JavaPolis 2004

Enterprise JavaBeans 3.0

Linda DeMichiel

Sun Microsystems

Agenda

- Goals of EJB 3.0
- Overview of the Simplified EJB API
- Persistence Update
- Summary and Current Status

JAVAPOLIS

EJB 3.0 Goals

- Focus: Ease of Development
- Simplify EJB: make it easier to use
  - Simplified set of APIs
  - Reduce the number of classes the developer needs to produce
  - Eliminate deployment descriptor from developer's view
  - Improve developer productivity
- Capture broader range of developers
  - Make it simpler for average developer
  - Increase developer base, target more corporate developers

EJB 3.0 Goals

- Retain / improve power of EJB 2.1
- Ensure EJB 2.1 APIs are always available
  - Provide interoperability between new APIs and existing EJB 2.1 APIs
  - Provide for reuse of existing components in new applications
  - Support migration scenarios
  - Simplifications / enhancements to improve use of these APIs as well
- Entity Beans
  - Entity Beans are one of the more complex areas of EJB 2.1
  - Need to simplify / improve EJB persistence as well

Approaches to Simplification

- Leverage Java language metadata
- Configuration by exception
- Simplification of all bean types
  - Beans as “POJOs”
  - Elimination of unnecessary interfaces
- Simplification of environmental access
  - Injection
- Lighter-weight entities
  - Focus on object/relational mapping
  - Expansion of EJB QL
- More work done by container; less by developer

Metadata Annotations

- Java language metadata (JSR 175) introduced as part of J2SE™ 5
- Leveraged heavily in EJB 3.0
  - Eliminate need for developer to provide deployment descriptor
  - Demarcation of injection and interpositioning points for container behavior
  - Specification of object/relational mapping
  - Reduce the number of interfaces programmer needs to implement
  - Reduce the number of classes programmer needs to produce
  - Gives developer simple-to-use and yet powerful capability

Metadata Annotations

- Configuration by exception
  - Reduce need to explicitly specify common, expected behaviors and requirements on container
  - Leverage defaulting in conjunction with metadata
  - Defaults for annotation members
  - Metadata itself used for non-default cases (or for program documentation)
- Preservation of ability to use XML
  - As alternative mechanism
  - For overriding of annotations
  - For supplementing annotations

Simplification of EJB Bean Types

- More closely resemble “POJOs”
- Elimination of requirements for EJB component interfaces
  - Business interfaces are plain Java interfaces, not EJBObject/EJBLocalObject interfaces
- Elimination of requirment for home interfaces
  - Use dependency injection or simple bean lookup
- Elimination of requirement to implement SessionBean, MessageDrivenBean,… interfaces
  - Use annotations for (now optional) callbacks
- Elimination of all required interfaces for Entity Beans

EJB 2.1 Stateless Session Bean

```
public interface Calculator extends EJBObject {
  public int add(int a, int b)
    throws RemoteException;
  public int subtract(int a, int b)
    throws RemoteException;
}

public interface CalculatorHome extends EJBHome {
  public Calculator create()
    throws CreateException, RemoteException;
}
```

Stateless Session Bean Class

```
public class CalculatorBean implements SessionBean{
  private SessionContext ctx;
  public void setSessionContext(SessionContext ctx){
    this.ctx=ctx;
  }
  public void ejbCreate() {}
  public void ejbActivate() {}
  public void ejbPassivate() {}
  public void ejbRemove() {}
  public int add(int a, int b) {
    return a + b;
  }
  public int subtract(int a, int b) {
    return a - b;
  }
}
```

Deployment Descriptor

```
<session>
<ejb-name>CalculatorEJB</ejb-name>
<home>com.example.CalculatorHome</home>
<remote>com.example.Calculator</remote>
<ejb-class>com.example.CalculatorBean</ejb-class>
<session-type>Stateless</session-type>
<transaction-type>Container</transaction-type>
</session>
…
…
```

Same Example: EJB 3.0

```
@Stateless public class CalculatorBean
    implements Calculator {

  public int add(int a, int b) {
    return a + b;
  }
  public int subtract(int a, int b) {
    return a - b;
  }
}


public interface Calculator {
  public int add(int a, int b);
  public int subtract(int a, int b);
}
```

Same Example: Generated Interface

```
@Stateless public class CalculatorBean {

  public int add(int a, int b) {
    return a + b;
  }
  public int subtract(int a, int b) {
    return a - b;
  }
}
```

Deployment Descriptor

Simplifications in this Example

- Elimination of Home interface
- Business interface is a plain Java interface
  - Bean can implement it or it can be generated
  - Bean can have more than one business interface
  - Can support remote access
  - EJB(Local)Object removed from client view
  - RemoteExceptions removed from client view
- No Implementation of SessionBean interface

Metadata and Defaulting

- Defaulting of transaction management types
- Defaulting of transaction attributes
- Default use of unchecked methods
- Default use of caller identity
- Defaulting of interface generation
- Defaulting of names
- Defaulting of O/R mapping (for Entities)
- Etc,…

Metadata annotations are available to further specify / customize all of these

Stateful Session Bean Example

```
@Stateful public class ShoppingCartBean {

  private String customer

  @Init public void startToShop (String customer){
      this.customer = customer;
      …
  }

  public void addToCart (Item item) {
      …
  }

  @Remove public void finishShopping () {
      …
  }
}
```

Simplified Environment Access

- Get JNDI APIs out of developer's view
- Express dependencies in metadata
- Injection of resources
  - Instance variable injection
  - Setter injection
- Simple lookup methods
  - lookup method added to EJBContext

Instance Variable Injection

```
@Stateless public class MySessionBean {

  @Resource public DataSource customerDB;

  public void myMethod(String myString) {
      …
      Connection conn = customerDB.getConnection();
      …
  }
}
```

Setter Injection

```
@Stateless public class MySessionBean {

   private DataSource customerDB;

   @Resource
   public void setCustomerDB(DataSource customerDB){
     this.customerDB = customerDB;
   }

   public void myMethod(String myString) {
     …
      Connection conn = customerDB.getConnection();
      …
     }
}
```

Dynamic Lookup

```
@Inject
 private void setSessionContext(SessionContext ctx){
   this.ctx = ctx;
 }
…
 myShoppingCart = (ShoppingCart)ctx.lookup
   (“shoppingCart”);
…
```

Entity Beans: Goals

- Simplify programming model
  - POJO/JavaBeans like other EJB 3.0 beans
- Support for light-weight domain modeling
  - Inheritance and polymorphism
  - Extension of query language
  - Focus on O/R mapping
- Make instances usable outside the container
  - Remove need for DTOs and similar anti-patterns
  - Facilitate testability
- Complete query capabilities

EJB QL

- Bulk update and delete operations
- Projection list (SELECT clause)
- Explicit joins (inner and outer)
- Group by, Having
- Subqueries (correlated and not)
- Additional SQL functions
  - UPPER, LOWER, TRIM, …
- Dynamic queries

Support for native SQL queries in addition to EJB QL enhancements

JAVAPOLIS

Entity Beans

- Concrete classes (no longer abstract)
- Support use of new()
- No required bean interfaces
- getter/setter methods
  - Can contain logic, e.g., for validation, transformation, etc.
- Direct persistent instance variable access in bean
  - No exposure of instance variables outside bean class
- Collection interfaces for relationships
- Usable outside the container
- No required callback interfaces

EntityManager

- Serves as untyped “home”
- Similar in functionality to Hibernate Session, JDO PersistenceManager, etc.
- Methods for lifecycle operations
  - create, remove, merge, flush, etc.
- Factory for Query objects

Lifecycle Model

- new()
  - Instance is not yet managed
- create()
  - Instance becomes managed
  - Instance becomes persistent (in database) upon transaction commit
- remove()
  - Deletion of persistent instance upon transaction commit
- Detached instances
  - Instances exist outside of original transaction context in which created / retrieved
    - E.g., serialized to other tiers
- merge()
  - Merge detached instance state back into persistence context

Entity Bean Example

```
@Entity public class Customer {
  private Long id;
  private String name;
  private Address address;
  private HashSet orders = new HashSet();

  @Id(generate=AUTO)
  public Long getID() {
    return id;
  }

  private void setID(Long id) {
    this.id = id;
  }
```

Example (cont.)

```
  public String getName() {
    return name;
  }

  public void setName(String name) {
    this.name = name;
  }

  @OneToMany(Cascade=ALL)
  public Set<Order> getOrders() {
    return orders;
  }

  public void setOrders(Set<Order> orders) {
    this.orders = orders;
  }
}
```

Client View

```
@Stateless public class OrderEntryBean {

  @Inject private EntityManager em;

  public void enterOrder(int cId, Order newOrder) {
    Customer cust = em.find(Customer.class, cID);
    cust.getOrders.add(newOrder);
    newOrder.setCustomer(cust);
  }

// other business methods
}
```

Object/Relational Mapping Approach

What do we mean by this?

Persistence design spectrum:

Database centric approach

Object centric approach

O/R Mapping Approach

- Managed Java classes
  - provide an automated facility for developer working with domain object model mapped to relational database
- Developer is aware of mapping between database and domain object model
  - At query language level
  - At object/data transfer and faulting level
- Hooks are provided over mapping between relational database and Java object instances
  - Semantically explicit approach
  - Programmer is in control
  - Classes are transparent; mapping semantics are not

Application Developer's Strategy

- Definition of object model
  - In conjunction with database model to facilitate expected data access
- Construction and manipulation of "persistence context"
  - Set of managed persistent instances used by application within a particular scope
  - Application developer knows the needed shape of this
- Use queries to bring in data sets of interest
  - Most efficient way to access relational data
  - EJB QL significantly enhanced
  - NativeQuery methods also available for direct SQL queries
- Use metadata to control faulting patterns
  - Explicit control of data prefetch

Points of Control

- Over lifecycle
  - Cascade capabilities (CREATE, REMOVE, MERGE, …)
- Scope of persistence context
- Queries
- Fetch / faulting behavior
  - FETCH JOINS
  - FetchType metadata
    - EAGER
    - LAZY
- Optimistic locking support
  - @Version; @Timestamp
- Isolation
  - FlushType, etc.

Example: Mapping Relationships

```
@Entity public class Employee {
  private Address address;

  @ManyToOne
   public Address getAddress() {
     return address;
   }

   public void setAddress(Address address) {
     this.address = address;
   }
}


@Entity public class Address {
   …
}
```

Use of Defaulting

- Entity Employee is mapped to table EMPLOYEE
- Entity Address is mapped to table ADDRESS

- Table EMPLOYEE contains foreign key to table ADDRESS
- Foreign key column has same name and type as primary key of ADDRESS

Mapping Classes to Tables

- Use Java metadata to specify mapping
- Support for usual strategies
  - Table per class
  - Table per class hierarchy
  - Joined subclass
- Default type mappings available
- Custom type mappings under consideration

Example: Inheritance Mapping

```
@Entity
@Table(name="CUST")
@Inheritance(strategy=SINGLE_TABLE,
             discriminatorType=STRING,
             discriminatorValue="CUST")
public class Customer {…}

@Entity
@Inheritance(discriminatorValue="PCUST")
public class PreferredCustomer extends Customer {
   …
}
```

Same Example, Using Defaults

```
@Entity
public class Customer {…}

@Entity
public class PreferredCustomer extends Customer {
…
}
```

Persistence Update: EJB 3 / JDO 2 Resolution

- Evolution of single persistence API based on EJB 3.0 draft
- Single persistence model, O/R mapping facility will be usable in J2EE and non-J2EE environments
- Persistence API in separate spec document, produced by JSR 220 (EJB 3.0) Expert Group
- Expansion of JSR-220 Expert Group
  - Inclusion of members from JDO Expert Group
  - Leading JDO vendors + individual members
  - Expert group is now ~35 members

Expansion of Scope: "Common API"

What It is:

- Single persistence model for common use in J2SE as well as J2EE
- Additional APIs to enable use outside of J2EE containers
  - APIs for acquiring EntityManager, managing transactions, configuration , etc

What It Isn't:

- It isn't "part" of J2SE (not in JDK)
- It doesn't shift primary focus from J2EE

JAVAPOLIS

Convergence Effort

What It Is:

- Merger of expertise
  - Leverage expertise of members from JDO expert community (as well as Hibernate, TopLink, EJB vendors,…)
  - Draw upon best of breed ideas from all these sources

What It Isn't:

- It isn't a merger of APIs

It is a merger of good ideas + experience

Current Focus

- Focus on O/R mapping
  - EJB 3.0 Early Draft as basis for expanded effort
  - Augment / improve / fill in holes
  - Fresh perspective of new members very helpful here
- Open Issues we're addressing
  - O/R mapping spec, including XML
    - XML as alternative / overriding mechanism
  - SQL queries
  - APIs to enable use outside the container

EJB QL Enhancements

- Bulk update and delete operations
- Projection list (SELECT clause)
- Explicit joins (inner and outer)
- Group by, Having
- Subqueries (correlated and not)
- Additional SQL functions
- Dynamic queries

Examples: Bulk Update / Delete

```
DELETE
FROM Customer c
WHERE c.status = 'inactive'

UPDATE Customer c
SET c.status = 'outstanding'
WHERE c.balance < 1000
```

Examples: Subqueries

```
SELECT preferredCustomer
FROM Customer preferredCustomer
WHERE preferredCustomer.balance < (
  SELECT avg(c.balance) from Customer c)
```

```
SELECT DISTINCT emp
FROM Employee emp
WHERE EXISTS (
  SELECT spouseEmp
  FROM Employee spouseEmp
  WHERE spouseEmp = emp.spouse)
```

Examples: Joins

```
SELECT i.category
FROM Item i JOIN i.bids b
WHERE b.amount > :amount
```

```
SELECT i
FROM Item i LEFT JOIN FETCH i.bids
WHERE i.category = 'paintings'
```

Examples: Projection

```
SELECT c.id, c.status
FROM Customer c JOIN c.orders o
WHERE o.count > 100
```

```
SELECT new CustomerDetails(c.id, c.status, o.count)
FROM Customer c JOIN c.orders o
WHERE o.count > :ordercount
```

SQL Queries

- Allow direct SQL over actual database schema
  - Very useful for some applications
  - Database portability not important for some applications
  - Escape mechanism for very complex queries not handled by EJB QL
- Allow SQL query results to be mapped into entity beans and/or instances of other Java classes

Summary: EJB 2.1 Component View

- Contractual view of components
- Container specifies contracts
  - Interfaces beans must adhere to
  - Deployment descriptor contract
- Beans written to contracts
  - Whether they need them or not
- Very static viewpoint; negative consequences in terms of application complexity

Summary: EJB 3.0 Shift in Viewpoint

- Container is a service injection facility
  - Injection points specified through metadata
  - Container calls bean if/when bean specifies its need
- Inversion of contractual view
  - Bean doesn't implement a predefined set of APIs
  - Instead, bean specifies its expectations on the container
- More extensible architecture

Current Status

- Early Draft 1 released in summer
- Targeting Early Draft 2 in ~1 month
  - Additions:
    - **Callbacks**
    - **Interceptors**
    - **EntityManager infilling**
    - **Improvements to O/R mapping spec**
    - **SQL queries**
- Plan an Early Draft 3 as well
  - Use Early Drafts to get community feedback

Conclusion

- POJO-centric view of all enterprise beans
- Simplification of environment access
- Simplification of entity beans
  - Clear O/R mapping orientation
  - Improvements of query language capabilities
- Metadata is major enabling technology
- Major collaboration effort by Expert Group