Java™ Persistence API: Best Practices and Tips

Rima Patel Sriganesh

Marina Vatkina

Mitesh Meswani

Sun Microsystems, Inc.

Session TS-4902

Goal of This Talk

Present the Best Practices, Gotchas, and Tips to help you develop Java™ Persistence API applications!

Agenda

- **Persistence Context**
- **Entities**
- **Concurrency**
- **Query Tips**
- **Resources and Q&A**

Agenda

- **Persistence Context**
  - **Persistence Context Types**
  - **Threading Model Mismatch and Injection**
  - **Persistence Context and Caching**
- **Entities**
- **Concurrency**
- **Query Tips**
- **Resources and Q&A**

Container vis-a-vis Application Managed Entity Managers

Guidelines

- Container-managed entity manager
  - Container propagates persistence context
  - Always look up or inject entity manager in a managed environment instead of passing entity manager proxy as a variable
- Application-managed entity manager
  - The only option outside of a Java Platform, Enterprise Edition (Java EE Platform) 5 container
  - Persistence.createEntityManagerFactory() is the only portable way to create EMF in a non-Java EE Platform 5 web container
  - Do not forget to call close() on EntityManager

Extended vis-a-vis Transactional Scoped Persistence Context

Guidelines

- Transaction-scoped persistence context
  - Choose it when your business transaction is stateless (spans a single request from the user)
  - Ideal place of injection/creation—request's entry and exit points
- Extended-scoped persistence context
  - Choose it when your business transaction spans multiple requests from the user
  - Ideal place of injection/creation—business transaction's entry and exit points (for example—a stateful session bean)
- Beware of "propagation" implications of mixing and matching container-managed transaction-scoped and extended-scoped persistence contexts

Injection and Threading Model Mismatch

- Field injection is only supported for instance variables
- Threading model of Java Persistence API Components
  - EntityManagerFactory is thread-safe
  - EntityManager is not thread-safe
- Threading model of Java EE Platform components
  - Servlets are multi-threaded
  - Session and application scoped JavaServer™ Faces technology managed beans are multi-threaded
  - Request scoped JavaServer Faces technology managed beans are single-threaded
  - Enterprise JavaBeans™ (EJB™) are single-threaded

Injecting EntityManager in Java EE Platform Components

Guidelines

- Never inject EntityManager into your Servlet or JavaServer Faces application/session scoped managed beans
- Instead, within Servlet or JavaServer Faces application/session scoped managed bean methods
  - Lookup EntityManager using Java Naming and Directory Interface™ (J.N.D.I.)
  - OR create EntityManager from EntityManagerFactory
- No caution is needed when injecting Java Persistence API components within Enterprise Beans
  - Consider refactoring your applications to use EJB technology as a facade to entities

Persistence Context and Caching

Consider this example

```
@Stateless
public class EmployeeDAO {
    @PersistenceContext
    EntityManager em;

    public Employee findById(Integer employeeId) {
// Load an instance of Employee in the persistence context
// cache
        Employee employee1 = em.find(Employee.class,
        employeeId);
        ...
// Imagine that someone changes the last name of this Employee in
// the meantime inside or outside
        ...
// Now get an instance of the same Employee again
        Employee employee2 = em.find(Employee.class,
        employeeId);
        ...
```

Do you think employee2.getLastname() will return you an updated last name?

Persistence Context and Caching

Consider this example(Continued)

Also, do you think that hitting the database again through dynamic query will get you an updated last name for our employee?

```
        ...

    // And in the same method, try retrieving Employee by issuing
    // a dynamic query
        Employee employee3 = (Employee)em.createQuery
        ("SELECT e FROM Employee e WHERE e.ID = :ID")
        .setParameter("ID", employeeId)
        .getSingleResult();

        ...
    }

    ...
}
```

Lessons Learned

Persistence context as a first-level cache

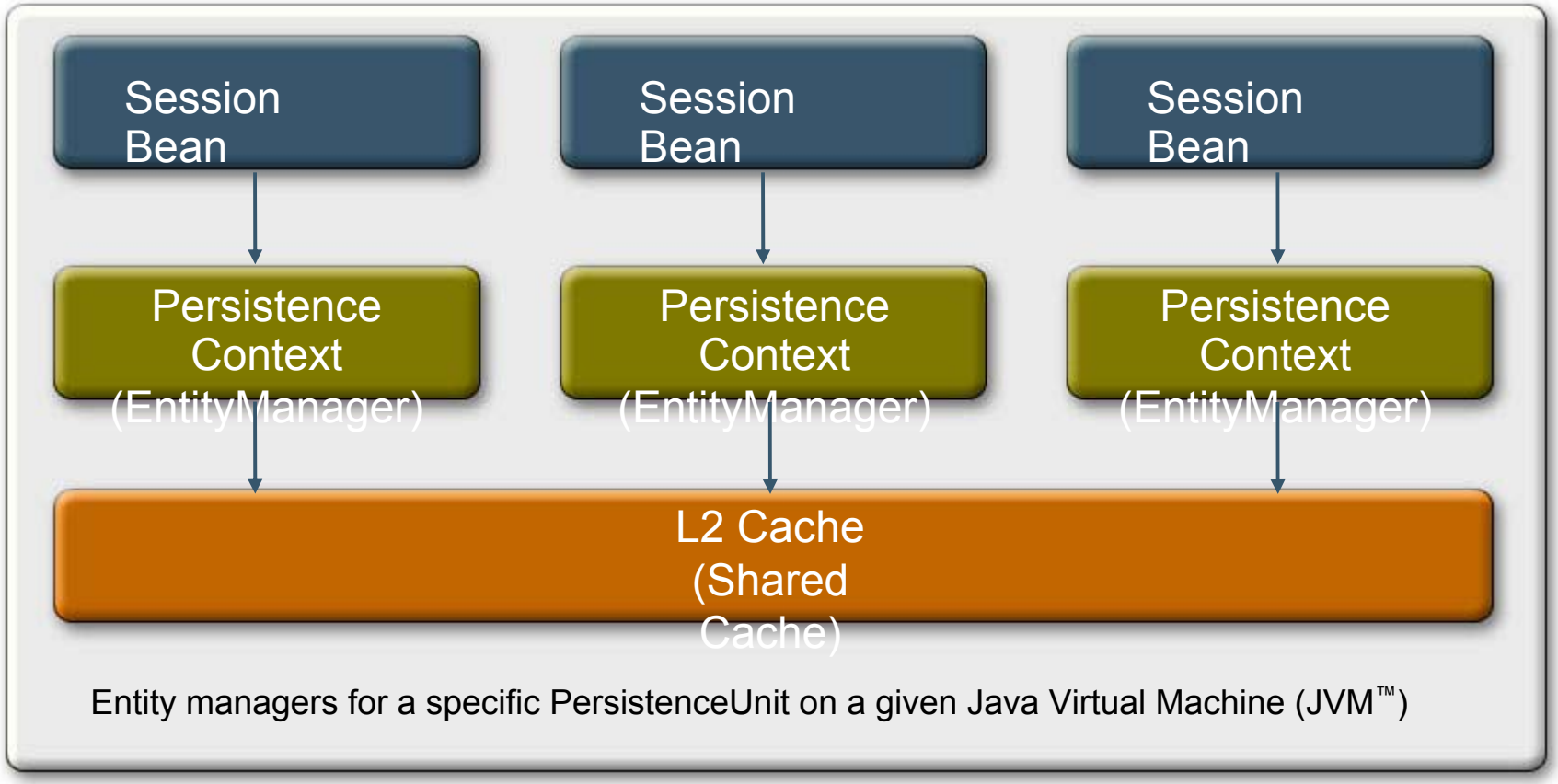
- The entities managed by persistence context
  - Are not refreshed until
    - EntityManager.refresh() is explicitly invoked
  - Are not synchronized with the database until
    - EntityManager.flush() is invoked implicitly or explicitly OR
    - The underlying transaction commits
  - Remain managed until
    - Extended-scoped: EntityManager.clear() is invoked
    - Transaction-scoped: the transaction commits or EntityManager.clear() is invoked

Second-level Cache

- An application might want to share entity state across various persistence contexts
  - This is the domain of second-level (L2) cache
  - If caching is enabled, entities not found in persistence context, will be loaded from L2 cache, if found
- Java Persistence API does not specify support of a second level cache
  - However, most of the persistence providers provide built-in or integrated support for second-level cache(s)
  - Basic support for second level cache in Project GlassFish™—TopLink Essentials is turned on by default

L1 and L2

Putting it all together



The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ Platform.
Source:http://weblogs.java.net/blog/guruwons/archive/2006/09/understanding_t.html

Agenda

- Persistence Context
- Entities
  - Access Types
  - Generated Primary Keys
  - Inheritance Hierarchy
  - Relationships
- Concurrency
- Query Tips
- Resources and Q&A

Access Types

- Defined by annotations placement or XML overrides
- Field-based
  - Separates client view from provider access
  - Validation/conversion logic in getters/setters for client only
- Property-based
  - CMP migration
  - Validation/conversion logic in getter/setter for the provider and the client

```
@Entity public class PartTimeEmployee extends Employee {
        public void setRate(int newrate) {
            if (rate > newrate)
                logger.warning("Lowering rate to " + newrate);
            rate = newrate;
        }
```

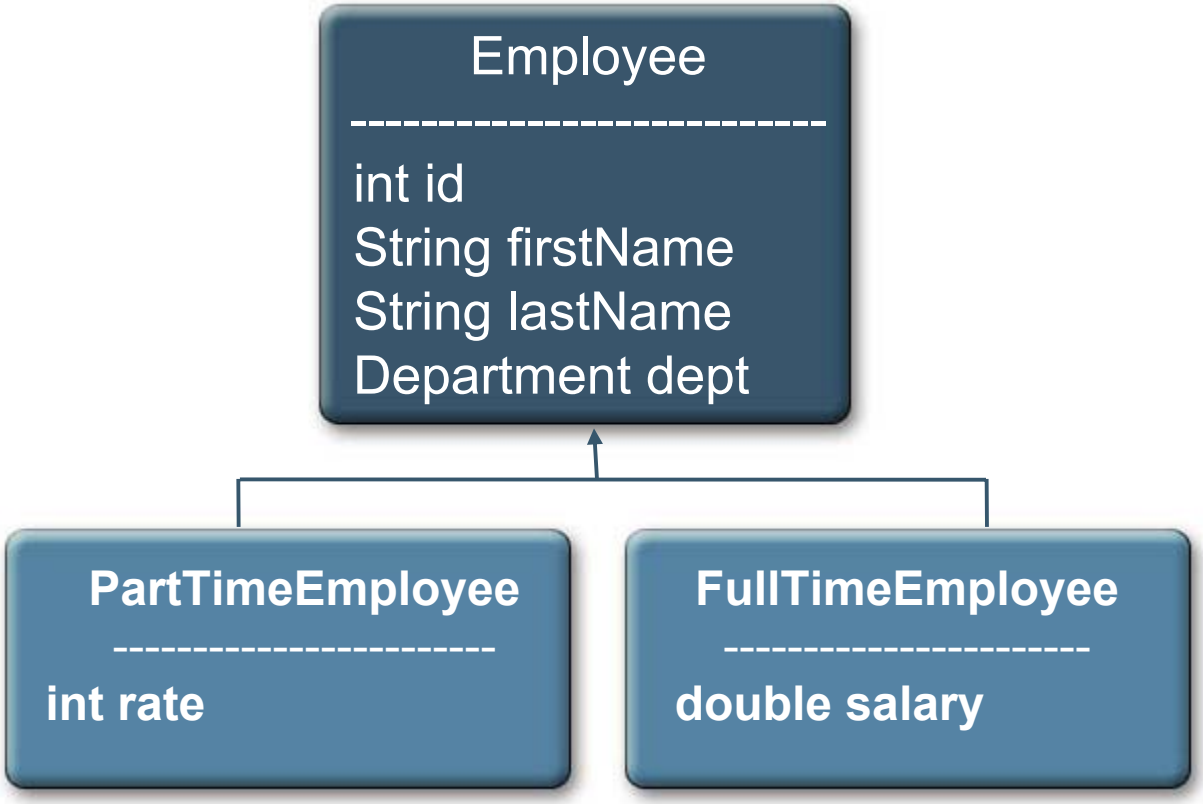
Generated Primary Keys

- Types of generators
  - TABLE—portable across databases and providers
  - SEQUENCE
  - IDENTITY
  - AUTO
- Sequence may not be portable across databases
- For portability across providers, specify generator to give mapping details

```
@Id
@GenerateValue(strategy=TABLE, generator="myGenerator")
long id;
```

Mapping of Inheritance Hierarchies

Domain model



Mapping of Inheritance Hierarchies

Single table per class mapping strategy

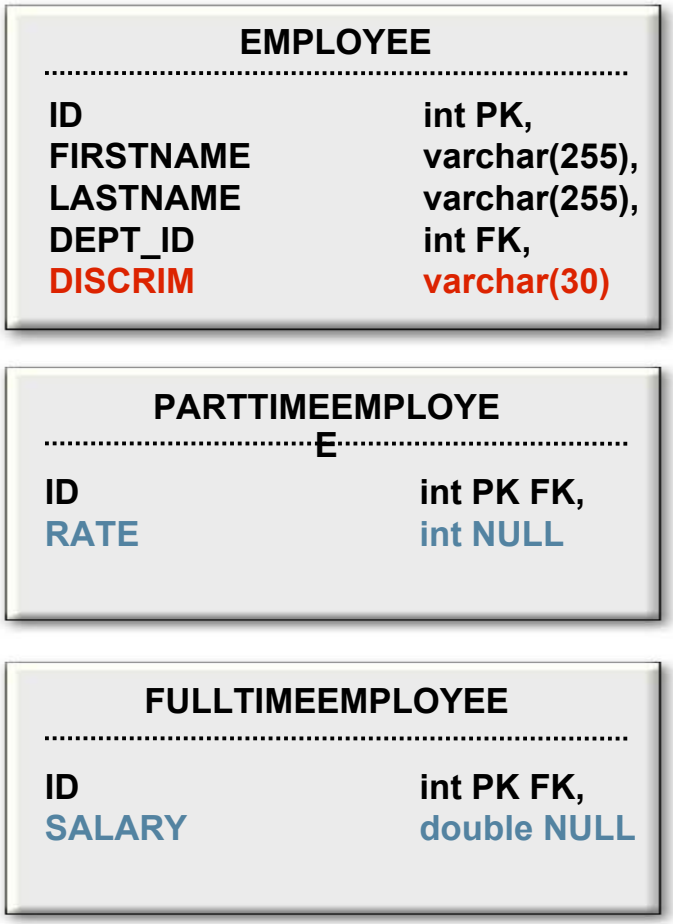
- Benefits
  - Simple
  - No joins required
- Drawbacks
  - Not normalized
  - Requires columns corresponding to subclasses' state be nullable
  - Table can have too many columns

EMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL,
SALARY	double NULL,
DISCRIM	varchar(30)

Mapping of Inheritance Hierarchies

Joined subclass mapping strategy

- Benefits
  - Normalized database
  - Database view same as domain model
  - Easy to evolve domain model
- Drawbacks
  - Poor performance in deep hierarchies
  - Poor performance for polymorphic queries and relationships
  - Might require discriminator column



Mapping of Inheritance Hierarchies

Table per concrete class strategy

- Benefits
  - No need for joins if only leaf classes are entities
- Drawback
  - Not normalized
  - Poor performance when querying non-leaf entities - unions
  - Poor support for polymorphic relationships
- Support for this strategy has not been mandated by the current specification

EMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK

PARTTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
SALARY	double NULL

Managing Relationships

Domain model

```
@Entity public class Employee {
    @Id private int id;
    private String firstName;
    private String lastName;
    @ManyToOne
    private Department dept;
       ...
}
@Entity public class Department {
    @Id private int id;
    private String name;
    @OneToMany(mappedBy = "dept")
    private Collection<Employee> emps = new ...;
          ...
 }
```

Managing Relationships

Lost relationships…

```
public int method1(...) {
    Employee e = new Employee(...);
    Department d = new Department(1, ...);
     em.getTransaction().begin();
      e.setDepartment(d);

    em.persist(e);
     em.persist(d);
    em.getTransaction().commit();

    return d.getEmployees().size();
}
```

Managing Relationships

Solved the problem!

```
public int method1(...) {
    Employee e = new Employee(...);
    Department d = new Department(1, ...);
     em.getTransaction().begin();
    e.setDepartment(d);
    d.getEmployees().add(e); //Manage relationships!
    em.persist(e);
     em.persist(d);
     em.getTransaction().commit();

    return d.getEmployees().size();
}
```

Managing Relationships

Another case—the same problem…

```
@Stateless
public class MyBean implements BeanInf {
    ...
    public void method1(...) {
        Employee e = new Employee(...);
        Department d = new Department(1, ...);
        e.setDepartment(d);
        em.persist(e);
        em.persist(d);
    }

    public int method2(...) {
        Department d = em.find(Department.class, 1);
        return d.getEmployees().size();
    }
```

Managing Relationships

Solution is still the same…

```
@Stateless
public class MyBean implements BeanInf {
    ...
    public void method1(...) {
        Employee e = new Employee(...);
        Department d = new Department(1, ...);
        e.setDepartment(d);
        d.getEmployees().add(e); //Manage relationships!
        em.persist(e);
        em.persist(d);
    }

    public int method2(...) {
        Department d = em.find(Department.class, 1);
        return d.getEmployees().size();
    }
```

Managing Relationships

Another solution

```
@Stateless
public class MyBean implements BeanInf {
    ...
    public void method1(...) {
        Employee e = new Employee(...);
        Department d = new Department(1, ...);
        e.setDepartment(d);
        em.persist(e);
        em.persist(d);
    }

    public int method2(...) {
        Department d = em.find(Department.class, 1);
        em.refresh(d);
        return d.getEmployees().size();
    }
```

Managing Relationships

And “orphan” instances…

- **What should happen to**
  - **an Employee instance if it’s removed from the collection of Employees?**
  - **a Department instance if all it’s Employees are deleted?**
- **If an instance can’t exist without being referenced, it’s an “orphan”**
- **How do I remove “orphan” instances?**
  - **No current spec support for a portable solution**
  - **Must track the changes before calling *merge* or *remove***

Navigating Relationships

Fetch type dilemma

```
Query q = em.createQuery("select d from Department d");
Collection departments = q.getResultList();
for (Department d : departments) {
    System.out.println(d.getEmployees().size());
}
```

Should I use Fetch Type EAGER or LAZY?

Navigating Relationships

Options…

- EAGER—Too many joins?

```
SELECT d.id, ..., e.id, ...
        FROM Department d left join fetch Employee e
        on e.deptid = d.id
```

  - What will it look like if you need to join more than two tables?

- LAZY—N +1 problem

```
SELECT d.id, ... FROM Department d // 1 time
SELECT e.id, ... FROM Employee e
        WHERE e.deptId = ? // N times
```

  - How many trips to the database will it be if there are more relationships?

Navigating Relationships

The solution

- Solution is in the Query string

```
Query q = em.createQuery(
        "select d from Department d
         LEFT JOIN FETCH d.employees");
```

- Using FETCH JOIN
  - Puts you in charge
  - Needs careful consideration when to use
- No similar solution for non-relationship fields or properties (BLOB, CLOB)

Navigating Relationships

And detached entities…

- **Accessing a LAZY relationship from a detached entity is not portable!**
  - **Can get an exception**
  - **Can get *null***
  - ***Can get a previously cached value—another problem***
- **Is there a portable solution?**
  - **Use JOIN FETCH for queries or fetch type EAGER**
  - **Access the collection before the entity is detached**

```
d.getEmployees().size();
```

    - **Not flexible**
    - **Can return too many instances**

Navigating Relationships

What else should I know about fetch type LAZY?

- Only a **HINT**
  - ***Not required to be supported***
- **Is the DEFAULT for @…ToMany relationship**
- **MUST be specified to be used for @…ToOne**
- **Choose wisely! (Think performance)**

Relationships and Weaving

Why weaving?

- Weaving is a solution, not a requirement
  - Proxy or subclassing
  - Bytecode manipulation (weaving or enhancement)

```
@OneToMany Collection<Employee> emps;
@ManyToOne Department dept;
```

- Advantages
  - Lazy fetching
  - More efficient dirty detection

Weaving

What options are there?

- Dynamic weaving
  - In a Java EE platform container (PU loading)
  - *With a -javaagent option (Java Platform client)*
- *Static weaving*
  - *In a non-Java EE Platform container (no requirement to be supported)*
  - *To be able to **deserialize** a weaved entity on the client side*
  - Requires use of *provider-specific tools*
  - *Makes classes non portable*

Agenda

- Persistence Context
- Entities
- Concurrency
  - Locking Mechanisms in Java Persistence API
  - Concurrency and Bulk Operations
- Query Tips
- Resources and Q&A

Concurrency Options

- “Overly optimistic” concurrency
  - No parallel updates expected or detected
- Optimistic concurrency
  - Parallel updates detected
- Pessimistic concurrency
  - Parallel updates prevented

Parallel Updates to Same Object

No parallel updates expected or detected

```
tx1.begin();
 //Joe's employee id is 5
 e1 = findPartTimeEmp(5);


 //Joe's current rate is $9
e1.raiseByTwoDollar();


tx1.commit();
//Joe's rate will be $11
```

```
tx2.begin();
 //Joe's employee id is 5
 e1 = findPartTimeEmp(5);
              ...

              ...
 //Joe's current rate is $9
 if(e1.getRate() < 10)
   e1.raiseByFiveDollar();
              ...

tx2.commit();
//Joe's rate will be $14
```

How to Guard Against Parallel Updates?

- Use Optimistic concurrency
  - Introduce Version attribute to Employee

```
public class Employee {
    @ID int id;
    ...
    @Version int version;
    ...
}
```

- Results in following SQL

```
"UPDATE Employee SET ..., version = version + 1
                WHERE id = ? AND version = readVersion"
```

- OptimisticLockException if mismatch

Parallel Updates to Same Object

@Version attribute enables detection of parallel updates

```
tx1.begin();
 //Joe's employee id is 5
 //e1.version == 1
 e1 = findPartTimeEmp(5);


 e1.raiseByTwoDollar();


tx1.commit();
//e1.version == 2 in db
```

```
tx2.begin();
 //Joe's employee id is 5
 //e1.version == 1
 e1 = findPartTimeEmp(5);
            ...

            ...
 //Joe's current rate is $9
 if(e1.getRate() < 10)
   e1.raiseByFiveDollar();
            ...
//e1.version == 1 in db?
tx2.commit();
//Joe's rate will be $14
//OptimisticLockException
```

Using Stale Data for Computation

@Version does not help here…

```
tx1.begin();
 d1 = findDepartment(dId);

 //d1's original name is
 //"Engrg"
 d1.setName("MarketEngrg");

//Check d1.version in db
tx1.commit();
```

```
tx2.begin();

 e1 = findEmp(eId);
 d1 = e1.getDepartment();
 if(d1's name is "Engrg")
    e1.raiseByTenPercent();


//Check e1.version in db
tx2.commit();
//e1 gets the raise he does
//not deserve
```

Using Stale Data for Computation

Read lock ensures non-stable data at commit

```
tx1.begin();
 d1 = findDepartment(dId);

 //d1's original name is
 //"Engrg"
 d1.setName("MarketEngrg");

tx1.commit();
```

```
tx2.begin();

 e1 = findEmp(eId);
 d1 = e1.getDepartment();
 em.lock(d1, READ);
 if(d1's name is "Engrg")
    e1.raiseByTenPercent();



//Check d1.version in db
tx2.commit();
//e1 gets the raise he does
//not deserve
//Transaction rolls back
```

Using Stale Data for Computation

Write lock prevents parallel updates

```
tx1.begin();
 e1 = findDepartment(dId);

 //d1's original name is
 //"Engrg"
 d1.setName("MarketEngrg");

tx1.commit();
//tx rolls back
```

```
tx2.begin();

 e1 = findEmp(eId);
 d1 = e1.getDepartment();
 em.lock(d1, WRITE);
 em.flush(); //version++ for d1
 if(d1's name is "Engrg")
    e1.raiseByTenPercent();



tx2.commit();
```

Optimistic versus Pessimistic Concurrency

- Pessimistic Concurrency
  - Lock the row when data is read in
    - Issue “SELECT ... FOR UPDATE” SQL to read data
    - Use different connection isolation level
  - Pros—Simpler application code
  - Cons—Database locks
    - No portable support in this version of spec
  - Suitable when application has many parallel updates
- Optimistic Concurrency
  - Pros—No database locks held
  - Cons—Requires a version attribute in schema
    Databases are not optimized for rollback
    Retries complicate application logic
  - Suitable when application has few parallel updates

Bulk Updates

Executed directly against database
Data in current persistence context not updated

```
tx.begin();
  int id = 5;  //Joe's employee id is 5
  e1 = findPartTimeEmp(id); //Joe's current rate is $9

  //Give Big raise
  em.createQuery(
     "Update Employee set rate = rate * 2").
      executeUpdate();

  //Joe's rate is still $9 in this persistence context
  if(e1.getRate() < 10)
    e1.raiseByFiveDollar();

tx.commit();
//Joe's salary will be $14
```

Bulk Updates and Concurrency

Version column not updated

```
tx1.begin();

//"Update Employee set
//      rate = rate * 2"
giveBigRaise();

//Version not updated in db
tx1.commit();
```

```
tx2.begin();
 //Joe's employee id is 5
 e1 = findPartTimeEmp(5);
              ...

              ...

 //Joe's current rate is $9
 if(e1.getRate() < 10)
   e1.raiseByFiveDollar();
              .
//Check e1.version in db
tx2.commit();
//Poor Joe, his rate
//will be $14
```

Bulk Updates and Concurrency

Explicitly update version column

```
tx1.begin();

//"Update Employee set
//    rate = rate * 2
//    version = version + 1"
giveBigRaiseCorrectly();

//Version not updated
//Version updated in db
tx1.commit();




//Joe's rate will be $18
```

```
tx2.begin();
 //Joe's employee id is 5
 e1 = findPartTimeEmp(5);
              ...

              ...

 //Joe's current rate is $9
 if(e1.getRate() < 10)
   e1.raiseByFiveDollar();
              ...

//Check e1.version in db
tx2.commit();
//Poor Joe, his rate
//will be $14
//OptimisticLockException
```

Agenda

- Persistence Context
- OR Mapping
- Concurrency
- Query Tips
  - Named Queries
  - Dynamic Queries
  - Native Queries
- Resources and Q&A

Named Queries

- Query compilation cached by providers

```
@NamedQuery(name ="findByName",
            query="SELECT e FROM Employee e WHERE
                   e.firstName LIKE :name")
    ...
emps = em.createNamedQuery("findByName").
setParameter("name", "John%").getResultList();
```

- Can be easily externalized into orm.xml
  - Refactoring friendly
  - Can be easily overridden
- Shares same name space
  - Use qualified name—"Employee.findByName"

Dynamic Queries

```
userInput = "foo' OR
    e.salary > 100000 OR
e.firstName = 'bar"
```

- Not the best practice
- Possible to inject malicious/incorrect query

```
em.createQuery("SELECT e FROM Employee e WHERE
        e.firstName LIKE '" + userInput + "'");
```

  - Use parameter markers

```
em.createQuery("select e from Employee e where
        e.name LIKE :name");
```

- The compiled form of dynamic queries might not be cached
- Might need to use dynamic query for variable number of parameters

```
// The IN clause is created in a loop from user
// parameters
em.createQuery("select e from Employee e where
        e.name IN (?1, ?2,...., ?n)");
```

Native Queries

- Use native queries in situations where
  - Native database querying facilities are needed OR
  - Support for SQL features such as Stored Procedures is needed
- Carefully evaluate the usage of native queries because it
  - Ties your queries to database schema
  - Mostly, persistence provider will do a better job of writing SQL than us
  - Reduces cross-database portability

Agenda

- Persistence Context
- Entities
- Concurrency
- Query Tips
- Resources and Q&A

For More Information

- Project GlassFish Forum
  http://forums.java.net/jive/forum.jspa?forumID=56
- Email
  persistence@glassfish.dev.java.net
- java.sun.com/persistence
- Blogs
  - blogs.sun.com/marina
  - weblogs.java.net/rimapatel
  - blogs.sun.com/GlassFishPersistence

GlassFish Community

http://glassfish.java.net/

GlassFish V2 Beta—Available Now!

Production-ready Java EE 5 Platform Application Server

- Clustering, high availability, load balancing
- Improved performance
- WSIT (Web Services Interoperability Tech)
- Ajax, Scripting and REST-based services
- Comet – enables event-driven Web Apps

GlassFish V3 Themes

Web 2.0 Application Server

- Open, modular, extensible Platform
- Multi-Language: RoR, PHP, JavaScript™ programming language
- Ease-of -use features

Simplify development with **Java EE Platform 5**

EJB 3.0, JavaServer Faces 1.2, JSP 2.1, Servlet 2.5, JAX-WS 2.1.1 and JAXB 2.1

Vibrant Ecosystem with over **26** Projects

Over **6900** members and **2.5M** downloads

Free to download, deploy and distribute

Support: java.sun.com/javaee/support

Adoption: blogs.sun.com/stories

News: blogs.sun.com/theaquarium

Blog about Project GlassFish for a chance to **Win a 52-inch LCD HD TV**
Go to java.sun.com/javaee and click “Fish for TV” button on the right

JSP = JavaServer Pages™ | JAX-WS = Java APIs for XML Web Services | JAXB = Java Architecture for XML Binding

Q&A

Backup Slides

Container Managed EM

How do I get a hold of it?

- Injected

```
@Stateless
public class MyBean implements MyInterface {
    @PersistenceContext(unitName = “MyPU”)
    private EntityManager em;
}
```

- Looked up from J.N.D.I. API

```
@PersistenceContext(name="xyz", unitName="MyPU")
public class MyServlet extends HttpServelet {
    public void doGet(...)  {
        InitialContext ic = new InitialContext();
        EntityManager em = (EntityManager)
            ic.lookup("java:comp/env/xyz");
    }
```

Application Managed EM

How do I get a hold of it?

- Always created from EntityMangerFactory

```
public class MyServlet extends HttpServlet {

    @PersistenceUnit(unitName="MyPU")
    private EntityManagerFactory emf;

    public void doGet(...)  {
        EntityManager em = emf.createEntityManager();
        ....
        em.close();
    }
}
```

Application Managed EM

How do I get a hold of an EntityManagerFactory?

- Java EE Platform 5 Container
  - Injected

```
@PersistenceUnit EntiyManagerFactory emf;
```

  - Looked up from J.N.D.I. API

- Java Platform, Standard Edition (Java SE Platform) or non-Java EE Platform 5 Container

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("MyPU");
.....
emf.close();
```

Application or Container Managed EM

Which components support injection?

- *Servlet*
  - *Servlets, servlet filters, event listeners*
- *JavaServer Pages™ (JSP page)*
  - *Tag handlers, tag library event listeners*
- *JavaServer Faces technology*
  - *Scoped managed beans*
- *EJB technology*
  - *Beans, interceptors*
- *Application Client Container*
  - *Main class (static only)*

Source:Java EE platform 5 spec, Table EE.5-1Component classes supporting injection

Stored Procedures

How to invoke them

- Not specified by Java Persistence API
- Three ways to do this
  - Can use persistence provider specific support
  - Can use the underlying Java DataBase Connectivity (JDBC™) Connection object, if provider allows it, and create a CallableStatement
  - Can specify User-Defined Functions (UDF) and wrap SQL Procedures in them
    - You can call UDFs through SQL SELECT statements, and hence, use native queries in Java Persistence API
- Options 1 and 2 locks into the provider whereas Option 3 is a persistence provider agnostic way