



pointing the way

White Papers

EJB

Callbacks and Interceptors



Introduction

This paper describes Enterprise JavaBeans 3.0 call back events and interceptor mechanism.

Interceptors

Interceptors are designed to implement objects behaviour in more efficient way. They provide separation of behaviour code and business logic.

The code is clearer to read and understand.

Interceptors solve following problems:

- ✓ Keep business logic clean from not business related code
- ✓ Allow to easily turn off some code behaviour(e.g. security, monitoring)
- ✓ Allow to apply same behaviour to many other components.

Interceptors can be defined for session beans (both stateful and stateless) and message-driven beans.

EJB interceptors' concept is similar to Aspect-Oriented Programming (AOP) but it is not that flexible. The most counter to AOP feature is that interceptors have to be declared explicitly in code.

Interceptor Definition

Interceptor is a typical java class that has methods annotated with `@javax.interceptor.AroundInvoke`. Such interceptor's methods are invoked in the same stack and context (transactions, security) that method which is intercepted.

Interceptor method signature:

`@AroundInvoke`

Object name(`InvocationContext invCtx`) throws Exception;

`javax.interceptor.InvocationContext` is a representation of business method invoked by client. It is possible to access:

- ✓ Invocation object (object that defines intercepted method)

EJB – Callbacks and Interceptors



- ✓ Method parameters (stored in array)
- ✓ Reference to `java.lang.reflect. method` that represents invoked method.

There are two rules to remember about EJB interceptors:

- ✓ Intercepted method is not invoked yet
- ✓ Interceptor's method is responsible for business method invocation (or cancelling invocation) and returning a result.

To invoke an intercepted method in interceptor's method call "proceed()" method on `InvocationContext` object. It is not needed to pass parameters because `InvocationContext` is aware of its context. Sometimes interceptors have to change input parameters – this can be done by `setParameters(Object[] newParams)` method.

Interceptor's method is able to omit business method invocation but it still has to return a result or throw an exception. In case of returning result it does not have to be even a result returned by business method. Interceptor can modify a business method result, replace it or return a null value.

As we see interceptor's method has a full control of what happens before and after business method invocation or if given method is even invoked or not. Business method parameters and result can be freely modified by interceptor.

Injecting Interceptors

Interceptors can be injected by using annotations or in XML descriptor. The easiest way is to add `@javax.interceptor.Interceptors(Intercl.class)` annotation:

- ✓ before business class definition, or before class definition (then interceptor will apply to all methods in given class).
- ✓ the more flexible way is to use XML descriptor:

```
<interceptor-binding>
  <ejb-name>BusinessBean</ejb-name>
  <interceptor-calss>Interceptor1
</interceptor-class>
  <method-name>myBusinessMethod
</method-name>
  <method-params>...</method-params>
</interceptor-binding>
```

XML descriptor allows changing an interceptor association during deployment without need of recompiling java sources.

Using annotations or XML descriptor should also concern if given interceptor's behavior is a business or environment deployment process. For example time measuring of business method execution is an environment aspect.

Assigning interceptors to all beans is possible via XML descriptor:

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

Default interceptors can be exclude using annotations before class declaration:

```
@ExcludeDefaultInterceptors.
```

To exclude interceptor on method level use:

- ✓ `@ExcludeClassInterceptor`, to exclude interceptor declared on class level,
- ✓ `@ExcludeDefaultInterceptors`, to exclude interceptors defined in descriptor.

Interceptors are kept in the same application context that object they intercept. It means that they can use `@Resources`, `@EJB`, `@PersistenceContext` annotations to perform injections.

Intercepting Life-cycle Events

Interceptors according to EJB specification allow intercepting life-cycle events' methods (see Callback Events pages). Interceptor signature is as follow:

```
@callbackEventAnnotation void
name(InvocationContext ctx);
```

EJB – Callbacks and Interceptors



In this case interceptor's methods can not throw checked exceptions and return any value. It is not guaranteed that listener for given intercepted callback event exist (because its definition is optional). Because of that `InvocationContext.getMethod()` always returns null.

Callback Events

EJB 3.0 specification defines callback events that apply for session, persistence and message-driven beans. Every component type has own callbacks events depending on its life-cycle.

Callback Events

Using annotations it is possible to make stateful session bean aware of life-cycle processes. For example method annotated with `@javax.ejb.PostActivate` is invoked after successful process of bean activation. Method annotated with `@javax.ejb.PrePassivate` is invoked before passivation process.

For entities exist a bunch of callback events annotations from `javax.persistence` package:

Event	Description
<code>PrePersist</code>	Before object is persisted
<code>PostPersist</code>	After object was successful persisted
<code>PostLoad</code>	Fires after entity was read by <code>find()</code> or <code>getReference()</code> method of Entity Manager
<code>PreUpdate</code>	Before synchronizing entity state with database
<code>PostUpdate</code>	After entity synchronization with database
<code>PreRemove</code>	Before removing entity from database
<code>PostRemove</code>	After deleting the row in database related to given entity

Entity's callback events can be implemented in entity class or entity listener. Entity listener's class intercepts callback events related with given entity.

Listener's methods do not return any type, throws only verified exceptions and take one argument that represents an entity.

`@PostPersist` void `postInsert(Object entity){...}`
Entity listener must implement a no-argument public constructor. To bind listeners to entity use following annotation before entity class declaration:

`@EntityListeners({Listener1.class, Listener2.class})`
It is possible to define default listeners for all entities in XML descriptor:

```
<entity-mappings>
  <entity-listeners>
    <entity-listener class="Listener"/>
  </entity-listeners>
</entity-mappings>
```

To exclude default listener in given entity use `@ExcludeDefaultListeners` annotation before entity class declaration.

Listeners are inherited from super class but they also can be excluded using

`@ExcludeSuperClassListeners` annotation.

Chaining listeners' invocation follows simple rules:

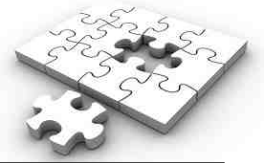
- ✓ First listener declared in listeners list executes as first
- ✓ Superclass listeners executes before descendant listeners (in order followed by rule above)
- ✓ Callback events handlers defined in entity class executes at the end.

Conclusion

Interceptor's mechanism allows creating applications code more clear and understandable.

Interceptors separate non-business-related code aspect from business logic. Behaviour introduced by interceptor can be easily turned off or assigned to many business components.

EJB – Callbacks and Interceptors



Callbacks events give an EJB programmer possibility to by handle cases related to components life-cycle. That gives a more control of beans behaviour, so developer can make sure that business logic executes properly despite EJB container management actions.

Links

Bill Burke, Richard Monson-Haefel – “Enterprise JavaBeans 3.0”. O’Reilly 2007

Enterprise JavaBeans Technology, EJB 3.0 Specification <http://java.sun.com/products/ejb/>

EJB 3 and AOP
http://blogs.codehaus.org/people/avasseur/archives/001002_ejb_3_and_aop_the_ejb_interceptor_dilemma.html