is the first message (it won't do an update for subsequent messages). Before committing the response back to the client we will do an asynchronous call to see if the Singleton meanwhile has received the message (textmsg).

The ear (*sendmessage*) contains the ejb project (*messagebean)* and the web project (*messageweb)*.

```
<application>
  <display-name>sendmessage</display-name>
  <module>
    <ejb>messagebean.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>messageweb.war</web-uri>
      <context-root>lab</context-root>
    </web>
  </module>
</application>
```

in the messagebean project:

```
package nl.notes.ejb.message;
// import statements left out
@Stateless
@LocalBean
public class WebMessageSender {

   @Resource (lookup = "jms/webmessage")
   ConnectionFactory conn;

   @Resource (lookup = "jms/websendqueue")
   Queue queue;

   public void sendWebMessage(String message){
    try {
       Connection connection = conn.createConnection();
       Session session = connection.createSession(true, 0);
       MessageProducer producer = session.createProducer(queue);
       TextMessage msg = session.createTextMessage(message);
       producer.send(msg);
       System.out.println("WebMessageSender EJB Sending message: " + message);
    } catch (JMSException e) {
       // do nothing for the sake of the example
    }
  }
}
```

```
package nl.notes.ejb.message;
// import statements left out
@MessageDriven( activationConfig =
  { @ActivationConfigProperty( propertyName = "destinationType",
                               propertyValue = "javax.jms.Queue"
                              )
  }, mappedName = "jms/websendqueue")
public class TestListener implements MessageListener {

   @EJB
   private SingleMessageReceiver receiver;

   public void onMessage(Message message) {
     try {
       TextMessage text = (TextMessage) message;
       System.out.println("TestListener received this text: "+ text.getText());
       receiver.setMessage(text.getText());
     } catch (JMSException e) {
       // do nothing for the sake of the example
     }
   }
}
```

```
package nl.notes.ejb.message;
// import statements left out
@Singleton
@LocalBean
@Startup
public class SingleMessageReceiver {

   private String singleMessage;

   public String getMessage() {
       return singleMessage;
   }
   // sets the text on the first message arrival
   public void setMessage(String message) {
       System.out.println("SingleMessageReceiver got the text");
```

```
        if (this.singleMessage==null){
                this.singleMessage = message;
        }
    }
    @PostConstruct
    private void init(){
       System.out.println("PostConstruct singleton SingleMessageReceiver,
                           message = " + this.singleMessage);
    }
}
```

in the messageweb project:

```
package nl.notes.servlet.message;
// import statements left out
@WebServlet("/SendWebMessage")
public class SendWebMessageServlet extends HttpServlet {
        @EJB
        private WebMessageSender sender;

        @EJB
        private SingleMessageReceiver receiver;
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
                                                throws ServletException, IOException {
    String par = request.getParameter("textmsg");
    ServletOutputStream out = response.getOutputStream();
    String msg;
    out.print("<html><body>");
    out.print("The singleton (SingleMessageReceiver) contains this message: <br>");
    try {
        msg = checkForMessage().get();
        out.print("<h3>"+msg+"</h3>");
    } catch (Exception e) {
        System.out.println("Something went wrong");
    }
    out.print("SendWebMessageServlet sent this message <br>");
    out.print("to queue called jms/websendqueue: <br>");
    out.print("<h3>"+par+"</h3>");
    System.out.println("SendWebMessageServlet Sending out message " + par);
```

```
   sender.sendWebMessage(par);
   out.print("The singleton (SingleMessageReceiver) contains this message: <br>");
   try {
       msg = checkForMessage().get();
       out.print("<h3>"+msg+"</h3>");
   } catch (Exception e) {
       System.out.println("Something went wrong");
   }
   out.print("</body></html>");
   }

@Asynchronous
private Future<String> checkForMessage(){
   System.out.println("@Asynchronous checkForMessage");
   String result = null;
   result = receiver.getMessage();
   System.out.println("@Asynchronous result = " + result);
   return new AsyncResult<String>(result);
}
}
```

The first time we use the application we might see this order of events:

URL: http://localhost:8080/lab/

Text: first

Output on browser:

The singleton (SingleMessageReceiver) contains this message:

null

SendWebMessageServlet sent this message

to queue called jms/websendqueue:

first

The singleton (SingleMessageReceiver) contains this message:

null

Output on console:

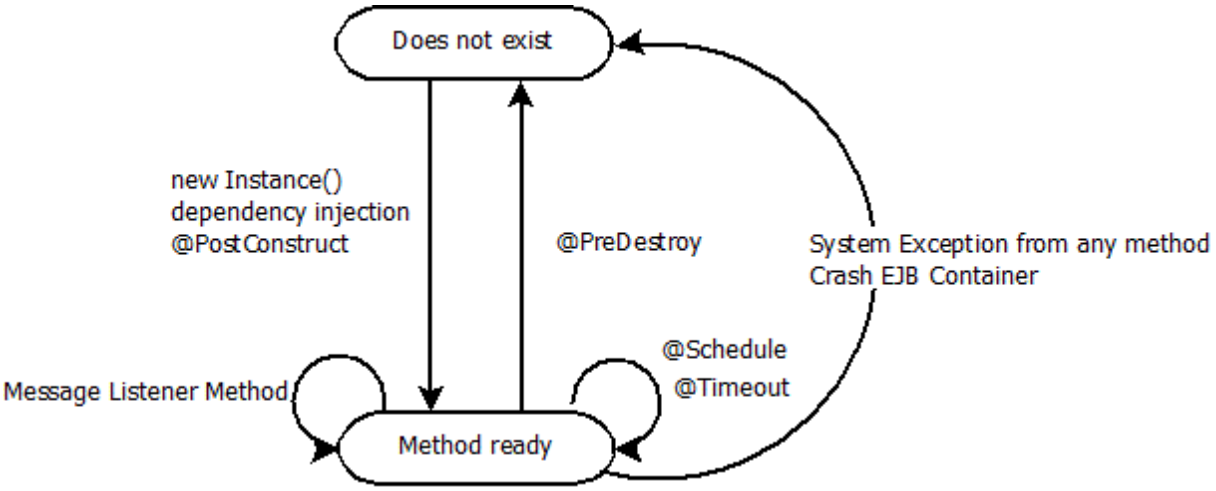
INFO: PostConstruct singleton SingleMessageReceiver, message = null

INFO: @Asynchronous checkForMessage

INFO: @Asynchronous result = null
INFO: SendWebMessageServlet Sending out message first
INFO: WebMessageSender EJB Sending message: first
INFO: @Asynchronous checkForMessage
INFO: @Asynchronous result = null
INFO: TestListener received this text: first
INFO: SingleMessageReceiver got the text

Note that the result of the @Asynchronous call can come after the SingleMessageReceiver has received the text (just type in a few times some text and see if the order changes)

3.5 LifeCycle Diagram



3.6 @PostConstruct, @PreDestroy

The PostConstruct and PreDestroy lifecycle callback interceptor methods execute in an unspecified security context and an unspecified transaction context.

The PostConstruct callback invocations occur before the first message listener method invocation on the bean.

If the bean implements the optional MessageDrivenBean interface (for previous EJB versions) and it uses the @PreDestroy annotation it must be placed on the *ejbRemove()* method

No Access to:

- Another Enterprise bean
- Resource manager
- EntityManager
- TimerService and Timer methods

An IllegalStateException is thrown for:

- Security related MessageDrivenContext methods
  - getCallerPrincipal(), isCallerInRole()
- Transaction related MessageDrivenContext methods
  - getRollbackOnly(), setRollbackOnly()
- All TimerService and Timer methods (except *SessionContext.getTimerService()*)

Note: getInvokedBusinessInterface(),getBusinessObject(), and wasCancelCalled() are not defined in the MessageDrivenContext

3.7 MessageListener interface

The message-driven bean class must implement the message listener interface for the messaging type that the message-driven bean supports or specify the message listener interface using the @MessageDriven parameter (*messageListenerInterface*).

Specifying the message listener interface by annotation metadata:

```
@MessageDriven(
activationConfig =  { @ActivationConfigProperty
               (propertyName = "destinationType", propertyValue = "javax.jms.Queue")
                    },
mappedName = "jms/websendqueue",
messageListenerInterface = javax.jms.MessageListener.Class )
```

3.8 MessageDrivenContext interface

The MessageDrivenContext interface extends the EJBContext interface and doesn't define any extra methods.

3.9 Concurrency

The container serializes calls to each message driven bean instance. Most containers will support multiple instances of a message driven bean to deal with concurrent calls. However each instance sees only a serialized sequence of method calls.

Because the container allows many instances of a message driven bean class to be executing concurrently, no guarantees are made to the order in which messages are delivered to the instances. The application should handle the possibility of messages arriving out of order.

3.10 Transactions

3.10.1 Container-managed

All messages are received and handled within the context of a transaction, when container-managed transactions are chosen.

- *MessageDrivenContext.setRollbackOnly()* causes a message to be redelivered
- The EJB container acknowledges the message receipt
- The REQUIRED transaction attribute, is the default for the *onMessage()* method

- It is not allowed to send and receive a JMS message within a single transaction (because a JMS message is typically not delivered until the transaction commits)
- It is not allowed to use the *Message.acknowledge()* either within a transaction or within an unspecified transaction context.
- The parameters of the following methods are ignored, use (true, 0) or (true, Session.SESSION_TRANSACTED)
    - createSession(boolean transacted, int acknowledgeMode), createQueueSession(boolean transacted, int acknowledgeMode) createTopicSession(boolean transacted, int acknowledgeMode)

3.10.2 Bean-managed

If you use bean-managed transactions, the delivery of a message to the *onMessage()* method takes place outside the distributed transaction context. The transaction begins when you call the *UserTransaction.begin()* method within the *onMessage()* method, and it ends when you call *UserTransaction.commit()* or *UserTransaction.rollback()*.

- Any call to the *Connection.createSession()* method must take place within the transaction
- If you call *UserTransaction.rollback()*, the message is not redelivered
- The EJB container is responsible for acknowledging a message that is successfully processed by the *onMessage()* method
- If the *onMessage()* method throws a RuntimeException, the container does not acknowledge the message.

3.10.3 Transaction Attributes

- REQUIRED or NOT_SUPPORTED for the message listener methods,
- REQUIRED, REQUIRES_NEW, or the NOT_SUPPORTED for timeout callback methods

3.10.4 Message receipt

- Bean managed: message receipt is not part of the transaction
- Container managed: message receipt is part of the transaction when REQUIRED is used

3.10.5 Message redelivery

- Bean managed: a message is redelivered if
    - the *onMessage()* method throws a RuntimeException
- Container managed: a message redelivered if
    - the *onMessage()* method throws a RuntimeException
    - the transaction is rolled back

3.11 Security

- A caller Principal may propagate into a message driven bean's message listener methods however this is not part of the specifications (you cannot get to it in the @PostConstruct method)
- You can use the @RunAs annotation (or corresponding deployment descriptor element) to define a sort of client identity for an EJB that this Message driven bean uses

3.12 @ActivationConfigProperty - activation configuration properties

The configuration of the message driven bean properties like *message acknowledgement modes*, *message selectors*, *expected destination* or *endpoint types* are defined in the activationConfig element of the MessageDriven annotation (activation-config deployment descriptor)

Activation configuration properties specified in the deployment descriptor are added to those specified in the MessageDriven annotation. If a property of the same name is specified in both, the deployment descriptor value overrides the value specified in the annotation.

3.12.1 JMS activation configuration properties

3.12.1.1 Message Acknowledgment – how to acknowledge a message

Bean managed: AUTO_ACKNOWLEDGE (default) or DUPS_OK_ACKNOWLEDGE

```
@ActivationConfigProperty ( propertyName=”acknowledgeMode”,
                            propertyValue=”Auto-acknowledge”)
```

Container managed: don't specify, container handles acknowledgement

3.12.1.2 Message Selectors – how to select certain messages

```
@ActivationConfigProperty ( propertyName=”messageSelector”,
                            propertyValue=”JMSType = ‘Human’ AND eyes = ‘blue’”)
```

3.12.1.3 Destination / endpoint – How to specify the endpoint for the message

```
@ActivationConfigProperty ( propertyName = "destinationType",
                            propertyValue = "javax.jms.Queue")
```

3.12.1.4 Durable / Non-durable (topic) - Is it allowed to miss certain messages

(default is “nondurable”)

```
@ActivationConfigProperty ( propertyName="subscriptionDurability",
                            propertyValue="Durable")
```

3.13 Predestroy not called

The following scenarios result in the PreDestroy lifecycle callback interceptor method(s) not being called for an instance:

1. A crash of the EJB container.
2. A system exception thrown from the instance's method to the container.

The application using the message driven bean should therefore provide some clean up mechanism for the resources that were not cleaned up (or released) by the lack of the PreDestroy calls.

3.14 Superclass Message Driven Bean

A message driven bean class can have a superclass that is a message driven bean.

A listener interface exposed by a particular message driven bean is not inherited by a subclass

```
@MessageDriven
public class A implements Echo { ... }
@MessageDriven
public class B extends A implements Talk
```

Message Driven Bean A exposes local business interface Echo and *Message Driven Bean* B exposes local business interface Talk, but not Echo (unless explicitly put as an implementer)

4 Interceptors for EJBs

- Interceptors specific to enterprise java beans were first introduced to the Java EE platform. You can now use them with Java EE managed objects of all kinds. Contexts and Dependency Injection (CDI) is specified in JSR-299.
- Interceptors may be used with session beans and message-driven beans.
- Default interceptors may be defined to apply to all components within an ejb-jar file or .war file.
- Default interceptors can only be specified in the deployment descriptor.
- The lifecycle of an interceptor instance is the same as that of the bean instance with which it is associated.
- When the target instance is created, a corresponding interceptor instance is created for each associated interceptor class.
- Both the interceptor instance and the target instance are created before any PostConstruct callbacks
- With Stateful Session Beans, the interceptor instances are passivated and activated.
- Interceptor methods may be defined for business methods of session beans and for the message listener methods of message-driven beans.
- Interceptors may be defined for PostConstruct, PreDestroy methods, and PostActivate, PrePassivate (for Stateful Session Beans) callback methods.

4.1 Order of interceptors

General rule:

- The deployment descriptor may be used to override the interceptor invocation order specified in annotations using the <interceptor-order> element

Other rules:

1. Interceptor's superclass. If an interceptor class itself has superclasses, the interceptor methods defined by the superclasses are invoked first
2. Default interceptors are invoked in the order of their specification in the deployment descriptor.
3. Target's class superclass. If a target class has superclasses, any around-invoke methods defined on those superclasses are invoked first.
4. Class interceptors bounded by the @Interceptors annotation in the order of the annotation order.
5. Method-level interceptors bounded by the @Interceptors annotation in the order of the annotation order.
6. The @AroundInvoke method, if any, on the target class itself is invoked.

4.2 Exceptions

- Business method interceptor methods may throw runtime exceptions or application exceptions that are allowed in the throws clause of the business method.
- *AroundInvoke()* methods are allowed to catch and suppress exceptions and recover by calling proceed().

- *AroundInvoke()* methods can mark the transaction for rollback by throwing a runtime exception or by calling the EJBContext.*setRollbackOnly()* method
- *AroundInvoke()* methods may cause this rollback before or after InvocationContext.*proceed()* is called.
- If a system exception is thrown from the interceptor chain the bean instance and any associated interceptor instances are discarded (except for Singleton Session Beans)
- If a system exception is thrown from the interceptor chain, the *PreDestroy()* method is not called and the interceptor methods in the chain should perform any necessary clean-up operations as the interceptor chain unwinds.

4.3 InvocationContext interface

Return	Method	throws Exception
Object	getTarget Returns the target instance, or in other words the object being intercepted.	
Object	getTimer Returns the timer object associated with a timeout method invocation on the target class, or a null value for method and lifecycle callback interceptor methods.	
Method	getMethod Returns the method of the target class for which the interceptor was invoked. For method interceptors, the method of the target class is returned. For lifecycle callback interceptors, a null value is returned.	
Object[]	getParameters Returns the parameter values that will be passed to the method of the target class	IllegalStateException if invoked within a lifecycle callback method
void	setParameters(Object[] params) Sets the parameter values that will be passed to the method of the target class.	IllegalStateException if invoked within a lifecycle callback method IllegalArgumentException if the types do not match or if the number of parameters do not match
Map< String, Object>	getContextData Returns the context data associated with this invocation or lifecycle callback	
Object	proceed Proceed to the next interceptor in the	Exception

	chain. Return the result of the next method invoked, or a null value if the method has return type void.	

4.4 @Interceptor

Defining a class as an interceptor class:

```
@Interceptor
public class ClassAndMethodLogger {
   . . .
}
```

4.5 @AroundInvoke

Method signature:

- Object <METHOD>(InvocationContext) throws Exception

The intercepting method inside the class annotated with @Interceptor

```
@AroundInvoke
 public Object methodLogger(final InvocationContext ctx) throws Exception { . . . }
```

4.6 @AroundTimeout

Method signature:

- Object <METHOD>(InvocationContext) throws Exception

Interceptor methods may be defined for EJB timer timeout methods of session beans and message-driven beans. These are referred to as AroundTimeout methods.

In the *AroundTimeout()* method, the InvocationContext.*getTimer()* method returns the Timer object associated with the timeout being intercepted.

```
@AroundTimeout
protected Object timeoutInterceptorMethod(final InvocationContext ctx) { . . . }
```

Note: you cannot have an Interceptor (with @AroundInvoke) on a Timeout method.

```
@Interceptors({PrincipalLogger.class})  interceptor is not called on a timeout when
@Timeout                                the PrincipalLogger has an @AroundInvoke
public void timerSignal(){
   System.out.println("Timer time-out");
}
```

4.7 @PostConstruct ,@PreDestroy

Method signature:

- In Target class: void <METHOD>()
- In Interceptor class void <METHOD>(InvocationContext)

Interceptor methods can be also bound to a lifecycle callback method:

```
@Interceptors({MyInterceptor.class})
@Stateless
public class EchoBean implements EchoBeanRemote {
   @PostConstruct void initBean() { ... }
}
public class MyInterceptor {
   @PostConstruct
   public void someMethod(InvocationContext ctx) {
      System.out.println("@PostConstruct interceptor");
      ctx.proceed();
   }
}
```

4.8 @Interceptors

Binding the interceptors on the class or method to be intercepted

```
@Interceptors ( {ClassAndMethodLogger.class, PrincipalLogger.class} )
public String echo(String phraseToEcho) { . . . }
```

4.9 Deployment descriptor

Defining the interceptors is done via the <interceptors> element

```
<ejb-jar>
 . . .
<interceptors>
  <interceptor>
   <interceptor-class>nl.notes.interceptor.ClassAndMethodLogger</interceptor-class>
      <around-invoke>
         <method-name>methodLogger</method-name>
      </around-invoke>
      <around-timeout>
         <method-name>timeoutLogger</method-name>
      </around-timeout>
      <post-construct>
         <lifecycle-callback-method>initBean</lifecycle-callback-method>
      </post-construct>
  </interceptor>
</interceptors>
. . .
</ejb-jar>
```

Binding an interceptor class is done via the <interceptor-binding> element:

```
<assembly-descriptor>
```

```
. . .
<interceptor-binding>
  <ejb-name>EchoBean</ejb-name>
  <interceptor-class>nl.notes.interceptors.ClassAndMethodLogger</interceptor-class>
  <interceptor-class>nl.notes.interceptors.PrincipalLogger</interceptor-class>
</interceptor-binding>
. . .
</assembly-descriptor>
```

4.10 Default interceptors

By using the "*" wildcard symbol

```
<assembly-descriptor>
. . .
<interceptor-binding>
  <ejb-name>*</ejb-name>
  <interceptor-class>nl.notes.interceptors.ClassAndMethodLogger</interceptor-class>
</interceptor-binding>
. . .
</assembly-descriptor>
```

4.10.1 @ExcludeDefaultInterceptors

By using @ExcludeDefaultInterceptors, you are disabling the default interceptors

```
@ExcludeDefaultInterceptors
public String echo(String phraseToEcho) {
       return "echo: " + phraseToEcho;
}
```

4.11 Class Interceptors

By defining the interceptors on the class level

```
@Stateless
@Interceptors ({ClassAndMethodLogger.class, PrincipalLogger.class})
public class EchoBean implements Echo {
}
```

4.11.1 @ExcludeClassInterceptors

By using @ExcludeDefaultInterceptors, you are disabling the class interceptors

```
@Stateless
@Interceptors ({ClassAndMethodLogger.class, PrincipalLogger.class})
public class EchoBean implements Echo {
       @Override
       @ExcludeClassInterceptors
```

```
	public String echo(String phraseToEcho) {
		return "echo: " + phraseToEcho;
	}
}
```

4.12 Example Interceptors

In this example the EchoBean's business method *echo()* is intercepted by two interceptors. One interceptor is printing the method and its parameters and the second one is printing the Principal's username. The project setup is the same as in 2.1.5.

In the echobean project:

```
@Remote
public interface Echo {
	public String echo (String phraseToEcho);
}
```

```
package nl.notes.ejb;
// import statements left out
@Stateless
public class EchoBean implements Echo {
	@Override
	@Interceptors( {ClassAndMethodLogger.class, PrincipalLogger.class} )
	public String echo(String phraseToEcho) {
		return "echo: " + phraseToEcho;
	}
}
```

This interceptor method will print the called method with parameters:

example: `INFO: Intercepting nl.notes.ejb.EchoBean.echo( hallo )`

```
package nl.notes.interceptors;
// import statements left out
@Interceptor
public class ClassAndMethodLogger {
  @AroundInvoke
  public Object methodLogger(final InvocationContext context) throws Exception{
	String clazz = context.getMethod().getDeclaringClass().getName();
	String methodName = context.getMethod().getName();
	String listOfParams = getParameters(context);
	System.out.println("Intercepting " + clazz + "." +
                                        methodName + listOfParams);
```

```
        // go to the next Interceptor or, if at the end of the chain,
        // call the intercepted method
        return context.proceed();
    }
    /*
     * Get the parameter values as a string, example: ( param1, param2, param3 )
     */
    private String getParameters(final InvocationContext context) {
        Object[] params = context.getParameters();
        StringBuffer listOfParams = new StringBuffer();
        listOfParams.append("( ");
        if (params!=null && params.length > 0) {
           for (int i = 0; i < params.length - 1; i++) {
              listOfParams = listOfParams.append(params[i].toString() + ", ");
           }
           listOfParams = listOfParams.append(params[params.length - 1]);
        }
        listOfParams.append(" )");
       return listOfParams.toString();
    }
}
```

This interceptor method will print the user invoking the method:

example: `INFO: User calling method: ad`

```
package nl.notes.interceptors;
@Interceptor
// import statements left out
public class PrincipalLogger {
  @AroundInvoke
  public Object userLogger(final InvocationContext context) throws Exception{
        String user = null;
        if (context.getTarget() instanceof EchoBean){
               EchoBean rb = (EchoBean) context.getTarget();
               user = rb.getEjbContext().getCallerPrincipal().getName();
        }
        System.out.println("User calling method: " + user);
        return context.proceed();
  }
}
```

In the echoweb project:

```
package nl.notes.servlet;
// import statements left out
@WebServlet ("/EchoServlet")
public class EchoServlet extends HttpServlet {
  @EJB Echo echoBean;
  @Override
  protected void doGet(HttpServletRequest request,
              HttpServletResponse response) throws ServletException, IOException {
       String phraseToEcho = request.getParameter("phrase");
       ServletOutputStream out = response.getOutputStream();
       out.print("<html><body>");
       out.print(echoBean.echo(phraseToEcho));
       out.print("</body></html>");
  }
}
```

in the web.xml

```
...
<security-constraint>
    <web-resource-collection>
      <web-resource-name>Web Permission</web-resource-name>
      <url-pattern>/*</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>
  <security-role>
    <role-name>admin</role-name>
  </security-role>
  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
...
```

In the sun-application.xml the user (Principal) “ad” is mapped to the role “admin” (after defining the user “ad” in the Server console of Glassfish with a password of “ad”).

```
<sun-application>
       <security-role-mapping>
```

```
            <role-name>admin</role-name>
            <principal-name>ad</principal-name>
      </security-role-mapping>
</sun-application>
```

Executing the example after setting up the Application Server:

```
URL: http://localhost:8080/lab/EchoServlet?phrase=hallo
login with user "ad" and password "ad"
Output on browser:
      echo: hallo
Output on console:
      INFO: Intercepting nl.notes.ejb.EchoBean.echo( hallo )
      INFO: User calling method: ad
```

5 Transactions

- Enterprise JavaBeans support flat transactions. A flat transaction cannot have any child (nested) transactions.
- Enterprise JavaBeans can access resource managers in a transaction only in the bean's methods in which there is a transaction context.
- A session bean or a message-driven bean can be *bean-managed* or *container-managed*. (But it cannot be both at the same time).
- A transaction management type (@TransactionManagement) cannot be specified for Java Persistence (**JPA**) entities.

5.1 Relation with JTA, JTS and JCA

- **Java Transaction API** (**JTA**) is a specification of the interfaces between a transaction manager and the other parties involved in a distributed transaction processing system: the application programs, the resource managers, and the application server.
- **Java Transaction Service** (**JTS**) provides transaction interoperability for transaction propagation between servers.
- **Java Connector Architecture** (**JCA**) is a Java-based technology solution for connecting application servers and enterprise information systems (EIS) including transaction management between the two
- The EJB architecture requires that the EJB container support **JTA** and **JCA**.

5.2 @TransactionManagement

5.2.1 @TransactionManagement(TransactionManagementType.BEAN)

The Bean Provider uses the UserTransaction interface to demarcate transactions.

All updates to the resource managers between the *UserTransaction.begin()* and *UserTransaction.commit()* methods are performed in a transaction

While an instance is in a transaction, the instance should not use the resource-manager specific transaction API (e.g. commit or rollback method on the java.sql.Connection interface or on the javax.jms.Session interface).

A **Stateful Session Bean** instance is not required to commit a started transaction *before* a business method or interceptor method returns.

A **Stateless Session Bean** instance must commit a transaction *before* a business method or interceptor method returns, otherwise:

- An application error will be logged to alert the System Administrator.
- The transaction will be rolled back
- The bean instance will be discarded
- Throw the javax.ejb.EJBException

A **Singleton Session Bean** instance must commit a transaction *before* a business method or interceptor method returns, otherwise:

- An application error will be logged to alert the System Administrator.
- The transaction will be rolled back

- Throw the javax.ejb.EJBException

A **Message Driven Bean** instance must commit a transaction *before* a message listener method or interceptor method returns, otherwise:

- An application error will be logged to alert the System Administrator.
- The transaction will be rolled back
- The bean instance will be discarded

A **Stateless Session Bean** or a **Message Driven Bean** instance must commit a transaction *before* a timeout callback method returns, otherwise:

- An application error will be logged to alert the System Administrator.
- The transaction will be rolled back
- The bean instance will be discarded

A **Singleton Session Bean** instance must commit a transaction *before* a timeout callback method returns, otherwise:

- An application error will be logged to alert the System Administrator.
- The transaction will be rolled back

events when no commit is done in method of BMT	Business	Message Listener	Timeout Callback	Interceptor
Stateful SB	no error	X	X	no error
Stateless SB	ARDE	X	ARD	ARDE
Singleton SB	ARE	X	AR	ARE
Message DB	X	ARD	ARD	ARD

A=Application Error, R=Rollback, D=Discard Bean, E=EJBException

An bean-managed EJB must not use the *getRollbackOnly()* and *setRollbackOnly()* methods of the EJBContext interface.

5.2.1.1 UserTransaction interface

The UserTransaction interface defines the methods that allow a Bean-Managed EJB to explicitly manage transaction boundaries.

Return	Method	throws Exception
void	begin create a new transaction and associate it with the current thread	NotSupportedException thread is already associated with a transaction or bean is Container managed SystemException unexpected error condition
void	commit complete the transaction associated with the current thread.	RollbackException the transaction has been rolled back rather than committed. HeuristicMixedException

		HeuristicRollbackException SecurityException thread is not allowed to commit the transaction. IllegalStateException current thread is not associated with a transaction. SystemException
void	rollback roll back the transaction associated with the current thread.	java.lang.IllegalStateException java.lang.SecurityException SystemException
void	setRollbackOnly modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.	IllegalStateException SystemException
int	getStatus obtain the status of the transaction associated with the current thread.	SystemException
void	setTransactionTimeout(int seconds) modify the timeout value that is associated with transactions started by the current thread with the begin method.	SystemException

5.2.1.2 UserTransaction.getStatus()

A bean-managed EJB should use *int getStatus()* method of the UserTransaction interface to get the status of the current transaction:

- Status.STATUS_ACTIVE, (there is a started transaction associated with current thread)
- Status.STATUS_NO_TRANSACTION,
- Status.STATUS_MARKED_ROLLBACK,
- Status.STATUS_COMMITTED
- Status.STATUS_ROLLEDBACK
- ….

5.2.2 @TransactionManagement(TransactionManagementType.CONTAINER)

The container managed beans

- business methods,
- message listener methods,
- business method interceptor methods,
- lifecycle callback interceptor methods,
- timeout callback methods

- timeout callback interceptor methods
  must not use any resource-manager specific transaction management methods.
  must not attempt to obtain or use the UserTransaction interface.

5.2.2.1 EJBContext.setRollbackOnly()

A container-managed EJB can use the *setRollbackOnly()* method of its EJBContext object to mark the transaction such that the transaction can never commit.

> An EJB marks a transaction for rollback to protect data integrity before throwing an application exception (if the application exception has not been specified to automatically rollback the transaction)

The container must throw the IllegalStateException if the method is invoked from a business method executing without a transaction context. This could be when the method has the transaction attribute SUPPORTS, but it will be when the method has the transaction attribute values of NOT_SUPPORTED or NEVER.

5.2.2.2 EJBContext.getRollbackOnly()

An EJB can use the *getRollbackOnly()* method of its EJBContext object to test if the current transaction has been marked for rollback.

> The transaction might have been marked for rollback by the enterprise bean itself, by other enterprise beans, or by other components (outside of the EJB specification scope)
>
> Note: if the transaction has rolled back this method will return false. After rolling back there is no active transaction context anymore attached to this EJBContext (i.e. it has the same state as when no transaction has been started)

The container must throw the IllegalStateException if the method is invoked from a business method executing without a transaction context. This could be when the method has the transaction attribute SUPPORTS, but it will be when the method has the transaction attribute values of NOT_SUPPORTED or NEVER.

5.3 Isolation

- The isolation level describes the degree to which the access to a resource manager by a transaction is isolated from the access to the resource manager by other concurrently executing transactions.
- The API for managing an isolation level is resource-manager-specific.
- Managing isolation levels are not part of the EJB architecture.
- If an enterprise bean uses multiple resource managers, the Bean Provider may specify the same or different isolation level for each resource manager.
- Changing the isolation level half way a transaction may cause a commit in the middle.

5.4 @TransactionAttribute

The transaction attribute for a method of a bean with container-managed transaction demarcation is REQUIRED.

A transaction attribute is a value associated with each of the following methods:

- a method of a bean's business interface

- a method exposed through the bean class no-interface view
- a message listener method of a message-driven bean
- a timeout callback method
- a session bean's web service endpoint method
- a singleton session bean's PostConstruct/PreDestroy lifecycle callback interceptor methods

Transaction attributes can be defined in the deployment descriptor as an alternative to annotations (to supplement or override annotations).

The application assembler and the deployer are allowed to override the transaction attribute values in the deployment descriptor.

Transaction Attributes	
Annotations	Deployment descriptor
MANDATORY	Mandatory
REQUIRED	Required
REQUIRES_NEW	RequiresNew
SUPPORTS	Supports
NOT_SUPPORTED	NotSupported
NEVER	Never

The following table described the transaction attribute values that are allowed for specific methods.

Type of method	MANDATORY	REQUIRED	REQUIRES_NEW	SUPPORTS	NOT_SUPPORTED	NEVER
Business method Session Bean	MANDATORY	REQUIRED	REQUIRES_NEW	SUPPORTS	NOT_SUPPORTED	NEVER
MessageListener method in Message Driven Bean	X	REQUIRED	X	X	NOT_SUPPORTED	X
Lifecycle Callback	X	X	X	X	X	X
Lifecycle Callback Singleton	X	REQUIRED	REQUIRES_NEW	X	NOT_SUPPORTED	X
Timeout Callback	X	REQUIRED	REQUIRES_NEW	X	NOT_SUPPORTED	X
Asynchronous method	X	REQUIRED	REQUIRES_NEW	X	NOT_SUPPORTED	X
Session Synchronization	MANDATORY	REQUIRED	REQUIRES_NEW	X	X	X
Passivation Activation methods	X	X	X	X	X	X

If a transaction attribute is MANDATORY for a business method and the caller doesn't have a transaction context the EJBTransactionRequiredException will be thrown.

If a transaction attribute is NEVER for a business method and the caller does have a transaction context the EJBException will be thrown.

5.4.1 Superclass and transaction attribute

Example:

```
public interface SomeBusiness {
   public void firstMethod();
   public void secondMethod();
   public void thirdMethod();
}
@TransactionAttribute (SUPPORTS)
public class PoJo {
   public void firstMethod () { ... }
   public void secondMethod () { ... }        // SUPPORTS class level
}
@Stateless
public class StatBean extends PoJo implements SomeBusiness {
   public void firstMethod () { ... }         // REQUIRED by default
   @TransactionAttribute (REQUIRES_NEW)
   public void thirdMethod () { ... }         // REQUIRES_NEW method level
   ...                                        // secondMethod SUPPORTS is inherited
}
```

5.4.2 Deployment descriptor <trans-attribute>

A <container-transaction> contains a <method> and a <trans-attribute> element. The <method> comes in three variations:

Style 1

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>*</method-name>
</method>
```

There can be at most one of this style

Style 2

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

A Style 2 element takes precedence over methods also defined in Style 1

Style 3

```
<method>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
```

```
  <method-params>
    <method-param>PARAMETER_1</method-param>
    ...
    <method-param>PARAMETER_N</method-param>
  </method-params>
</method>
```

A Style 3 element takes precedence over methods also defined in Style 1 or 2

Interface Style

```
<method>
  <method-intf>Remote</method-intf>
  <ejb-name>EJBNAME</ejb-name>
  <method-name>METHOD</method-name>
</method>
```

Same method in different interfaces, use the <method-intf> element:

5.4.3 Transaction Attributes Table Bean-managed

A client transaction is always suspended when it calls a Bean-managed EJB (even if the client is another Bean Managed EJB).

Client's transaction	Transaction currently associated with instance	Transaction associated with the method
none	none	none
T1	none	none
none	T2	T2
T1	T2	T2

The transaction currently associated with the instance is only possible for a Stateful SB

5.4.4 Transaction Attributes Table Container-managed

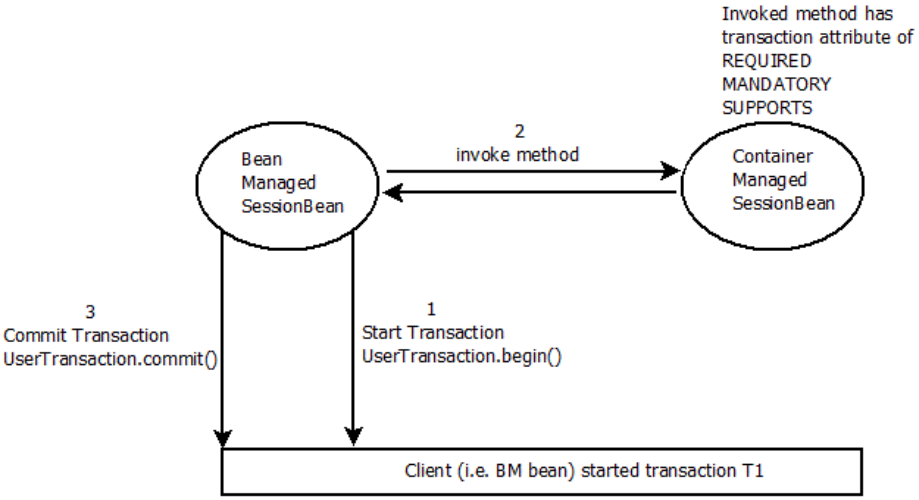
If the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction on the basis of the setting of the *transaction attribute*.

Transaction attribute	Client's transaction	Transaction associated with business method	Transaction associated with resource managers
NOT_SUPPORTED	none	none	none
	T1	none	none
REQUIRED	none	T2	T2
	T1	T1	T1
SUPPORTS	none	none	none
	T1	T1	T1
REQUIRES_NEW	none	T2	T2
	T1	T2	T2
MANDATORY	none	EJBTransactionRequiredException	N/A

	T1	T1	T1
NEVER	none	none	none
	T1	EJBException	N/A

T1 is a transaction passed with the client request, while T2 is a transaction initiated by the container

Example:



Note that the BM-Session Bean is considered client of the CM-Session Bean

The BM Session bean starts a transaction (1) by invoking the method UserTransaction.*begin()*. The transaction T1 is started and will be the client's transaction context. The BM Session bean invokes a method (2) on a CM Session bean. According to the table the Container Managed Session bean will now run under the transaction context (T1) of the client (i.e. Bean-Managed Session bean). Finally the BM Session bean commits the transaction (3) by invoking UserTransaction.*commit()*.

6 Exceptions

6.1 Application Exception

- An application exception is an exception defined by the Bean Provider as part of the business logic of an application.
- A client can typically recover from an application exception.
- An application exception may be a "checked exception" (extends Exception)

Example:

```
package nl.notes.ejb.exception;

public class NotEnoughCreditException extends Exception {

}
```

- An application exception may not be a subclass of the RemoteException (although it extends Exception, it is considered a system exception)
- An application Exception that is a checked exception is defined in the throws clause of a method
- Application exceptions are intended to be handled by the client, so they should not be used for reporting system level problems.
- An application exception thrown by an enterprise bean should be reported to the client precisely
- An application exception thrown by an enterprise bean should not automatically rollback a client's transaction.

6.1.1 @ApplicationException

- An application exception may be an "unchecked exception" (extends RuntimeException).
- An unchecked exception is an ApplicationException when annotated with @ApplicationException (or in the deployment descriptor in the application-exception element)

Example:

```
package nl.notes.ejb.exception;

@ApplicationException

public class SystemNotAvailableException extends RuntimeException {

}
```

- To rollback: @ApplicationException (rollback=true)
- If the application exception is not specified to cause transaction rollback, mark the transaction for rollback using the *EJBContext.setRollbackOnly()* method before throwing the application exception

6.1.2 Subclass Application Exception

When an unchecked exception is an application exception, it also applies to subclasses of that exception. The inheriting behavior can be disabled by setting the @ApplicationException (inherited = false)

example:

```
@ApplicationException (inherited=false, rollback=false)
public class ExceptionOne
public class ExceptionTwo extends ExceptionOne
```

ExceptionTwo is not an ApplicationException

6.2 System Exception

A system exception is an exception that is a RemoteException (or one of its subclasses) or a RuntimeException that is not an ApplicationException.

A system exception is either:

- An unexpected exception
- An expected exception, but not recoverable

Examples:

- failure to obtain a database connection,
- JNDI exceptions,
- unexpected RemoteException from invocation of other enterprise beans,
- unexpected RuntimeException,
- JVM errors,
- etc.

If the enterprise bean method encounters a system-level exception or error that does not allow the method to successfully complete, the method should throw a suitable non-application exception that is compatible with the method's throws clause.

Guidelines on how to handle System Exceptions:

- The EJB bean method does not have to catch the exception, it should simply propagate the error from the bean method to the container.
- If the EJB method performs an operation that results in a checked exception that the bean method cannot recover, the bean method should throw the EJBException that wraps the original exception.
- Any other unexpected error conditions should be reported using the EJBException.

Note that the EJBException is a subclass of the RuntimeException, and therefore it does not have to be listed in the throws clauses of the business methods.

If the container catches a non-application exception, it throws an EJBException, or if the web service client view is used, a RemoteException.

6.2.1 EJBException and subclasses

EJBException	Is thrown to report that the invoked business method or callback method could not be completed because of an unexpected error (e.g. the instance failed to open a database connection).
ConcurrentAccessException	Indicates that the client has attempted an invocation on a stateful session bean or singleton bean while another invocation is in progress and such concurrent access is not allowed.

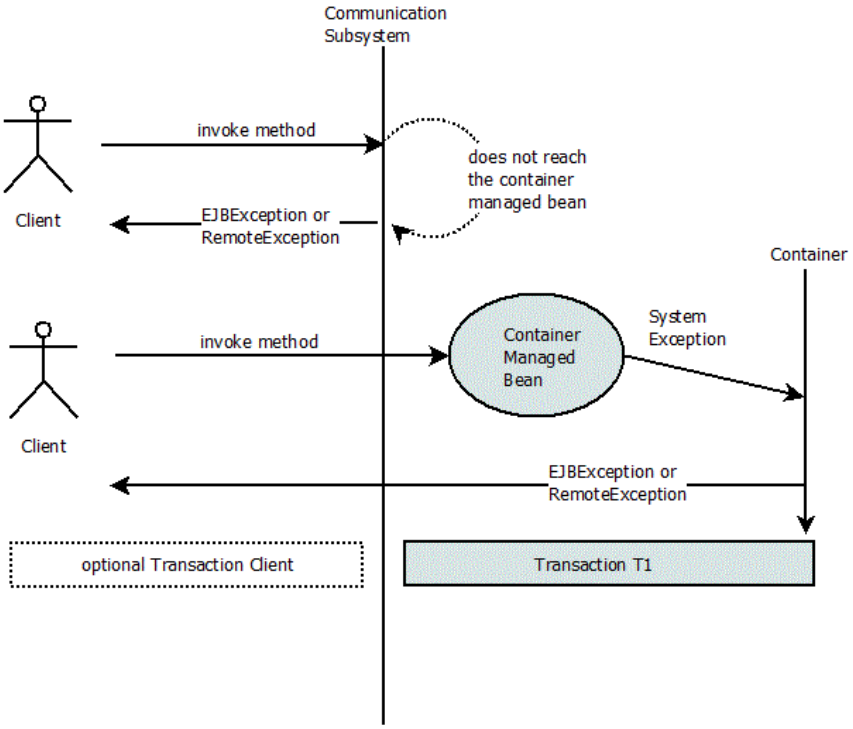
ConcurrentAccessTimeoutException	Indicates that an attempt to concurrently access a stateful session or singleton bean method resulted in a timeout.
IllegalLoopbackException	Indicates that an attempt was made to perform an illegal loopback invocation.
EJBAccessException	Indicates that client access to a business method was denied.
EJBTransactionRolledbackException	Is thrown to a remote client to indicate that the transaction associated with processing of the request was marked to roll back.
NoMoreTimeoutsException	Indicates that a calendar-based timer will not result in any more timeouts.
NoSuchEJBException	Is thrown if an attempt is made to invoke a business method on a stateful session or singleton object that no longer exists.
NoSuchEntityException	Is thrown by an entity bean instance to its container to report that the invoked business method or callback method could not be completed because of the underlying entity was removed from the database. (JPA)
NoSuchObjectLocalException	Is thrown if an attempt is made to invoke a method on a local object (e.g. timer) that no longer exists.
EJBTransactionRequiredException	This exception is thrown to a remote client to indicate that a request carried a null transaction context, but the target object requires an active transaction.

6.3 Client's View of Exceptions

- The methods of the **business interface** typically do not throw the java.rmi.RemoteException, regardless of whether the interface is a remote or local interface. The throws clauses may include an arbitrary number of ApplicationExceptions.
- The JAX-RPC **web service endpoint interface** are Java Remote Method Invocation (RMI) interfaces, and therefore the throws clauses of all their methods (including those inherited from superinterfaces) include the mandatory java.rmi.RemoteException. The throws clauses may include an arbitrary number of ApplicationExceptions.
- The **no-interface view** is a local view, and the throws clauses of all its methods must not include the java.rmi.RemoteException. The throws clauses may include an arbitrary number of ApplicationExceptions.

6.3.1 java.rmi.RemoteException and javax.ejb.EJBException

- If a client receives the javax.ejb.EJBException or the java.rmi.RemoteException as an indication of a failure to invoke an enterprise bean method or to properly complete its invocation. The exception can be thrown by the container or by the communication subsystem between the client and the container.
- If the client receives the javax.ejb.EJBException or the java.rmi.RemoteException from a method invocation, the client, in general, does not know if the enterprise bean's method has been completed or not.

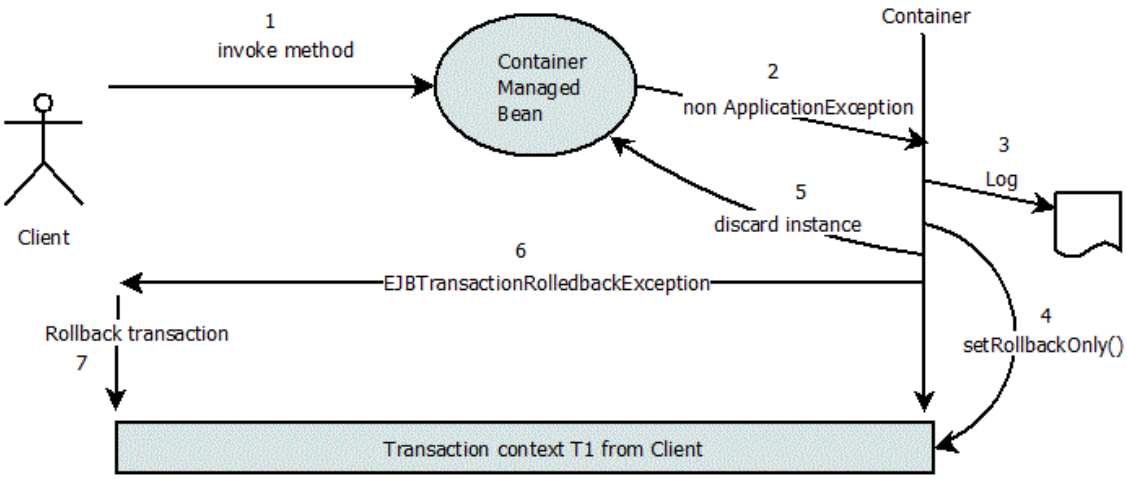


6.4 Handling of exceptions

The container will handle system exceptions differently based on the type of transaction, the type of EJB, type of method, and who started the transaction.

Example:

We have a Client of a Container Managed Bean (e.g. another CM bean, BM bean, or application client). This client is running under an existing transaction context (T1). When the client invokes a method (1) on the CM bean, and that method throws a non ApplicationException (2), the container will log (3) the exception, mark the transaction for Rollback (4), discard the EJB (5, not for Singleton) and throw an EJBTransactionRolledBackException (6) to the Client. The Client will rollback (7) the transaction because of the mark set by the CM bean.



The next table describes the events in case of a non ApplicationException (i.e. System Exception) being thrown from a particular method.

Type of method	ApplicationException allowed	System Exception events
Business method Session Bean	Yes	• Log Exception • Discard EJB instance (not Singleton) • Rollback (or mark for rollback: setRollBackOnly() if transaction was started by client) • throw EJBException or EJBTransactionRollbackException (if transaction was started by client, not MDB)
MessageListener method Message Driven Bean	Yes (must be a RuntimeException annotated with @ApplicationException)	
Business Interceptor @AroundInvoke	Yes	
Lifecycle Callback @PostConstruct, @PreDestroy	No	• Log Exception • Discard EJB instance • Rollback (only Singleton has transaction context)
Lifecycle Callback Interceptor @PostConstruct, @PreDestroy	No	
Timeout Callback @Timeout, @Schedule	No	• Log Exception • Discard EJB instance (not Singleton) • Rollback
Timeout Callback Interceptor @AroundTimeout	No (yes according to the spec: the interceptor is allowed to throw any Exception that the intercepted method is throwing, so practically no)	
Other Callback methods @PrePassivate, @PostActivate, @AfterBegin, @BeforeCompletion, @AfterCompletion	No	• Log Exception • Discard EJB instance • Rollback (or mark for rollback if transaction was started by client) • throw EJBException or EJBTransactionRollbackException (if transaction was started by client)
Asynchronous method Session Bean (return value void) @Asynchronous	No	[1][2] • When dispatched, any System Exception from the asynchronous method will not be received by the client
Asynchronous method Session Bean (return value Future<V>) @Asynchronous	Yes	[1][2] • When dispatched, any System Exception thrown from the asynchronous method will be received by the client. *Future.get()* will get an ExecutionException and the original

		System Exception will be available via *ExecutionException.getCause()*

[1] This explanation of a System Exception is only about the client's behavior, when the target instance executing the asynchronous method throws one. What happens to the target instance (might be discarded) is not explained.
[2] If calling the asynchronous method results in an EJBException immediately the container cannot handle the asynchronous method (lack of resources).

7 Injection and Environment Naming Context (ENC)

- Most EJBs must access resource managers and external information. The JNDI naming context and Java language metadata annotations make it possible for enterprise beans that they can locate external information without prior knowledge of how the external information is named and organized in the target operational environment.
- For enterprise beans packaged in an **ejb-jar**, each enterprise bean defines its own set of environment entries in the namespace java:comp/env
- In a **.war**, there is only a single naming environment shared between all web-components and enterprise beans in the module.
- Enterprise bean instances are not allowed to modify the bean's environment at runtime
- In general, lookups of objects in the JNDI "java:" namespace are required to return a new instance of the requested object every time. Exceptions are allowed for the following:
    - The container knows the object is immutable (for example, objects of type java.lang.String), or knows that the application can't change the state of the object.
    - The object is defined to be a singleton, such that only one instance of the object may exist in the JVM.
    - The name used for the lookup is defined to return an instance of the object that might be shared. The name "java:comp/ORB" is such a name.
- Each injection of an object corresponds to a JNDI lookup
- A field or method of a bean class may be annotated to request an entry from the environment to be injected (e.g. resource manager or simple environment entry)
- Injection may also be requested using entries in the deployment descriptor corresponding to each of these (resource) types.
- The field or method may have any access qualifier (public, private, etc.) but must not be static.
- If a field of the bean class is the target of injection, that field must not be final.
- A JNDI lookup from the SessionContext is relative to "java:comp/env"
- An advantage from a JNDI lookup from the SessionContext to the InitialContext is that only runtime exceptions are thrown from the lookup() (no checked exception handling).

7.1 @EJB – injecting an EJB

The JNDI name of a (injected) EJB is as follows:

java:comp/env/<package_name>.<bean_name>/<field_name>

Note that the following injection examples have a JNDI lookup just to show the JNDI name. The only time a lookup is needed is when the @EJB or @Resource is declared on the bean class.

example:

```
package nl.notes.ejb;
@Stateless
@LocalBean
public class InjectedBean {
	@EJB EchoBeanRemote echo;
	@Resource SessionContext context;
```

```
      public String getInjectedBean(String phrase) {
            EchoBeanRemote g = (EchoBeanRemote) context.lookup
                              ("java:comp/env/nl.notes.ejb.InjectedBean/echo");
            return g.echo(phrase);
      }
}
```

Or when using the “name” attribute of the EJB annotation:

```
package nl.notes.ejb;
@Stateless
@LocalBean
public class InjectedBean {
   @EJB (name=”ejb/echo”)
   EchoBeanRemote echo;
   @Resource SessionContext context;

   public String getInjectedBean(String phrase) {
      EchoBeanRemote g = (EchoBeanRemote) context.lookup("ejb/echo");
                        // or ("java:comp/env/ejb/echo");
                        // when using the InitialContext
      return g.echo(phrase);
   }
}
```

Or when using injection on a setter method:

```
package nl.notes.ejb;
@Stateless
@LocalBean
public class InjectedBean {
   EchoBeanRemote echo;
   @Resource SessionContext context;

   @EJB (name="ejb/echo")
   public void setEcho(EchoBeanRemote echo) {
      this.echo = echo;
   }

   public String getInjectedBean(String phrase) {
      EchoBeanRemote g = (EchoBeanRemote) context.lookup("ejb/echo");
                       // or ("nl.notes.ejb.InjectedBean/echo")
```

```
                        // when "name" attribute is not used
                        // or ("java:comp/env/nl.notes.ejb.InjectedBean/echo")
                        // when InitialContext.lookup() is used
      return g.echo(phrase);
   }
}
```

Annotations (like @EJB, @EJBs, @Resource, @Resources) may also be applied to the bean class itself. These annotations declare an entry in the bean's environment, but do not cause the resource to be injected. Instead the bean is expected to use the EJBContext lookup method or the methods of the JNDI API to lookup the entry.

(*name* and *beanInterface* are mandatory)

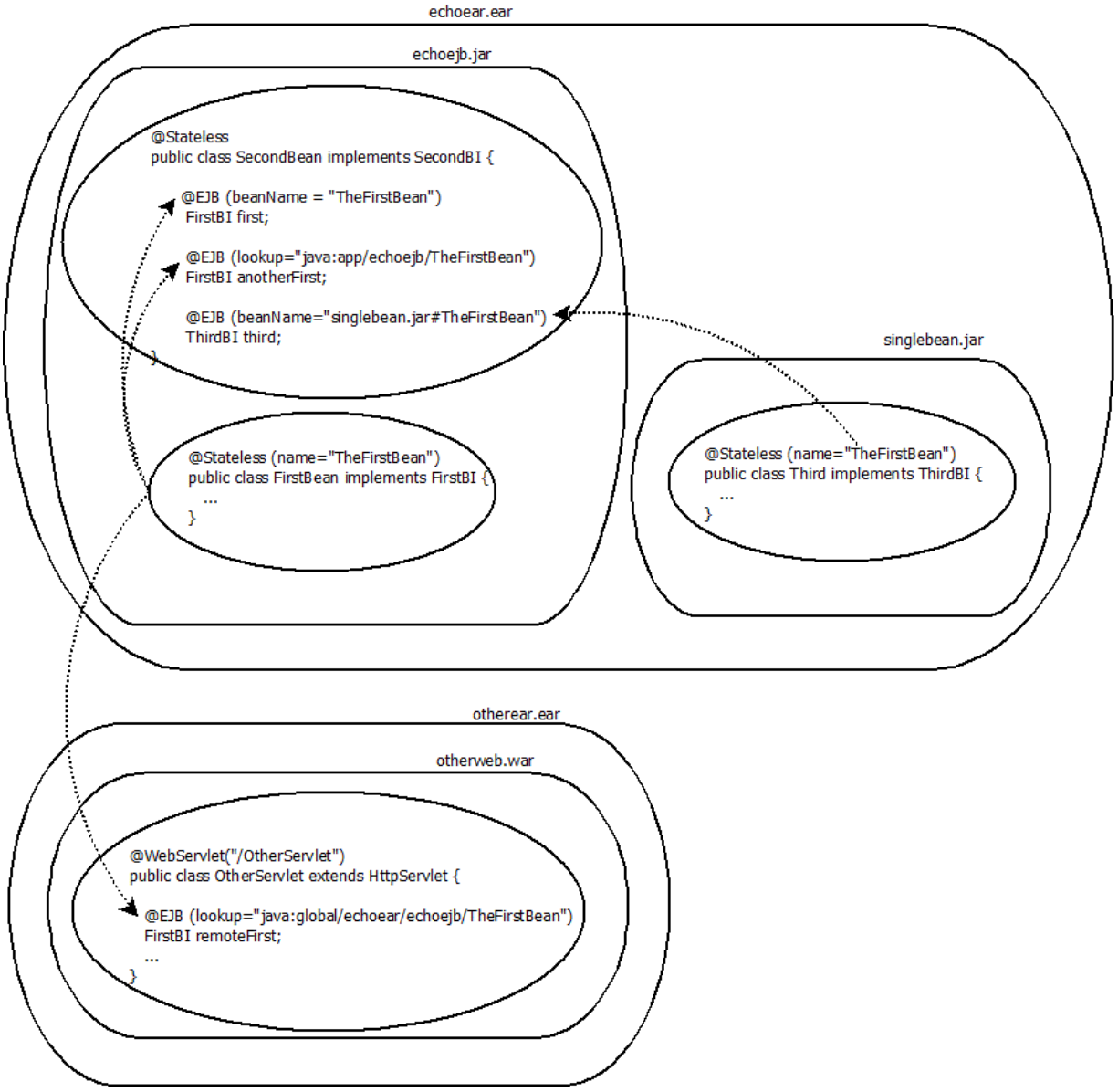
```
package nl.notes.ejb;
@Stateless
@LocalBean
@EJB(name="echo", beanInterface=EchoBeanRemote.class)
public class InjectedBean {
       @Resource SessionContext context;

       public String getInjectedBean(String phrase) {
              EchoBeanRemote g = (EchoBeanRemote) context.lookup("echo");
              return g.echo(phrase);
       }
}
```

7.2 @EJB - beanName vs lookup

In the @EJB there are two attributes to find the EJB you want to inject. The beanName can be used within the same application or stand-alone module, whereas lookup can be used from another application. The beanName refers to the EJB that was given that name in the "name" attribute (e.g. *@Stateless (name="TheFirstBean")* and the referring *@EJB (beanName="TheFirstBean")*).

If you have two beans with the same name inside an ear, you can differentiate by naming the jar of the bean (e.g. *@EJB (beanName="singlebean.jar#TheFirstBean"*).



7.3 @EJBs - declaring multiple EJBs

When declaring multiple EJBs on a class, you will have to use the @EJBs annotation

```
@EJBs ( {
          @EJB (name="single", beanInterface=SingleBean.class),
          @EJB (name="packagedBean", beanInterface=PackagedBean.class) } )
public class EchoServlet extends HttpServlet {
   @Override
   protected void doGet(HttpServletRequest request,
       HttpServletResponse response) throws ServletException, IOException {


   InitialContext context;
   try {
    context = new InitialContext();
    PackagedBean pb = (PackagedBean) context.lookup("java:comp/env/packagedBean");
```

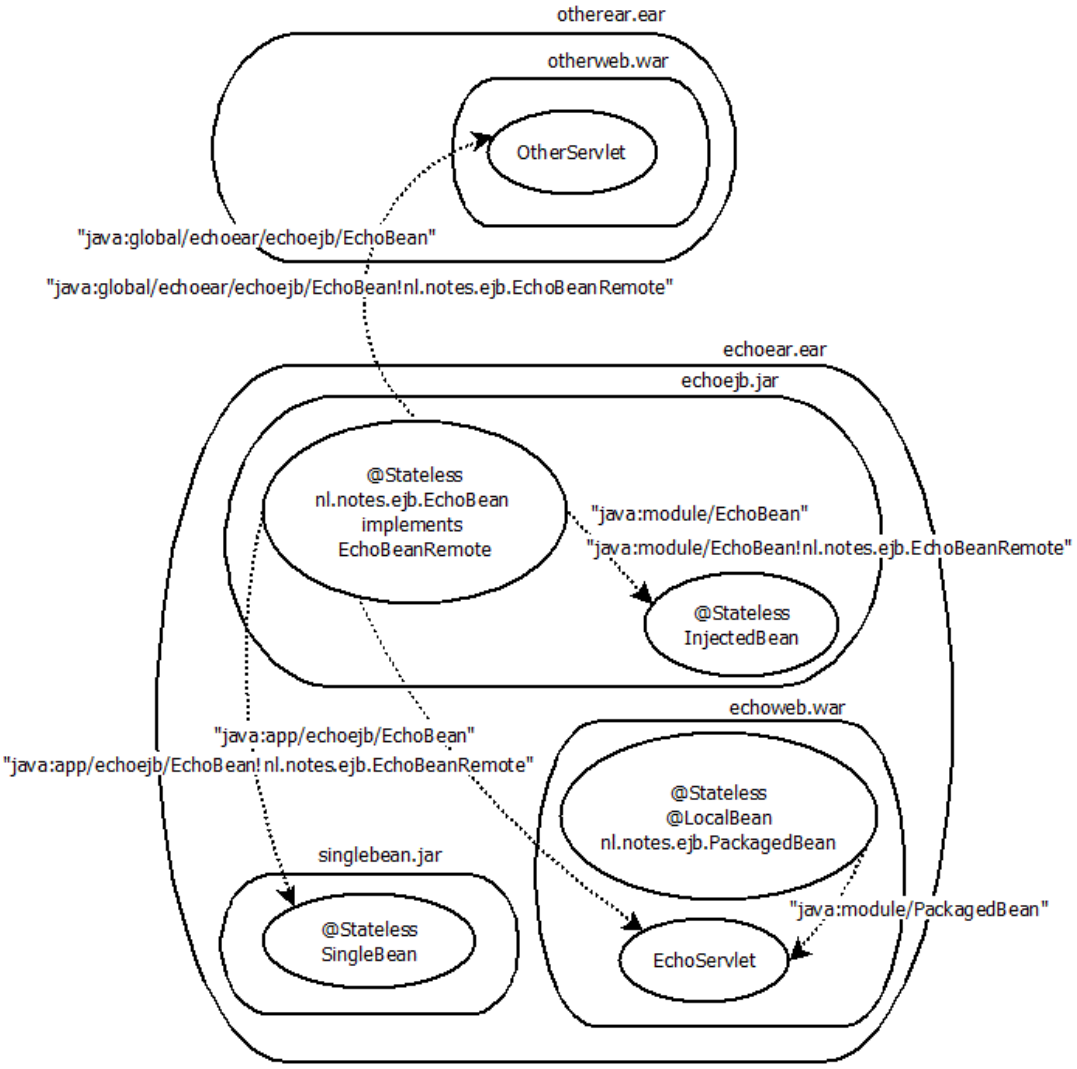
```
    pb.talk("JNDI via class EJBs");
   } catch (NamingException e) {
        e.printStackTrace();            }
   }
}
```

7.4 Portable JNDI names (java:global, java:app, java:module)

The Java EE Platform Specification defines a standardized global JNDI namespace and a series of related namespaces that map to the various scopes of a Java EE application. These namespaces can be used by applications to portably retrieve references to components (e.g. EJBs) and resources.

```
java:global[/<app-name>]/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
java:app/<module-name>/<bean-name>[!<fully-qualified-interface-name>]
java:module/<bean-name>[!<fully-qualified-interface-name>]
```

example:



Note: that <bean-name> is the unqualified name of the Session Bean (PackagedBean for nl.notes.ejb.PackagedBean)

7.5 @Resource

Enterprise beans can declare dependencies on external resources and other objects in their environment with the @Resource annotation. Examples of these resources are:

Database resource:

- DataSource

JMS resource:

- Queue
- Topic
- ConnectionFactory (QueueConnectionFactory, TopicConnectionFactory)

Common Object Request Broker Architecture resource:

- CORBA ORB reference

EJB Container resources:

- UserTransaction
- TimerService
- EJBContext

JPA resources:

- PersistenceUnit
- PersistenceContext

Simple Environment Entries

- String, Character, Integer, Boolean, Double, Byte, Short, Long, and Float.

7.5.1 shareable and authenticationType

shareable - by default, connections to a resource manager are shareable across other enterprise beans in the application that use the same resource in the same transaction context. Do not to share a resource: @Resource(shareable=false) or <res-sharing-scope> Unshareable </res-sharing-scope>

authenticationType - there are two choices with respect to dealing with associating a principal with the resource manager access:

- Allow the Deployer to set up principal mapping or resource manager sign-on information. default, @Resource(authenticationType=AuthenticationType.CONTAINER) or <res-auth>CONTAINER</res-auth>
- Sign on to the resource manager from the bean code @Resource(authenticationType=AuthenticationType.APPLICATION) or <res-auth> APPLICATION </res-auth>

7.5.2 name vs. lookup

You can make a resource in the global JNDI (lookup="jms/websendqueue") available under another name (name="q") in the bean class.

```
package nl.notes.ejb;
@Stateless
```

```
@LocalBean
@Resource (name="q", lookup = "jms/websendqueue", type=Queue.class)
public class WebMessageSender {
   @Resource SessionContext sc;
   public Queue getQueue(){
       Queue queue = (Queue) sc.lookup ("q");     // or "java:comp/env/q"
       return queue;
   }
}
```

7.6 @Resources – declaring multiple resources

With the @Resources annotation you can declare multiple resources for a bean class.

```
package nl.notes.ejb;
@Stateless
@LocalBean
@Resources( { @Resource(name="q", lookup = "jms/websendqueue", type=Queue.class),
              @Resource(name="sc", type=SessionContext.class) } )
public class WebMessageSender {
   ...
}
```

7.7 Important references in the global JNDI

- java:comp/UserTransaction
- java:comp/TimerService
- java:comp/EJBContext

7.8 Environment entries

An environment entry <env-entry> is a configuration parameter used to customize an enterprise bean's business logic.

- The environment entry is declared in the ejb-jar.xml.
- The environment entry values may be one of the following Java types: String, Character, Integer, Boolean, Double, Byte, Short, Long, and Float.
- The *authenticationType* and *shareable* elements of the @Resource annotation must not be specified.

example:

```
<ejb-jar>
...
<enterprise-beans>
      <session>
            <ejb-name>EchoBean</ejb-name>
            <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
```

```
            <session-type>Stateless</session-type>
            <env-entry>
                  <env-entry-name>initNumber</env-entry-name>
                  <env-entry-type>java.lang.Integer</env-entry-type>
                  <env-entry-value>10</env-entry-value>
            </env-entry>
      </session>
</enterprise-beans>
...
</ejb-jar>
```

7.8.1 Injecting the environment entry by annotation

```
@Stateless
public class EchoBean implements EchoBeanRemote {
   @Resource(name="initNumber")
   Integer init;
   . . .
}
```

7.8.2 Injecting the environment entry by deployment descriptor

```
@Stateless
public class EchoBean implements EchoBeanRemote {
   Integer init;
   . . .
}
and corresponding ejb-jar.xml
<ejb-jar>
<enterprise-beans>
<session>
...
<env-entry>
      <env-entry-name>initNumber</env-entry-name>
      <env-entry-type>java.lang.Integer</env-entry-type>
      <env-entry-value>10</env-entry-value>
      <injection-target>
       <injection-target-class>nl.notes.ejb.EchoBean</injection-target-class>
       <injection-target-name>init</injection-target-name>
      </injection-target>
```

```
</env-entry>
...
</session>
</enterprise-beans>
</ejb-jar>
```

7.8.3 @Resource in combination with ejb-jar.xml

In this section we zoom in on the containers behavior if we have both the @Resource annotation as well as the ejb-jar.xml deployment descriptor with simple environment entries.

7.8.3.1 @Resource without parameters

If this is our EJB and we want to inject an Integer (simple environment entry) into it:

```
package nl.notes.ejb;
@Stateless
public class EchoBean implements EchoBeanRemote {
   @Resource SessionContext context;
   @Resource Integer init;
   public String echo(String phraseToEcho) {
      Integer lookupInit = (Integer) context.lookup("nl.notes.ejb.EchoBean/init");
      return "echo: " + lookupInit + " : " + anotherInit;
   }
}
```

When the "name" and "type" are not specified in the @Resource annotation, the container will infer the name and type from the resource, so it will look for an entry with (the default) JNDI-name: "nl.notes.ejb.EchoBean/init" and type "java.lang.Integer". If it finds that entry in the ejb-jar.xml it will inject the value from there into the EJB (value 10 in this case).

```
<ejb-jar>
<enterprise-beans>
   <session>
   <ejb-name>EchoBean</ejb-name>
   <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
   <session-type>Stateless</session-type>
      <env-entry>
         <env-entry-name>nl.notes.ejb.EchoBean/init</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>10</env-entry-value>
      </env-entry>
    </session>
</enterprise-beans>
</ejb-jar>
```

7.8.3.2 @Resource with name parameter

If you define the @Resource with a “name” parameter it will work a bit differently:

```
package nl.notes.ejb;
@Stateless
public class EchoBean implements EchoBeanRemote {
   @Resource SessionContext context;
   @Resource (name="initNumber") Integer init;
   public String echo(String phraseToEcho) {
      Integer lookupInit = (Integer) context.lookup("initNumber");
      return "echo: " + lookupInit + " : " + init;
   }
}
```

The container will now do a lookup without the default name, because there is an explicit name specified. It will look for an entry with the JNDI-name "initNumber". This means that if there is an entry with the JNDI-name "initNumber" and type "java.lang.Integer" in the deployment descriptor for this SessionBean (EchoBean), it will inject the value (11 in this case).

```
<ejb-jar>
<enterprise-beans>
   <session>
   <ejb-name>EchoBean</ejb-name>
   <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
   <session-type>Stateless</session-type>
      <env-entry>
         <env-entry-name>initNumber</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>11</env-entry-value>
      </env-entry>
    </session>
</enterprise-beans>
</ejb-jar>
```

7.8.3.3 Injection from ejb-jar.xml

If the EJB is now defined like this:

```
package nl.notes.ejb;
@Stateless
public class EchoBean implements EchoBeanRemote {
   @Resource SessionContext context;
   @Resource Integer init;
   public String echo(String phraseToEcho) {
   // this will throw a javax.naming.NameNotFoundException:
```

```
   // No object bound to name java:comp/env/init
   // Integer lookupInit = (Integer) context.lookup("nl.notes.ejb.EchoBean/init");
      return "echo: " + init;
   }
}
```

The "name" and "type" are not specified in the @Resource annotation, so the container will infer the name and type from the resource. It will look for an entry with (the default) JNDI-name: "nl.notes.ejb.EchoBean/init" and type "java.lang.Integer", but it won't find the entry in the ejb-jar.xml so no injection is done (at first).

However if there is another environment entry that specifies an injection target on the instance variable "init", the injection is done from the deployment descriptor into the field "init".

For instance the environment-entry “OtherInit” specifies the field “init” of the EchoBean as injection target. In this example the value 12 will be injected in the field “init”.

```
<ejb-jar>
<enterprise-beans>
   <session>
   <ejb-name>EchoBean</ejb-name>
   <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
   <session-type>Stateless</session-type>
      <env-entry>
         <env-entry-name>OtherInit</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>12</env-entry-value>
         <injection-target>
            <injection-target-class>nl.notes.ejb.EchoBean</injection-target-class>
            <injection-target-name>init</injection-target-name>
         </injection-target>
      </env-entry>
   </session>
</enterprise-beans>
</ejb-jar>
```

7.8.3.4 @Resource with name and lookup parameters

If we define the @Resource annotation with a “name” and “lookup” parameter:

```
@Stateless
public class EchoBean implements EchoBeanRemote {
   @Resource (name="i", lookup="java:comp/env/init3")
   Integer thirdInit;
   @Resource
   SessionContext context;
```

```
    @Override
    public String echo(String phraseToEcho) {
        // do a lookup from the ENC via SessionContext is relative to java:comp/env
        // you will notice that entry java:comp/env/i
        // contains the same as java:comp/env/init3
        Integer lookupInit3 = (Integer) context.lookup("i");
        return "echo: " + phraseToEcho + thirdInit + lookupInit3;
    }
}
```

and we have defined “init3” in the deployment descriptor:

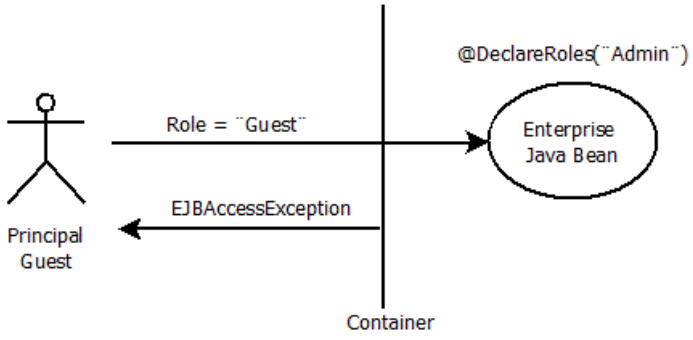
```
<ejb-jar>
<enterprise-beans>
   <session>
   <ejb-name>EchoBean</ejb-name>
   <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
   <session-type>Stateless</session-type>
      <env-entry>
         <env-entry-name>init3</env-entry-name>
         <env-entry-type>java.lang.Integer</env-entry-type>
         <env-entry-value>13</env-entry-value>
      </env-entry>
    </session>
</enterprise-beans>
</ejb-jar>
```

The container will inject the value 13 into the field “thirdInit” and it will create another entry in the beans environment with name “java:comp/env/i” because of the “name” parameter in the @Resource annotation.

Note: an entry is only created when both “name” and “lookup” are specified in the @Resource annotation.

8 Security

- A security role is a semantic grouping of permissions that a given type of users of the application must have in order to successfully use the application.
- The set of security roles used by the application is taken from the @DeclareRoles and @RolesAllowed annotations and the security roles define in the <security-role> element of the deployment descriptor element.
- If the container denies a client access to a business method, the container should throw the javax.ejb.EJBAccessException



8.1 EJBContext

The EJBContext provides two methods that allow the Bean Provider to access security information about the enterprise bean's caller.

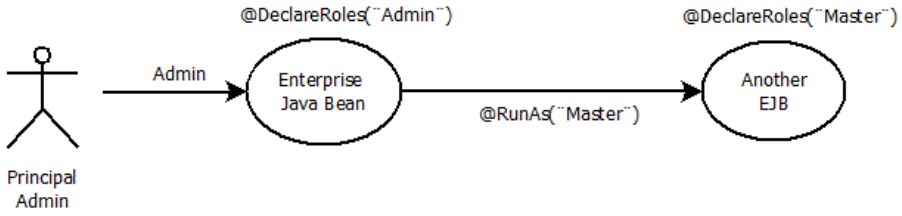
```
Principal getCallerPrincipal();                -> use the Principal.getName()

boolean isCallerInRole(String roleName);       -> to verify if caller is allowed
```

- If they are otherwise invoked when no security context exists, they should throw the IllegalStateException runtime exception.
- The container must never return a null from the *getCallerPrincipal()* method.

8.2 @RunAs

The security principal under which a method invocation is performed is typically that of the component's caller. By specifying a run-as identity, however, it is possible to specify that a different principal be substituted for the execution of the methods.



Define a @RunAs identity by annotation

```
@RunAs ("admin")

@Stateless public class EmployeeServiceBean implements EmployeeService{

...

}
```

or by using the deployment descriptor:

```
<ejb-jar>
...
   <enterprise-beans>
   <session>
      <ejb-name>EmployeeService</ejb-name>
      ...
      <security-identity>
         <run-as>
            <role-name>admin</role-name>
         </run-as>
      </security-identity>
      ...
   </session>
   </enterprise-beans>
...
</ejb-jar>
```

8.3 @DeclareRoles

The @DeclareRoles annotation is used to declare the security role names used in the enterprise bean code. The deployment descriptor equivalent is the <security-role> element:

```
<assembly-descriptor>
…
   <security-role>
      <description>
       This role should be assigned Administrators of the application
      </description>
      <role-name>admin</role-name>
   </security-role>
…
</assembly-descriptor>
```

The @DeclareRoles annotation is specified on a bean class, where it serves to declare roles that may be tested by calling *isCallerInRole()* from within the methods of the annotated class

8.3.1 Declarative security

```
@DeclareRoles ("payroll", "manager")
@Stateless public class PayrollBean implements Payroll {
   @Resource SessionContext context;
   // The salary can only be requested by callers with the
```

```
   // security role "manager" and "payroll". So if there is a security role
   // "employee", that caller wouldn't be allowed to execute this method
   @RolesAllowed("payroll", "manager")
    public Integer getSalary(String name, Integer employeeNr){
       Integer sal = ... read from database;
       return sal;
    }
    ...
}
```

8.3.2 Programmatic security

```
@DeclareRoles ("payroll", "manager")
@Stateless public class PayrollBean implements Payroll {
   @Resource SessionContext context;
   // The salary field can be changed only by callers who have the security
   // role "payroll". Note that callers with the role "manager" are allowed
   // to invoke this method, but wouldn't succeed in updating
   public void updateSalary(EmplInfo info) {
      oldInfo = ... read from database;
      if (info.salary != oldInfo.salary && !context.isCallerInRole("payroll"))
      { throw new SecurityException(...); }
      // update salary
    }
}
```

or via the deployment descriptor:

```
<enterprise-beans>
...
<session>
 <ejb-name>AardvarkPayroll</ejb-name>
 <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
 ...
 <security-role-ref>
    <description>
       This security role should be assigned to the employees of the payroll
       department who are allowed to update employees' salaries.
    </description>
    <role-name>payroll</role-name>
```

```
  </security-role-ref>
...
</session>
...
</enterprise-beans>
```

8.4 Method Permissions with Annotations

Specifying the @RolesAllowed or @PermitAll or @DenyAll annotation on the bean class means that it applies to all business methods of the class. Method permissions may be specified on a method of the bean class to override the method permissions value specified on the bean class.

8.4.1 @RolesAllowed

The value of the @RolesAllowed is a list of security roles that are permitted to execute the specified method(s).

8.4.2 @PermitAll

The @PermitAll specifies that all security roles (and unauthenticated users) are permitted to execute the specified method(s).

8.4.3 @DenyAll

The @DenyAll specifies that no security roles are permitted to execute the specified method(s).

8.4.4 Superclass and security permissions

```
public interface SomeBusiness {
   public void firstMethod();
   public void secondMethod();
   public void thirdMethod();
}
@RolesAllowed ("admin")
public class SomeClass {
   public void firstMethod () {...}  // @RolesAllowed("admin") by class
   public void secondMethod () {...}
   ...
}
@Stateless public class MyBean extends SomeClass implements SomeBusiness {
   @RolesAllowed ("HR")
   public void secondMethod () {...}  // @RolesAllowed("HR") by method override
   public void thirdMethod () {...}   // not defined
   ...                          // firstMethod @RolesAllowed("admin") is inherited
}
```

8.5 Method Permissions by deployment descriptor

You can define the method permissions relation in the deployment descriptor using the method-permission elements as follows:

- Each <method-permission> element includes a list of one or more security roles and a list of one or more methods. All the listed security roles are allowed to invoke all the listed methods. Each security role in the list is identified by the <role-name> element, and each method is identified by the <method> element.
- The method permissions relation is defined as the union of all the method permissions defined in the individual method-permission elements.
- A security role or a method may appear in multiple method-permission elements.
- Method permissions in the deployment descriptor will override the method permissions specified by annotations.

8.5.1 Permit methods - <unchecked/>

You can indicate that all roles are permitted to execute one or more specified methods by using the <unchecked> element (i.e., the methods should not be "checked" for authorization prior to invocation by the container):

```
<assembly-descriptor>
   . . .
   <method-permission>
      <method>
         <unchecked/>
         <ejb-name>EjbName</ejb-name>
         <method-name>EjbMethod</method-name>
      </method>
      . . .
   </method-permission>
   . . .
</assembly-descriptor>
```

If the method permission relation specifies both the unchecked element for a given method and one or more security roles, all roles are permitted for the specified methods.

8.5.2 Deny methods - <exclude-list>

The <exclude-list> element can be used to indicate the set of methods that should not be called.

```
<assembly-descriptor>
   . . .
   <method-permission>
      <description>Allow role employee to call method</description>
      <role-name>employee</role-name>
      <method>
         <ejb-name>EjbName</ejb-name>
```

```
         <method-name>EjbMethod</method-name>
      </method>
   </method-permission>
   . . .
   <exclude-list>
      <description>Do not allow any role</description>
         <method>
            <ejb-name>EjbName</ejb-name>
            <method-name>EjbNoAllowed</method-name>
         </method>
   </exclude-list>
</assembly-descriptor>
```

If a given method is specified both in the <exclude-list> element and in the method permission relation: no access is permitted to the method.

8.6 Unspecified Method Permissions

It is possible that some methods are not assigned to any security roles nor annotated as @DenyAll or contained in the <exclude-list> element. Those methods must be treated by the container as <unchecked/>.

8.7 Linking Security Role References to Security Roles

The security role references used in the components of the application are linked to the security roles defined for the application.

You can link each <security-role-reference> to a <security-role> using the <role-link> element. The value of the <role-link> element must be the name of one of the security roles defined in a <security-role> element or by means of the @DeclareRoles or @RolesAllowed annotations.

The <role-link> need not be specified when the <role-name> used in the code is the same as the name of the <security-role> (to be linked).

Example

The code using the role “payroll”

```
@Stateless
public class MyBean extends SomeClass implements SomeBusiness {
   @RolesAllowed (“payroll”)
   public void someMethod () {...}
   ...
}
```

the descriptor linking “payroll” to “payroll-department”

```
<assembly-descriptor>
  . . .
 <enterprise-beans>
```

```
  <session>
   <ejb-name>AardvarkPayroll</ejb-name>
   <ejb-class>com.aardvark.payroll.PayrollBean</ejb-class>
   <security-role-ref>
     <description> Employees of the payroll department.</description>
     <role-name>payroll</role-name>
     <role-link>payroll-department</role-link>
   </security-role-ref>
  </session>
  </enterprise-beans>
  . . .
  <security-role>
     <role-name>payroll-department</role-name>
  </security-role>
  . . .
</assembly-descriptor>
```

9 Timer Service

- The EJB Timer Service is a container-managed service that allows callbacks to be scheduled for time-based events
- An enterprise bean accesses this service by means of dependency injection, through the EJBContext interface, or through lookup in the JNDI namespace.
- The EJB Timer Service is a coarse-grained timer notification service that is designed for use in the modeling of application-level processes. It is not intended for the modeling of real-time events.
- Timers cannot be created for Stateful Session beans.
- The @Schedule annotation can be used to automatically create a timer with a particular timeout schedule.
- Optional TimedObject interface, defining method *ejbTimeout()*
- If the TimedObject interface is implemented and the @Timeout annotation are both used, the @Timeout can only be on the *ejbTimeout()* method
- Timer instances must not be serializable, whereas a TimerHandle is serializable.
- A timer may be cancelled by a bean before its expiration by calling the *cancel()* method on the timer
- Timers survive container crashes, server shutdown, and the activation/passivation (container handles serialization) and load/store cycles of the enterprise beans that are registered with them (unless disabled on a per-timer basis)

A timer is created to schedule timed callbacks. The bean class of an enterprise bean that uses the timer service must provide one or more timeout callback methods

9.1.1 Programmatic Timers vs. Automatic Timers

Automatically created timers are created by the container as a result of application deployment.

Programmatically created timers need the enterprise bean method to be called, otherwise they are never created.

For automatically created timers, the timeout method is the method that is annotated with the @Schedule annotation

For programmatically created timers, the timeout method may be a method that is annotated with the @Timeout annotation, or if the bean implements the javax.ejb.TimedObject interface the timeout-method is *ejbTimeout().*

For programmatically created timers there is one timeout callback method per class.

For automatically created timers there can be more than one time callback method per class.

Automatically created timers can specify an information String.

Programmatically created timers can specify a Serializable object as extra information.

9.2 @Schedule

There are seven attributes in a calendar-based time expression:

	Attribute	Possible values	Notes
1	second	[0,59]	
2	minute	[0,59]	

3	hour	[0,23]	
4	dayOfMonth	[1,31] [-7, -1] “Last” {“1st”, “5th”, “Last”} {“Sun”, “Sat”}	-x (where x is in the range [-7, -1]) means x day(s) before the last day of the month
5	month	[1,12] {“Jan”, “Feb”, “Mar”, ‘’Apr”, “May”, “Jun”, “Jul”, “Aug”, “Sep”, “Oct”, “Nov”, Dec”}	
6	dayOfWeek	[0,7] {“Sun”, “Mon”, “Tue”, “Wed”, “Thu”, “Fri”, “Sat”}	“0” and “7” both refer to Sunday
7	year	a four-digit calendar year	

9.2.1 Wild Card

“*” represents all possible values for a given attribute.

Example: “every Monday”

- by defining: every year, every month, every dayOfMonth, dayOfWeek=”Mon”, when the hour is 0, the minute is 0, and the second is 0

```
@Schedule (second=”0”, minute=”0”, hour=”0”, dayOfMonth=”*”, month=”*”,
dayOfWeek=”Mon”, year=”*”)
```

9.2.2 Range

Constrains the attribute to an inclusive range of values, with a dash separating both ends of the range.

Members of a range cannot themselves be lists, wild-cards, ranges, or increments.

The range “x-x”, where both range values are the same, evaluates to the single value x

The dayOfWeek range “0-7” is equivalent to “*”

In range ”x-y”, if x is larger than y, the range is equivalent to “x-max, min-y”

```
@Schedule(dayOfMonth = “27-3”) Equivalent to @Schedule(dayOfMonth=“27-Last,1-3”) or
@Schedule(dayOfMonth = “27-Last”) and @Schedule(dayOfMonth=“1-3”)
```

9.2.3 List

Separate the elements by comma

```
@Schedule(dayOfWeek = “Mon,Wed,Fri”, minute = “0-10,30,40”, second = “10,20,30”)
```

9.2.4 Increments

The forward slash constrains an attribute based on a starting point and an interval, and is used to specify

“Every N { seconds | minutes | hours } within the { minute | hour | day }”.

```
@Schedule(minute = “*/5”)        Every five minutes within the hour
@Schedule(second = “30/10”)      Every 10 secs within the min, starting at second 30
```

```
@Schedule(minute = "*/14", hour="1,2")     Every 14 minutes within the hour,
                                           for the hours of 1 and 2 a.m.)
@Schedule(hour="12/2", dayOfMonth="2nd Tue")  Every other hour within the day
                                              starting at noon on the 2nd
                                              Tuesday of every month.
```

For expression x/y, the attribute is constrained to every *yth* value within the set of allowable values beginning at time x. The x value is inclusive. The wildcard character (*) can be used in the x position, and is equivalent to 0.

9.2.5 Time Zone Support

Schedule-based timer times are evaluated in the context of the default time zone associated with the container in which the application is executing.

A schedule-based timer may optionally override this default and associate itself with a specific time zone.

```
@Schedule(minute="15", hour="3", timezone="America/New_York")
```

9.2.6 Expression Rules

- The second, minute, and hour attributes have a default value of "0".
- The dayOfMonth, month, dayOfWeek, and year attributes have a default value of "*".
- If dayOfMonth has a non-wildcard value and dayOfWeek has a non-wildcard value, then either the dayOfMonth field or the dayOfWeek field must match the current day (even though the other of the two fields need not match the current day).
- Whitespace is ignored, except for string constants and numeric values.
- All string constants ("Sun", "Jan", "1st" etc.) are case insensitive.
- Duplicate values within List attributes are ignored.
- Increments are only supported within the second, minute, and hour attributes.

9.2.7 Info string

A @Schedule annotation can optionally specify an information string. This string is retrieved by calling Timer.*getInfo()* on the associated Timer object. If no information string is specified, the method returns null.

9.2.8 Non-persistent Timers

A non-persistent timer is a timer whose lifetime is tied to the JVM in which it is created. A non-persistent timer is considered cancelled in the event of application shutdown, container crash, or a failure/shutdown of the JVM on which the timer was started.

```
@Schedule(minute="*/10", hour="*", persistent=false)
public void getLatestNews() {


}
```

Or when using programmatic timers:

```
@Resource TimerService timerService;
. . .
```

```
private void createTimer() {
       long duration = 1000; // miliseconds
       TimerConfig tf = new TimerConfig();
       tf.setPersistent(false);
       timerService.createSingleActionTimer(duration, tf);
}
```

9.3 @Schedules

Multiple automatic timers can be applied to a single timeout callback method using the Schedules annotation.

```
@Schedules(
              {@Schedule(dayOfMonth="21", month="1-11", info="MonthlyInvitation"),
               @Schedule(dayOfMonth="15", month="Last", info="LastMonthInvitation")
} )
public void sendEndOfMonthInvitation() { ... }
```

9.4 Timer Service Interface

The TimerService interface provides EJBs access to the container-provided Timer Service.

The timer duration is expressed in terms of milliseconds.

By default, all timers created using the timer creation methods are persistent. A non-persistent timer can be created by calling setPersistent(false) on a TimerConfig object

Return	Method	throws Exception
Timer	createTimer(long duration, Serializable info) create a single-action timer that expires after a specified duration (duration in miliseconds)	IllegalArgumentException If duration is negative IllegalStateException if this method is invoked while the instance is in a state that does not allow access to this method EJBException if this method could not complete due to a system-level failure
Timer	createTimer(Date expiration, Serializable info) create a single-action timer that expires at a given point in time	IllegalArgumentException if expiration is null or expiration.getTime() is negative IllegalStateException EJBException

Timer	createSingleActionTimer(long duration, TimerConfig timerConfig) create a single-action timer that expires after a specified duration	IllegalArgumentException IllegalStateException EJBException
Timer	createSingleActionTimer(Date expiration, TimerConfig timerConfig) create a single-action timer that expires at a given point in time	IllegalArgumentException IllegalStateException EJBException
Timer	createTimer(long initialDuration, long intervalDuration, Serializable info) create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval	IllegalArgumentException if initialDuration is negative or intervalDuration is negative IllegalStateException EJBException
Timer	createTimer(Date initialExpiration, long intervalDuration, Serializable info) create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval	IllegalArgumentException IllegalStateException EJBException
Timer	createIntervalTimer(long initialDuration, long intervalDuration, TimerConfig timerConfig) create an interval timer whose first expiration occurs after a specified duration, and whose subsequent expirations occur after a specified interval	IllegalArgumentException IllegalStateException EJBException
Timer	createIntervalTimer(Date initialExpiration, long intervalDuration, TimerConfig timerConfig) create an interval timer whose first expiration occurs at a given point in time and whose subsequent expirations occur after a specified interval	IllegalArgumentException IllegalStateException EJBException
Timer	createCalendarTimer(ScheduleExpression schedule) create a calendar-based timer based on the input schedule expression	IllegalArgumentException if Schedule represents an invalid schedule expression IllegalStateException EJBException
Timer	createCalendarTimer(ScheduleExpression schedule, TimerConfig timerConfig)	IllegalArgumentException IllegalStateException

	create a calendar-based timer based on the input schedule expression	EJBException
Collection <Timer>	getTimers() get all the active timers associated with this bean	IllegalStateException EJBException

9.5 TimerConfig

The TimerConfig is used to specify additional timer configuration settings during timer creation.

Return	Method	throws Exception
Serializable	getInfo() The info object represents a serializable object made available to corresponding timer callbacks	
void	setInfo (Serializable info)	
Boolean	isPersistent() true when timer is persistent.	
void	setPersistent (boolean b)	
String	toString()	

9.5.1 ScheduleExpression

A Programmatically created calendar-based timer can use the `ScheduleExpression` class to define the timer schedule.

This code creates a timer that expires every Monday at 12 noon.

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

9.6 Timeout Callbacks

There are two kinds of timeout callback methods for:

- timers created programmatically via a TimerService timer creation method.
- timers created automatically created via the @Schedule annotation or the deployment descriptor
- Timeout callback methods must not throw application exceptions.
- A timeout callback method must not be declared as final or static
- A timeout callback method can have public, private, protected, or package level access
- Since a timeout callback method is an internal method of the bean class, it has no client security context. When *getCallerPrincipal()* is called from within a timeout callback method, it returns the container's representation of the unauthenticated identity

- If the timer is a single-action timer, the container removes the timer after the timeout callback method has been successfully invoked (if invoked again, the container will throw a NoSuchObjectLocalException)
- If the bean invokes the *getNextTimeout()* or *getTimeRemaining()* method on the timer associated with a timeout callback while within the timeout callback, and there are no future timeouts for this calendar-based timer, a NoMoreTimeoutsException must be thrown.

9.6.1 @Timeout - Programmatic Timers – 1 timeout method

All timers created via one of the TimerService timer creation methods for a particular component use a single timeout callback method.

The timeout method may be a method annotated with the @Timeout annotation (or a method specified as a timeout method in the deployment descriptor) or the bean may implement the TimedObject interface.

A bean can have at most one timeout method for handling programmatic timers.

A bean that creates a programmatic timer must have one timeout method.

9.6.2 @Schedule / @Schedules - Automatic Timers – multiple timeout methods

Each automatically created timer is associated with a single timeout callback method.

A bean can have any number of automatically created timers.

Each timeout method is declared using the @Schedule / @Schedules annotation or the deployment descriptor.

9.7 The Timer Interface

Return	Method	throws Exception
void	cancel()	IllegalStateException if this method is invoked while the instance is in a state that does not allow access to this method NoSuchObjectLocalException if invoked on a timer that has expired or has been cancelled EJBException if this method could not complete due to a system-level failure
long	getTimeRemaining()	IllegalStateException NoSuchObjectLocalException NoMoreTimeoutsExceptions indicates that the timer has no future timeouts EJBException

Date	getNextTimeout()	IllegalStateException NoSuchObjectLocalException NoMoreTimeoutsExceptions EJBException
ScheduleExpression	getSchedule()	IllegalStateException NoSuchObjectLocalException EJBException
TimerHandle	getHandle()	IllegalStateException NoSuchObjectLocalException EJBException
Serializable	getInfo()	IllegalStateException NoSuchObjectLocalException EJBException
boolean	isPersistent()	IllegalStateException NoSuchObjectLocalException EJBException
boolean	isCalendarTimer()	IllegalStateException NoSuchObjectLocalException EJBException

9.8 TimerHandle Interface

A Timer object cannot be serialized, but a TimerHandle can. To get hold of the TimerHandle you can call **Timer.*getHandle()*** which returns a serializable handle to the timer. To get hold of the original Timer again you can call **TimerHandle.*getTimer()***.

Timer handles are only available for persistent timers.

Since timers are local objects, a TimerHandle must not be passed through a bean's remote business interface, or web service interface.

Return	Method	throws Exception
Timer	getTimer()	IllegalStateException NoSuchObjectLocalException EJBException

9.9 Transactions

An enterprise bean typically creates a timer within the scope of a transaction

If the transaction is rolled back, the timer creation is rolled back.

An enterprise bean typically cancels a timer within the scope of a transaction. If the transaction is rolled back, the container reverses the timer cancellation.

A timeout callback method on a bean with container-managed transactions has transaction attribute REQUIRED or REQUIRES_NEW

The container must begin a new transaction (container-managed transactions) prior to invoking the timeout callback method. If the transaction fails or is rolled back, the container must retry the timeout at least once.

In the event of a container crash or container shutdown, any

- single-event persistent timers that have expired during the intervening time before container restart must cause the corresponding timeout callback method to be invoked upon restart.
- interval persistent timers or schedule based persistent timers that have expired during the intervening time must cause the corresponding timeout callback method to be invoked at least once upon restart.

If the *setRollbackOnly()* method is invoked from within the timeout callback method, the container must rollback the transaction in which the timeout callback method is invoked. This has the effect of reversing the timer expiration. The container must retry the timeout *after* the transaction rollback.

10 Packaging

- The EJB deployment descriptor is optional for both a **.jar** file and a **.war** file
- In a **.jar** file, the deployment descriptor is stored with the name META-INF/ejb-jar.xml
- In a **.war** file, the deployment descriptor is stored with the name WEB-INF/ejb-jar.xml
- The **.jar** file or **.war** file must also contain, either by inclusion or by reference, the class files for all the classes and interfaces that each enterprise bean class and/or web service endpoints depend on, (except Java EE and Java SE classes)
- An **.jar** file in a **.war**'s WEB-INF/lib that contains enterprise beans is not considered an independent Java EE "module" (namespaces are scoped to the enclosing **.war** file)
- The packaging of enterprise bean classes in a **.jar** file in the WEB-INF/lib is semantically equivalent to packaging the classes within WEB-INF/classes.
- if you use <metadata-complete>true</metadata-complete> in an *ejb-jar.xml* or a *web.xml*, no annotations will be processed.

10.1 EJB Client Jar

The **.jar** file or **.war** file producer can create an ejb-client JAR file. The ejb-client JAR file contains all the class files that a client program needs to use the client view of the enterprise beans that are contained in the **.jar** file or **.war** file.

The ejb-client JAR file is specified in the deployment descriptor of the **.jar** file or **.war** file using the optional <ejb-client-jar> element.

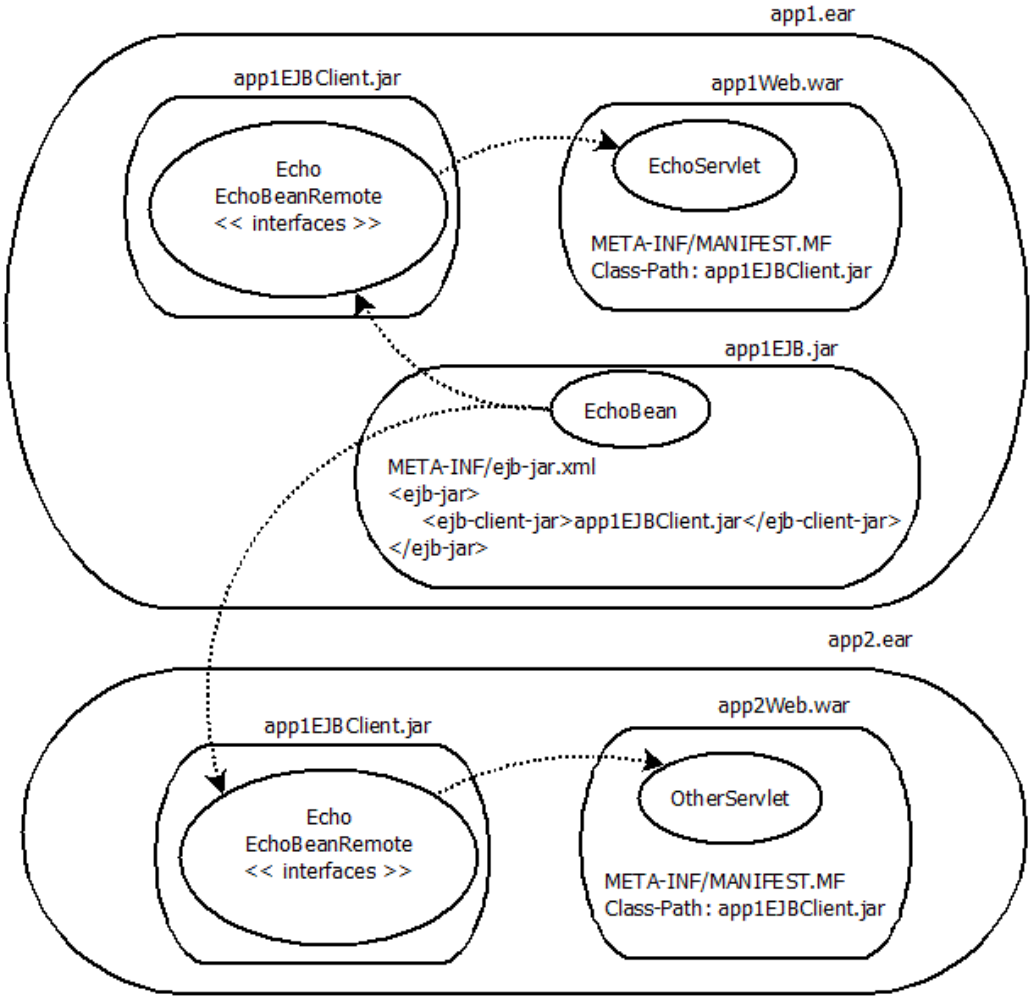
The EJB specification does not specify whether a **.jar** file or **.war** file should include by copy or by reference the classes that are in an ejb-client JAR file.

- by copy:
    - include all the class files in the ejb-client JAR file also in the **.jar** file or **.war** file
- by reference:
    - use a Manifest Class-Path entry in the **.jar** file or **.war** file to specify that it depends on the ejb-client JAR at runtime.

10.2 Example packaging

We have got 2 Enterprise Applications **app1.ear** and **app2.ear** and they both make use of the remote EJB EchoBean. The EchoBean EJB is in **app1EJB.jar** and is packaged in **app1.ear**. The interfaces for the EchoBean are packaged in the ejb-client-jar **app1EJBClient.jar**. Both **app1Web.war** (app1.ear) and **app2Web.war** (app2.ear) make use of the EchoBean.

10.2.1 Graph of example



In the following paragraphs the contents of the Enterprise Application aRchives (. ear's) are described.

10.2.2 app1.ear

De application.xml file of app1 contains 3 modules: the ejb jar file, the ejb-client jar file and the .war module.

```
+---app1
|   \---META-INF
|           application.xml
```

application.xml

```
<application>
  <display-name>app1</display-name>
  <module>
    <web>
      <web-uri>app1Web.war</web-uri>
      <context-root>app1Web</context-root>
    </web>
  </module>
  <module>
    <ejb>app1EJB.jar</ejb>
  </module>
  <module>
    <java>app1EJBClient.jar</java>
  </module>
</application>
```

10.2.2.1 app1EJBClient.jar

The ejb client jar file contains are business interfaces and classes that are declared in those interfaces (here we only have interfaces).

```
+---app1EJBClient.jar
|   \---nl
|       \---notes
|           \---ejb
|                   Echo.class
|                   EchoBeanRemote.class
```

Echo

```
package nl.notes.ejb;
// import statements left out
@Remote
public interface Echo {
   public String echo(String phraseToEcho);
```

```
}
```

EchoBeanRemote

```
package nl.notes.ejb;
// import statements left out
@Remote
public interface EchoBeanRemote extends Echo {
	@Override
	public String echo (String phraseToEcho);
}
```

10.2.2.2 app1EJB.jar

This jar contains the EJB implementation class. In the deployment descriptor (ejb-jar.xml) you will have to put the <ejb-client-jar> element. As the EchoBean class implements a business interface EchoBeanRemote, the MANIFEST.MF file is used to indicate that the interfaces can be found in the ejb-client-jar on the classpath.

```
+---app1EJB.jar
|   +---META-INF
|   |       ejb-jar.xml
|   |       MANIFEST.MF
|   \---nl
|       \---notes
|           \---ejb
|                   EchoBean.class
```

ejb-jar.xml

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>EchoBean</ejb-name>
      <ejb-class>nl.notes.ejb.EchoBean</ejb-class>
      <session-type>Stateless</session-type>
      <env-entry>
        <env-entry-name>initNumber</env-entry-name>
        <env-entry-type>java.lang.Integer</env-entry-type>
        <env-entry-value>10</env-entry-value>
        <injection-target>
          <injection-target-class>nl.notes.ejb.EchoBean</injection-target-class>
          <injection-target-name>init</injection-target-name>
        </injection-target>
      </env-entry>
```

```
    </session>
  </enterprise-beans>
  <ejb-client-jar>app1EJBClient.jar</ejb-client-jar>
</ejb-jar>
```

MANIFEST.MF

```
Manifest-Version: 1.0
Class-Path: app1EJBClient.jar
```

EchoBean

```
package nl.notes.ejb;
// import statements left out
@Stateless
public class EchoBean implements EchoBeanRemote {
       Integer init;
       @Override
       public String echo(String phraseToEcho) {
           System.out.println("echo:  "+ "environment entry = " +
                                                 init + ", " + phraseToEcho);
          return phraseToEcho;
       }
}
```

10.2.2.3 app1Web.war

The web-application needs the business interface that is contained in the ejb-client-jar and therefore a MANIFEST.MF classpath entry is used.

```
+---app1Web.war
|   +---META-INF
|   |       MANIFEST.MF
|   \---WEB-INF
|       |   web.xml
|       +---classes
|       |   \---nl
|       |       \---notes
|       |           \---servlet
|       |                   EchoServlet.class
```

MANIFEST.MF

```
Manifest-Version: 1.0
Class-Path: app1EJBClient.jar
```

web.xml

```
<web-app>
  <display-name>app1Web</display-name>
  <welcome-file-list>
    <welcome-file>EchoServlet</welcome-file>
  </welcome-file-list>
</web-app>
```

EchoServlet

```
package nl.notes.servlet;
// import statements left out
@WebServlet("/EchoServlet")
public class EchoServlet extends HttpServlet {
       @EJB EchoBeanRemote echo;
       @Override
       protected void doGet(HttpServletRequest request,
              HttpServletResponse response) throws ServletException, IOException {
          ServletOutputStream out = response.getOutputStream();
          out.print("<html><body>");
          out.print(echo.echo("echo from the app1"));
          out.print("</body></html>");
       }
}
```

10.2.3 app2.ear

The second enterprise application just contains one web-application module.

```
+---app2
|   \---META-INF
|           application.xml
```

application.xml

```
<application>
  <display-name>app2</display-name>
  <module>
    <web>
      <web-uri>app2Web.war</web-uri>
      <context-root>app2Web</context-root>
    </web>
  </module>
</application>
```

10.2.3.1 app1EJBClient.jar

The ejb-client-jar is exactly the same as the one in the app1.ear

```
+---app1EJBClient.jar
|   |   \---nl
|   |       \---notes
|   |           \---ejb
|   |                   Echo.class
|   |                   EchoBeanRemote.class
```

See 10.2.2.1

10.2.3.2 app2Web.war

As the web-application makes use of the EchoBeanRemote business interface a classpath entry in the Manifest file is used.

```
+---app2Web.war
|   |   +---META-INF
|   |   |       MANIFEST.MF
|   |   \---WEB-INF
|   |       |   web.xml
|   |       +---classes
|   |       |   \---nl
|   |       |       \---notes
|   |       |           \---servlet
|   |       |                   OtherServlet.class
```

MANIFEST.MF

```
Manifest-Version: 1.0
Class-Path: app1EJBClient.jar
```

web.xml

```
<web-app>
  <display-name>app2Web</display-name>
  <welcome-file-list>
    <welcome-file>OtherServlet</welcome-file>
  </welcome-file-list>
</web-app>
```

OtherServlet

```
package nl.notes.servlet;
// import statements left out
@WebServlet("/OtherServlet")
public class OtherServlet extends HttpServlet {
```

```
        @EJB EchoBeanRemote echo;
        @Override
        protected void doGet(HttpServletRequest request,
                HttpServletResponse response) throws ServletException, IOException {
           ServletOutputStream out = response.getOutputStream();
           out.print("<html><body>");
           out.print(echo.echo("echo from the app2"));
           out.print("</body></html>");
        }
}
```

10.2.4 Running the example

Add both app1.ear and app2.ear to a server and start the server

```
URL: http://localhost:8080/app1Web/
Output on browser:
        echo from the app1
Output on console:
        INFO: echo:  environment entry = 10, echo from the app1
```

```
URL: http://localhost:8080/app2Web/
Output on browser:
        echo from the app2
Output on console:
        INFO: echo:  environment entry = 10, echo from the app2
```

Note: the web-applications make use of a <welcome-file> entry in the web.xml. When the URL http://localhost:8080/app1Web/ is requested the container will add the welcome-file entry behind the URL to see whether that URL is part of the web-application. So in the example of the app1Web application: the container constructs the following URL with the welcome file entry “EchoServlet”: http://localhost:8080/app1Web/EchoServlet and after that the EchoServlet is invoked by the container.

11 Runtime Environment

11.1 EJB Lite vs EJB Full API

	EJB 3.1 Lite	Full EJB 3.1 API
Session beans	Yes	Yes
Message-driven Beans	No	Yes
Java Persistence 2.0	Yes	Yes
Session Bean - Local business interface	Yes	Yes
Session Bean - No-interface	Yes	Yes
Session Bean - Remote business interface	No	Yes
JAX-WS Web Service Endpoint	No	Yes
JAX-RPC Web Service Endpoint	No	Yes
EJB Timer Service	No	Yes
Asynchronous session bean invocations	No	Yes
Interceptors	Yes	Yes
RMI-IIOP Interoperability	No	Yes
Container-managed transactions	Yes	Yes
Bean-managed transactions	Yes	Yes
Declarative and Programmatic Security	Yes	Yes
Embeddable API	Yes	Yes

Note: the *Java EE Web Profile* includes EJB 3.1 Lite

11.2 Restrictions for EJBs

An Enterprise Java Bean is not allowed to use:

- Read/write static fields. Using read-only static fields (declared final) is allowed.
- Thread synchronization (e.g. *synchronize(), wait(), notify()…*) (except bean-managed Singleton Session Beans).
- The AWT functionality to attempt to output information to a display
- Input information from a keyboard.
- The java.io package to attempt to access files and directories in the file system.

The enterprise bean must not attempt to:

- Listen on a socket, accept connections on a socket, or use a socket for multicast.
- Interact with a class loader.
- Manage threads. (e.g. start, stop, suspend)
- Load a native library.
- To access or modify the security configuration objects (Policy, Security, Provider, Signer, and Identity).
- To pass *this* as an argument or method result. The enterprise bean must pass the result of SessionContext.*getBusinessObject()* instead.

12 Embeddable Usage

Instead of the Java EE server-based execution of the EJB container, the embeddable container allows client code and its corresponding enterprise java beans to run within the same JVM and class loader. This provides better support for testing, offline processing (e.g. batch), and the use of the EJB programming model in desktop applications.

Embeddable usage requirements allow client code to instantiate an EJB container that runs within its own JVM and classloader. The embeddable container is instantiated using a bootstrapping API.

- `EJBContainer ec = EJBContainer.createEJBContainer();`

By default, the embeddable container searches the JVM classpath (the value of the Java System property java.class.path) to find the set of EJB modules for initialization. A classpath entry is considered a matching entry if it meets one of the following criteria:

- It is an ejb-jar according to the standard module-type identification rules defined by the Java EE platform specification
- It is a directory containing a META-INF/ejb-jar.xml file or at least one .class with an enterprise bean component-defining annotation

12.1 Starting an embeddable container

Example: a Client (with the main() method) will load the EJB-modules *ejb-foo.jar* and *ejb-bar.jar* into the embeddable container which implementation is in the *embeddable-cont-ejb.jar*

```
java -classpath foo.jar:bar.jar:embeddable-cont-ejb.jar:client.jar com.acme.Client
```

You can start the EJBContainer with a set of properties for customization:

```
Properties props = new Properties();
props.setProperty(EJBContainer.MODULES, "bar");
EJBContainer ec = EJBContainer.createEJBContainer(props);
```

12.2 Stopping the embeddable container

It is not required to call close() but its use is recommended for optimal resource cleanup. The embeddable container must call the PreDestroy method(s) of any Singleton bean instances in the application.

Use:

- `EJBContainer.close()`

12.3 JNDI lookup of Session Bean

Lookup of local and no-interface view Session Beans is done with the global JNDI namespace:

```
Context ctx = EJBContainer.getContext();
FooLocal foo = (FooLocal) ctx.lookup("java:global/foo/FooBean");
```

13 Annotation Summary

Client View		
	@LocalBean	
	@Local	
	@Remote	
Session Bean		
	@PostConstruct	
	@PreDestroy	
	@Asynchronous	
	@Stateless	
	@Stateful	
		@PrePassivate
		@PostActivate
		@Remove
		@AfterBegin
		@BeforeCompletion
		@AfterCompletion
		@StatefulTimeout
	@Singleton	
		@Startup
		@DependsOn
		@Lock
		@ConcurrencyManagement
		@AccessTimeout
Message Driven Bean		
	@PostConstruct	
	@PreDestroy	
	@Asynchronous	
	@MessageDriven	
		@ActivationConfigProperty
Interceptor		
	@Interceptor	@Interceptors
	@AroundInvoke	
	@AroundTimeout	
	@ExcludeDefaultInterceptors	
	@ExcludeClassInterceptors	
Transaction		
	@TransactionManagement	
	@TransactionAttribute	
Exception		
	@ApplicationException	
Injection and ENC		
	@EJB	@EJBs
	@Resource	@Resources
Security		
	@RunAs	
	@DeclareRoles	
	@RolesAllowed	
	@PermitAll	
	@DenyAll	
Timer Service		
	@Schedule	@Schedules
	@Timeout	