Java EE 5 – EJB 3, JSF

Gerald Loeffler

Enterprise Software Architect, ShipServ Ltd

Contents

- Where we are coming from
- EJB 3: simplified API
- EJB 3: Java Persistence API
- Unit testing with EJB 3
- JSF meets EJB 3
- Where are we now?

References

- JSR 220: Enterprise JavaBeans, Version 3.0
  - EJB 3.0 Simplified API, Proposed Final Draft, 18 Dec 2005, EJB 3.0 Expert Group, Sun Microsystems
  - Java Persistence API, Proposed Final Draft, 19 Dec 2005, EJB 3.0 Expert Group, Sun Microsystems
- JBoss documentation
  - JBoss EJB 3.0 Documentation
  - Embeddable EJB 3.0
  - JBoss Seam Documentation
- JavaServer Faces Specification, Version 1.2 Proposed Final Draft, Ed Burns, Roger Kitain (ed.), Sun Microsystems

Software versions

- JBoss 4.0.3 SP1, 24 Oct 2005
  - Includes EJB 3.0 container "on top of" J2EE 1.4
    - Includes snapshot of Hibernate for persistence
- Hibernate as an EJB 3 persistence provider
  - Hibernate Core 3.1.1, 18 Jan 2006
  - Hibernate Annotations 3.1 beta 8, 20 Jan 2006
  - Hibernate Entity Manager 3.1 beta 6, 20 Jan 2006
- Glassfish Milestone 4, 20 Dec 2005
  - Nascent reference implementation for Java EE 5
- Kodo 4.0.0 persistence provider early access
  - SolarMetric now owned by BEA
- JOnAS EJB 3 early preview, 19 Dec 2005

Where we are coming from

Where we are coming from

- J2EE development judged as too complex
  - Over-use of XML-based configuration
- Light-weight Java ideas firmly established
- Entity beans considered obsolete
- Unit-testing of business objects commonplace
- Dependency injection with stateless objects well understood and widely used
  - Service object -> DAO
- AOP with "defensive" advices well-established
  - Logging, performance measurement, TX management
- Plethora of web UI frameworks
  - Many popular ones (e.g. Tapestry) shunning JSPs

EJB 3: simplified API

Themes

- Simplifying the developer's task
- Metadata annotations in addition to XML
- Configuration by exception
- Session beans: no required home interface
- Entity beans: light-weight ORM
- Interceptors for session beans and MDBs
- Architectural properties
  - of session beans and MDBs remain essentially unchanged
  - of EJB 3 entities are very similar to Hibernate entities

Session and message-driven beans

- Focus on (annotated) enterprise bean class
- Deployment descriptors available
  - To override annotations
  - May be sparse
- Support interception of invocation of
  - Business methods
  - Lifecycle callbacks
- Can be target of dependency injection
- Transaction management:
  - Default is container-managed
  - `@TransactionManagement` on bean class to specify bean-managed
  - `@TransactionAttribute` on bean or business methods

FamilyService1.java

```java
// plain interface
public interface FamilyService1 {
  Object createNewFamily(String spec);
}
```

FamilyServiceImpl1.java

```
// stateless, local, CMT, two interceptors
// name defaults to FamilyServiceImpl1
@Stateless
@Interceptors( { LogInterceptor.class })
public class FamilyServiceImpl1 implements
     FamilyService1 {
   public Object createNewFamily(String spec) {
     Object family = null;
     // ...
    return family;
   }
   @AroundInvoke
   public Object spy(InvocationContext ictx)
       throws Exception {
     // tell the neighbours...
     return ictx.proceed();
   }
}
```

LogInterceptor.java

```
public class LogInterceptor {
  @AroundInvoke
  public Object log(InvocationContext ictx)
      throws Exception {
    try {
      // log args
      Object ret = ictx.proceed();
      // log return value
      return ret;
    } catch (Exception e) {
      // log exception
      throw e;
    }
  }
}
```

Lifecycle and lifecycle callbacks

1. Instantiation of bean instance
2. Dependency injection
3. `@PostConstruct`
   - Unspecified transaction and security context
4. Invocations of business methods
5. Invocation of business method with @Remove
   - Stateful session beans only
6. `@PreDestroy`
   - Unspecified transaction and security context
7. Destruction of bean instance

Lifecycle and lifecycle callbacks

- For stateful session beans only: between `@PostConstruct` and `@PreDestroy` possibly arbitrary pairs of
  - `@PrePassivate`
  - Passivation (serialization) to external storage
  - Activation (deserialization) from external storage
  - `@PostActivate`

Session beans

- Business interface is plain Java interface
  - No required component interface (`EJB(Local)Object`)
- Default is local business interface
  - Use `@Local` and `@Remote`
    - If more than one interface implemented
    - If a remote interface is required
    - May appear on business interface or bean class
  - `Serializable`, `Externalizable` can't be business interfaces
- Declare arbitrary exceptions on business methods
  - Don't use `RemoteException`
  - Remote client sees `EJBException`

Session beans

- No need for home interface
  - Lookup (incl. injection) returns EJB instance

FamilyService2.java

```
@Remote
public interface FamilyService2 {
   void initFamily(String mothersName);
   void addChild(String name);
   void enough() throws NotYetEnough;
}
```

WorkService2.java

```java
// plain interface
public interface WorkService2 {
  double earnIncome();
}
```

FamilyServiceImpl2.java

```java
// stateful, remote and local, CMT
@Stateful(name = "StatefulFamilyService")
@Local( { WorkService2.class })
public class FamilyServiceImpl2 implements
    FamilyService2, WorkService2 {
  public void initFamily(String mothersName) {}
  public void addChild(String name) {}
  @Remove(retainIfException = true)
  public void enough() throws NotYetEnough {}
  @TransactionAttribute(
    TransactionAttributeType.REQUIRES_NEW)
  public double earnIncome() {
    return 0.0;
  }
}
```

Stateless session beans

- `@Stateless` annotation
  - No need to implement `SessionBean`
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`
- `@WebService`, `@WebMethod` for definition of web services (JSR 181)

Stateful session beans

- `@Stateful` annotation
  - No need to implement `SessionBean` or `Serializable`
- Implementation of `SessionSynchronization` supported
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`
  - `@PostActivate`, `@PrePassivate`
- Lookup returns new instance
  - Typically needs initialization via business method(s)!
- `@Remove` annotates a "normal" business method
  - Client initiates removal by calling this method
  - Removal through container after completion

Message-driven beans

- `@MessageDriven` annotation
  - No need to implement `MessageDrivenBean`
- Business interface is defined by messaging type
  - For JMS: `javax.jms.MessageListener`
- Lifecycle callbacks
  - `@PostConstruct`, `@PreDestroy`

AsynchNappyChanger.java

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName="destinationType",
    propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(
    propertyName="destination",
    propertyValue="jms/NappyQ")
})
public class AsynchNappyChanger implements
    MessageListener {
  public void onMessage(Message msg) {
    // ...
  }
}
```

Interceptor methods

- Specialised AOP facility
  - Only around advice
  - Only on session beans and MDBs
    - Only business methods and lifecycle callbacks
- Method with `@AroundInvoke`
  - Only one per bean class
  - `public Object *(InvocationContext) throws Exception`
  - `InvocationContext` passed around as data holder
  - `InvocationContext.proceed()` to proceed
- Become "part of" method invocation
  - Share transaction and security context
  - Can throw same exceptions as "their" method
  - Can invoke JNDI, JDBC, JMS, EJBs, Entity Manager

FamilyServiceImpl1.java

```
// stateless, local, CMT, two interceptors
// name defaults to FamilyServiceImpl1
@Stateless
@Interceptors( { LogInterceptor.class })
public class FamilyServiceImpl1 implements
     FamilyService1 {
   public Object createNewFamily(String spec) {
     Object family = null;
     // ...
    return family;
   }
   @AroundInvoke
   public Object spy(InvocationContext ictx)
       throws Exception {
     // tell the neighbours...
     return ictx.proceed();
   }
}
```

Interceptor classes

- Holds interceptor method
  - Same rules as for those
  - Only one `@AroundInvoke` per interceptor class
- Stateless, associated with enterprise bean
  - `InvocationContext` passed around as data holder
- Public no-arg constructor
- Definition and assocation with beans is static
  - Default interceptors
    - Apply to all session beans and MDBs in ejb-jar
    - Defined in deployment descriptor
  - Denoted on bean using `@Interceptors`
    - Definition order is invocation order
- Can be target of dependency injection

LogInterceptor.java

```
public class LogInterceptor {
  @AroundInvoke
  public Object log(InvocationContext ictx)
      throws Exception {
    try {
      // log args
      Object ret = ictx.proceed();
      // log return value
      return ret;
    } catch (Exception e) {
      // log exception
      throw e;
    }
  }
}
```

Dependency injection

- Injection of
  - `EJBContext` (`SessionContext`) (first)
  - `DataSource`
  - `UserTransaction`
  - `EntityManager`
  - Anything (?) that can be looked up via JNDI in `java:comp/env`
- Injection into fields or with setters
  - `@EJB`, `@Resource`
  - Missing information inferred from field or setter
    - Resource type
      - From field/property type
    - Resource name
      - From field/property name

FamilyService3.java

```
public interface FamilyService3 {
  Object generateBoyfriend(String spec);
}
```

FamilyServiceImpl3.java

```
// declarative security, EJB ref, resource ref
@Stateless
@DeclareRoles( { "Adolescent", "MidlifeCrisis" })
@RunAs("Adolescent")
public class FamilyServiceImpl3 implements
    FamilyService3 {
  @EJB
  private FamilyService1 consequences;
  @Resource(name = "jms/NappyQ")
  private QueueConnectionFactory qcf;
  @RolesAllowed( { "Adolescent“,"MidlifeCrisis" })
  public Object generateBoyfriend(String spec) {
    Object boy = null;
    // ...
    Object f = consequences.createNewFamily(spec);
    // ...
    return boy;
  }
}
```

What we didn't talk about

- Enterprise bean context and lookup
  - Explicit lookups:
    - New `EJBContext.lookup()` method
    - JNDI lookups
- New deployment descriptor

EJB 3: Java Persistence API

What it's about

- Object/relational mapping
- Focus is on (annotated) Java domain classes
  - Light-weight, local
- Query language is new version of EJB QL
- Can be used outside of EJB container
  - Web container
  - Java SE
- Very similar to Hibernate in "look&feel"
- Annotations and DDs supported
- Core concepts:
  - Persistence provider, Entity, EntityManager, persistence context, persistence.xml, orm.xml

First a “realistic” example…

Family.java

```
// table: Family
// col: id primary key
// sequence: FAMILY_SEQU
@Entity
@SequenceGenerator(name="FamilySequ“,
  sequenceName="FAMILY_SEQU“
public class Family {
  @Id
  @GeneratedValue(strategy=GenerationType.
    SEQUENCE, generator="FamilySequ")
  private Integer id;
  // inverse side of bidirectional 1-to-many
  @OneToMany(mappedBy="family", cascade=ALL)
  private Set<FamilyMember> members;
}
```

FamilyMember.java

```
// table: FAMILY_MEMBER
// col: id primary key
// col: family_id references FAMILY.id
@Entity
@Table(name = "FAMILY_MEMBER")
@Inheritance(strategy = InheritanceType.JOINED)
public class FamilyMember {
    @Id
    private Integer id;
    // owning side of birectional many-to-1
    @ManyToOne(cascade = { CascadeType.MERGE,
        CascadeType.PERSIST, CascadeType.REFRESH
    })
    private Family family;
}
```

Parent.java

```
// table: Parent
// col: id primary key refs FAMILY_MEMBER.id
@Entity
public class Parent extends FamilyMember {
  // ...
}
```

Child.java

```
// table: Child
// col: id primary key references FAMILY_MEMBER.id
// col: reatreat_id unique refs PRIV_CHILD_ROOM.id
// join table: Child_Toy
// col: Child_id references Child.id
// col: toys_name references Toy.name
@Entity
@NamedQuery(name="findAChild",
  query="select c from Child c where...")
public class Child extends FamilyMember {
  // owning side of bidirectional 1-to-1
  @OneToOne(cascade = { MERGE, PERSIST, REFRESH })
  private PrivateChildRoom retreat;
  // (owning side of) unidirectional many-to-many
  @ManyToMany(cascade = { MERGE, PERSIST, REFRESH })
  private Set<Toy> toys = new LinkedHashSet<Toy>();
}
```

PrivateChildRoom.java

```
// table: PRIV_CHILD_ROOM
// col: id primary key
@Entity
@Table(name = "PRIV_CHILD_ROOM")
public class PrivateChildRoom {
  @Id
  private Integer id;
  @OneToOne(mappedBy = "retreat", cascade = {
      CascadeType.MERGE, CascadeType.PERSIST,
      CascadeType.REFRESH })
  // inverse side of bidirectional 1-to-1
  private Child occupiedBy;
}
```

Toy.java

```
// table: Toy
// col: name varchar2(255) primary key
public class Toy {
   @Id
   private String name;
}
```

…now the theory

Entities

- `@Entity` annotation
- Public/protected no-arg constructor
- Not inner class
- Not final, no methods final
- Can be `Serializable`
  - E.g. if used as a Data Transfer Object
- Can be abstract
- Can be part of inheritance tree

Persistent state and access to it

- Access to fields
  - By persistence provider
    - Property-based: through JavaBeans property accessors
      - Annotate getter (!)
      - Accessors must be public or protected
      - Exclusion from persistent state: `@Transient`
    - Field-based: direct
      - Annotate fields
      - Exclusion from persistent state: `transient` or `@Transient`
  - By clients of entity: only through accessors
    - Fields must not be public
    - (And should really be private)
- Personal recommendation:
  - Use field-based access

Persistent state

- Types of fields/properties
  - Primitives, primitive wrappers, `String`
  - `BigInteger`, `BigDecimal`
  - `Date`, `Calendar`
  - `java.sql`-types: `Date`, `Time`, `Timestamp`
  - `Char[]`, `Character[]`
  - `Byte[]`, `Byte[]`
  - `Collection` (sub-) interfaces
    - `Collection`, `Set`, `List`, `Map`
  - Enums
  - Entities
  - Embeddables

Entity identity: primary keys

- Every identity has an id, corresponds to primary key in the database
  - Is immutable after `persist()`
- Simple primary key
  - Single field mapped to single column: `@Id`
- Composite primary key
  - Primary key class with multiple fields mapped to multiple columns
    - Either embedded in entity: `@EmbeddedId`
    - Or referenced from entity and primary key fields duplicated in entity: `@IdClass`
    - Public no-arg constructor, `Serializable`, `equals()` and `hashCode()` based on primary key columns

Family.java

```
// table: Family
// col: id primary key
// sequence: FAMILY_SEQU
@Entity
@SequenceGenerator(name="FamilySequ“,
  sequenceName="FAMILY_SEQU“
public class Family {
  @Id
  @GeneratedValue(strategy=GenerationType.
    SEQUENCE, generator="FamilySequ")
  private Integer id;
  // inverse side of bidirectional 1-to-many
  @OneToMany(mappedBy="family", cascade=ALL)
  private Set<FamilyMember> members;
}
```

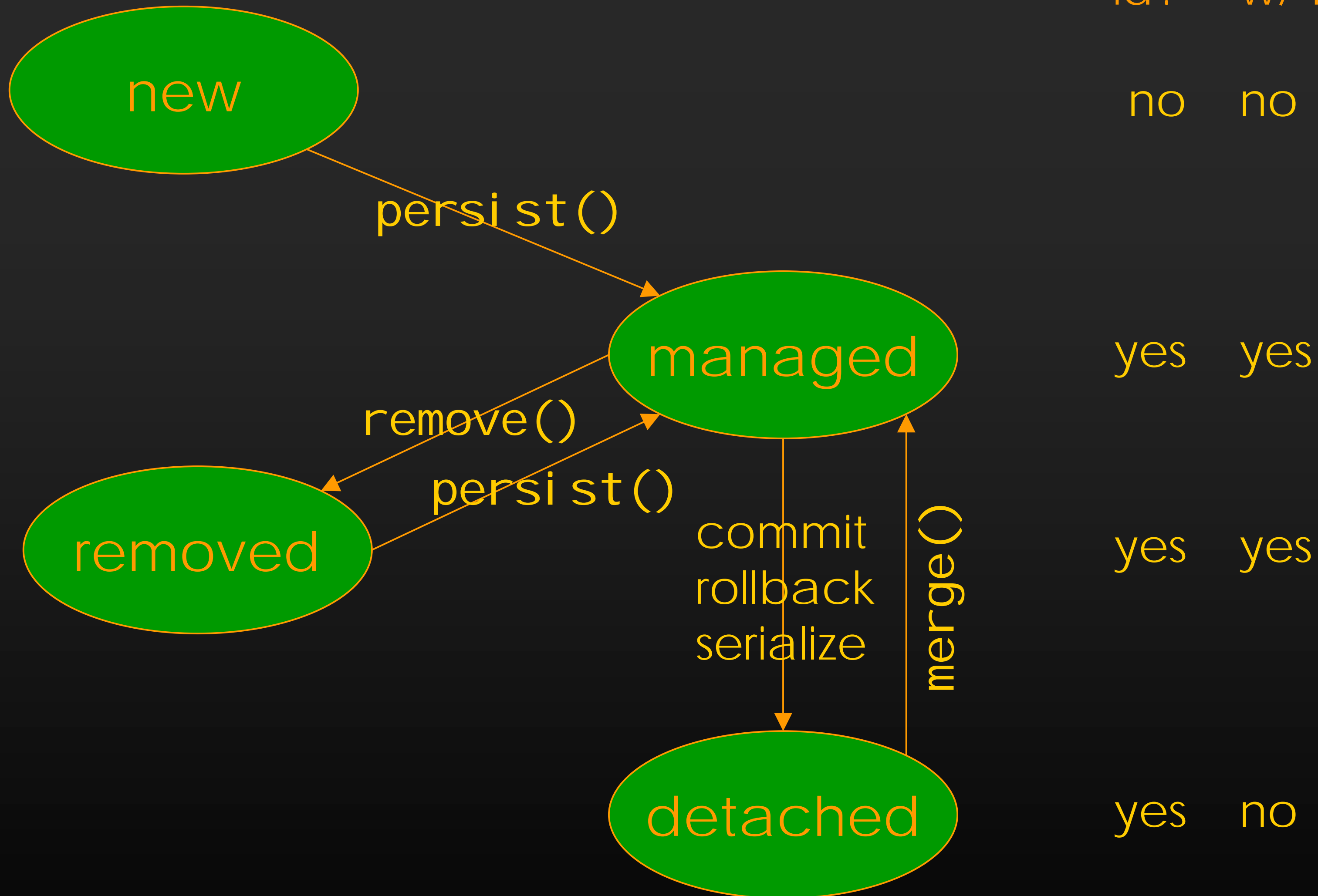
Persistence context

- Accessed through `EntityManager`
- Comprises a set of entity instances where at most one instance exists for each persistent entity identity (= primary key value)
  - Is exactly what you expect: "one row of a table corresponds to one object"
- Persistence context lifetime
  - Defined when EntityManager is created (injected)
  - Transaction-scoped persistence context
    - Default and most natural
  - Extended persistence context
    - PC spans more than one transaction

EntityManager

- Works on exactly one persistence context
  - Defines the API to interact with persistence context
- Important operations:
  - `void persist(Object entityInstance)`
  - `void remove(Object entityInstance)`
  - `T merge (T entityInstance)`
    - Detached object support
  - `T find(Class<T> entity, Object id)`
  - `void refresh(Object entityInstance)`
  - `Query createNamedQuery(String queryName)`
    - EJB QL query string defined with `@NamedQuery`
    - SQL query string defined with `@NamedNativeQuery`

Entity states



Cascading entity state changes

- Each entity-entity relationship may be annotated with one or more `CascadeTypes`:
  - Default is to not cascade!
  - `CascadeType`: `PERSIST`, `REMOVE`, `MERGE`, `REFRESH`, `ALL` to propagate the corresponding state changes (method calls) on the entity instance w/ the annotated reference to the referenced entity instance(s).
  - No delete-orphan!
  - REMOVE not portable for ManyToOne or ManyToMany
- A note on `CascadeType.MERGE`
  - As `merge()` can result in a copy of the instance to be returned, cascading `merge()` requires that the reference be re-set to that copy. This is done by the persistence provider.
  - Can thus result in a completely new object graph!

Synchronisation with database

- Flush: entity instance state written to database
  - Only effects entities currently associated with PC
  - Simplest case: automatic flush at commit
  - Explicitly through `EntityManager.flush()`
  - Typically (Hibernate) before query execution
    - Ensure that entity state always obeys all constraints
- Refresh: database to entity instance state
  - Only explicitly for each instance through `EntityManager.refresh()`
  - Overwrites un-flushed changes to entity instance

...to be continued

Gerald Loeffler

Enterprise Software Architect, ShipServ Ltd