Enterprise POJOs

How EJB3 and AOP Simplify Development

Bill Burke

March 8, 2005

Agenda

- Overview
- JDK 5 Annotations
- EJB 3.0 Pojos
- EJB 3.0 extensions
- Create new Java syntax

What makes a good framework?

- Easy to learn
- Quick to prototype
- No special tools
  - ✓ Shouldn't need special IDE plugins
- Easy to extend
  - ✓ Advanced users need flexibility
- Easy to test
  - ✓ Can be tested inside and outside a container

JDK 5 Annotations

- XDoclet like metadata, C#-like metatags
  - ✓ Typesafe, compiler-checked
- Syntax you can add to the Java language
- Metadata you can attach to class, method, field, constructor, or parameter
  - ✓ Metadata is compiled into class file
  - ✓ Available at compile time and/or runtime
- EJB 3.0 makes extensive use of annotations to replace XML

JDK 5 Annotations

```
public @interface MyMetadata {
        String value() default "hello";
}
-----------------------------------------------------
import org.acme.MyMetadata;

@MyMetadata("stuff")
public class MyClass
{
        @MyMetadata
        public void someMethod() {...}
}
```


EJB 3.0

First step towards Enterprise POJOs

Goals of EJB 3.0

- Simplify programming model
  - ✓ Reduce number of artifacts
  - ✓ Remove dependence on API
  - ✓ Write POJOs
  - ✓ Get out of "XML Hell"
  - ✓ Exploit new JDK 5 features
- Facilitate test-driven development
- Increase EJB developer base

Goals of EJB 3.0

- Remove unnecessary interfaces
- Remove unnecessary callbacks
- Make deployment descriptors optional
- Make beans pojo-like
- Use default values where they make sense

Session Beans

POJOs with attitude

Required Interfaces

- Homeless
- Methods don't throw RemoteException
  - ✓ Remove dependency on RMI
- No verbose interfaces to implement
  - ✓ Remove dependency on EJB APIs

```
@Remote
public interface ShoppingCart
{
        public void addItem(int product, int quantity);
        public void checkout();
}
```

Bean class

- No verbose interfaces to implement
  - ✓ Remove dependency on EJB APIs
- It's a POJO!

```
@Stateful
public class ShoppingCartBean implements ShoppingCart {
        public void addItem(int product, int quantity) {
                ...
        }
        @Remove
        public void checkout() {
                ...
        }
}
```

Metadata: Tx and Security

```
@Stateful
public class ShoppingCartBean implements ShoppingCart
{
        @TransactionAttribute(NOT_SUPPORTED)
        public void addItem(int product, int quantity) {
                ...
        }

        @Remove
        @MethodPermission({"registered_customer"})
        public void checkout() {
                ...;
        }
}
```

Unchecked default

REQUIRED default

Message Driven Beans

MDB

- Just implements MessageListener
- XML turns into annotations.

```
@MessageDriven(
    activation={
        ActivationSpecAttribute(name="destinationType", value="javax.jms.queue"),
        ActivationSpectAttribute(name="destination", value="queue/email")
    }
)
public class EmailBean implements MessageListener {

        void onMessage(Message msg) { }

}
```


Dependency Injection

I'll take a Big Mac, fries, and a coke please

Dependency Injection

- Tell the container what you want
- Injection through a field or setter

```
@Stateful public class ShoppingCartBean implements ShoppingCart {

        @Inject protected SessionContext ctx;

        @EJB(name="CreditProcessorEJB")
        protected CreditCardProcessor processor;

        private DataSource jdbc;
        @Resource(jndiName="java:/DefaultDS")
        public void setDataSource(DataSource db) { this.jdbc = db; }
}
```


Dependency Injection

- Facilitates test-driven development
- Test outside of the container
- Easier to write/inject mock objects

Callbacks

Call me back when you get a chance please!

Callbacks

- Callback methods still available
- Annotate on an as-needed basis

```
@Stateless
public class FacadeBean implements Facade {
        @EjbTimeout void licenseExpired(Timer t) {...} // committee is leaning towards this

        @PostCreate public void initialize() {}
        @PreDestroy public void destroy() {}
}
```

Deployment descriptors

But I like XML!

EJB 3 Deployment descriptors

- Believe it or not, people like XML deployment descriptors
  - ✓ Externalize configuration
  - ✓ Externalize system architecture
  - ✓ Deploy same bean class in different ways
- Although not in draft, XML DDs are optional
- Replace annotations with XML and you get pure POJOs!

EJB 3 Deployment Descriptors

```
@Remote public interface ShoppingCart {
        public void addItem(int prodId, int quantity);
        public void checkout();
}
@Stateful
public class ShoppingCartBean implements ShoppingCart {
        @Inject private EntityManager manager;
        ...

        @Remove
        @TransactionAttribute(REQUIRED)
        @MethodPermission({"valid_customer"})
        public void checkout() {
          ...
        }
```

EJB 3 Deployment Descriptors

```
public interface ShoppingCart {
        public void addItem(int prodId, int quantity);
        public void checkout();
}


public class ShoppingCartBean implements ShoppingCart {
         private EntityManager manager;
         ...

         public void checkout() {
           ...
         }
}
```

JBoss Extensions to EJB 3.0

From EJBs to Enterprise POJOs

The Service Bean

Service bean

- Singleton services
- Multi-threaded and stateful
- Lifecycle and dependency management
- Next iteration of the JBoss MBean

Service bean

- Full EJB 3 library of annotations available
  - ✓ @TransactionAttribute, @MethodPermissions, @Inject
- @Remote and @Local interface publishing
- New @Management interface
  - ✓ JMX is now an aspect

Service interfaces

- @Remote, @Local
- Just like EJB

```
@Remote
public class CalculatorRemote
{
        BigDecimal add(float x, float y);
        BigDecimal divide(float x, float y);
}
```


Service interfaces

- @Management
- JMX interface, JMX aspect

```
@Management
public class CalculatorManagement
{
        int getPrecision();
        void setPrecision();
}
```


Service bean class

- @Service deploys a singleton
- Multi-threaded and stateful

```
@Service
public class CalculatorService implements CalculatorRemote, CalculatorManagement
{
        private int precision;

        public int getPrecision() { return precision; }
        void setPrecision(int p) { precision = p; }

        BigDecimal divide(float x, float y) {...}
}
```

Message Driven POJOs

Typed MDBs

Message Driven POJOs

- MDBs with a typed interface
- MDBs that publish a well known interface
- Simplify JMS
- Invoke methods, don't build messages
- Cleaner, clearer code

Message Driven POJOs

- Define a @Consumer bean class
- @Consumer implements 1-N @Producer interfaces
- @Producer a means to send JMS messages
- @Consumer a means to receive JMS messages

@Consumer

```
@Consumer(activation={
        ActivationSpecAttribute(name="destinationType", value="javax.jms.queue"),
        ActivationSpectAttribute(name="destination", value="queue/foo")
    })
public class QueueConsumerBean implements QueueRemote, QueueXA {

        public void method1(String arg1, int arg2) {
          ...
        }

        public void method2(HashMap map) {
          ...
        }
}
```


@Producer

```
@Producer
public interface QueueRemote {
        public void method1(String arg1, int arg2);
        public void method2(HashMap map);

}
```

```
@Producer(connectionFactory="java:/JmsXA")
public interface QueueXA {
        public void method1(String arg1, int arg2);
        public void method2(HashMap map);

}
```


Message Driven POJOs

- Consumer registers all implements @Producer interfaces into JNDI
- Clients look up these @Producers in JNDI
- Clients invoke methods on the Producers
- Methods turn into JMS messages and published to the queue/topic

@Producer

- Producers implicitly extend ProducerObject interface

```
public interface ProducerObject {
   ProducerManager getProducerManager();
}
public interface ProducerManager {
   void setUsername(String user);
   void setPassword(String passwd);
   void connect() throws JMSException;
   void close() throws JMSException;
   void commit() throws JMSException;
   void rollback() throws JMSException;
}
```

37

Using a Producer

- Producers implicitly extend ProducerObject interface

```
{
  QueueRemote queue = (QueueRemote)ctx.lookup(QueueRemote.class.getName());

  ProducerManager con = ((ProducerObject)queue).getProducerManager();

  con.connect();
  queue.method1("hello", 5);
  queue.method2(new HashMap());
  con.close();
}
```

38

Configuring Message Properties

```
@Producer(transacted=true)
@MessageProperties(delivery=NON_PERSISTENT)
public interface QueueRemote {

        @MessageProperties(delivery=PERSISTENT, timeToLive=1000, priority=1)
        public void method1(String arg1, int arg2);

        public void method2(HashMap map);

}
```

39

Asynchronous EJBs

Lightweight asynchronous communication

Asynchronous EJBs

- Goals
  - ✓ Provide a lightweight asynchronous API for all EJB types
  - ✓ VM local and Remote capabilities
  - ✓ Zero changes to existing interfaces/code
  - ✓ Support obtaining method results asychronously
    - Not just oneway.

41

Asynchronous EJBs

- Available for Stateless, Stateful, and Service
- Each @Remote and @Local interface implements EJBProxy

```
public interface EJBProxy  {
        FutureProvider getAsynchronousProxy();
}
...

CalculatorRemote calc = (CalculatorRemote)ctx.lookup("calculator");
EJBProxy proxy = (EJBProxy)calc;
CalculatorRemote calcAsynch = (CalculatorRemote)proxy.getAsynchronousProxy();
```

42

Asynchronous EJBs

- Asynchronous Proxy same exact interface
- Invoking methods on proxy happen in background

```
CalculatorRemote calc = (CalculatorRemote)ctx.lookup("calculator");
EJBProxy proxy = (EJBProxy)calc;
CalculatorRemote calcAsynch = (CalculatorRemote)proxy.getAsynchronousProxy();

calcAsynch.add(1, 1);      // invoked in background, in another thread

calcAsynch.divide(10, 2); // invoked in background
```

43

Asynchronous EJBs

- Obtain response asynchronously
  - ✓ FutureProvider and Future interfaces

```
FutureProvider provider = (FutureProvider)calcAsynch;

calcAsynch.add(1, 1);                  // invoked in background, in another thread
Future result1 = provider.getFuture();
calcAsynch.divide(10, 2);              // invoked in background
Future result2 = provider.getFuture();

int val1 = result1.get();              // block until first method call returns
if (result2.isDone()) {                // poll to see if done
  int val2 = result2.get();
}
```

44

JBoss Aspect Library

Embedding middleware directly into Java Language

Aspect Library

- Bring the enterprise directly to the Java language
- J2EE as a library rather than as a server
- Fine-grain middleware a la carte

46

EJB a la carte

- Transactions everywhere
- Fine-grain security

```
public class POJO {
    @Permissions({"ROOT"})
    public POJO () {
      // secured constructor
    }

    @TransactionAttribute(REQUIRED)
    public static void someMetho() {
      // transactions on static methods
    }
}
```

47

Transactional Properties

- Objects/fields with transactional state
- Objects with transactional locking

```
@Transactional
public class POJO {

    @TxLocked
    public void someMethod() {
      // transactions on static methods
    }
}
```

48

Concurrency and Locking

- Latches, Semaphores, Read/Write locks

```
@Latched
public class POJO {...}

@Semaphored(permits=2)
public class AnotherPOJO {...}

public class YetAnother {

   @ReadLocked void method() {...}
   @WriteLocked void method2() {...}
}
```

Concurrency and Locking

- Per method locks

```
public class POJO {

   // one and only one thread at a time can access this method
   @MutexedJoinpoint
   public static void method()
   {...}

    // only 5 threads are allowed to call this method at one time
   @SemaphoredJoinpoint(permits=5)
   public void method2() {...}
}
```

Hibernate Aspects

- Transactional Sessions
- Inject session associated with current Tx
- No session? Create one and register with TX
- Session automatically flushed and closed at end of TX

```
public class POJO {

   @TxSession(name="InventoryFactory)
   private HibernateSession session;

}
```

Conclusion

- We're trying to make things:
  - ✓ Easier to use
  - ✓ Easier to extend
  - ✓ Easier to package
  - ✓ Easier to deploy
- Come to JBoss AOP BOF to learn how to write your own extensions!