



In-Depth OpenJPA

Patrick Linskey
plinskey@bea.com

Introductions

Patrick Linskey

- EJB Team Lead at BEA
- Frequent presenter at JUGs, conferences (JavaOne, JavaPolis, JA00, etc.)
- Active member of EJB and JDO expert groups
- Co-author of *Bitter EJB*

EJB 3.0: A Two-Part Specification

- EJB 3 specification consists of two documents:
 - ▶ Session and Message-Driven Beans
 - ▶ EJB 3 Java Persistence API
- Future versions of the Persistence portion of the EJB 3.0 specification will be spun off into a separate specification

Some Code

```
import javax.persistence.*;

@Repository
public OrderManager {

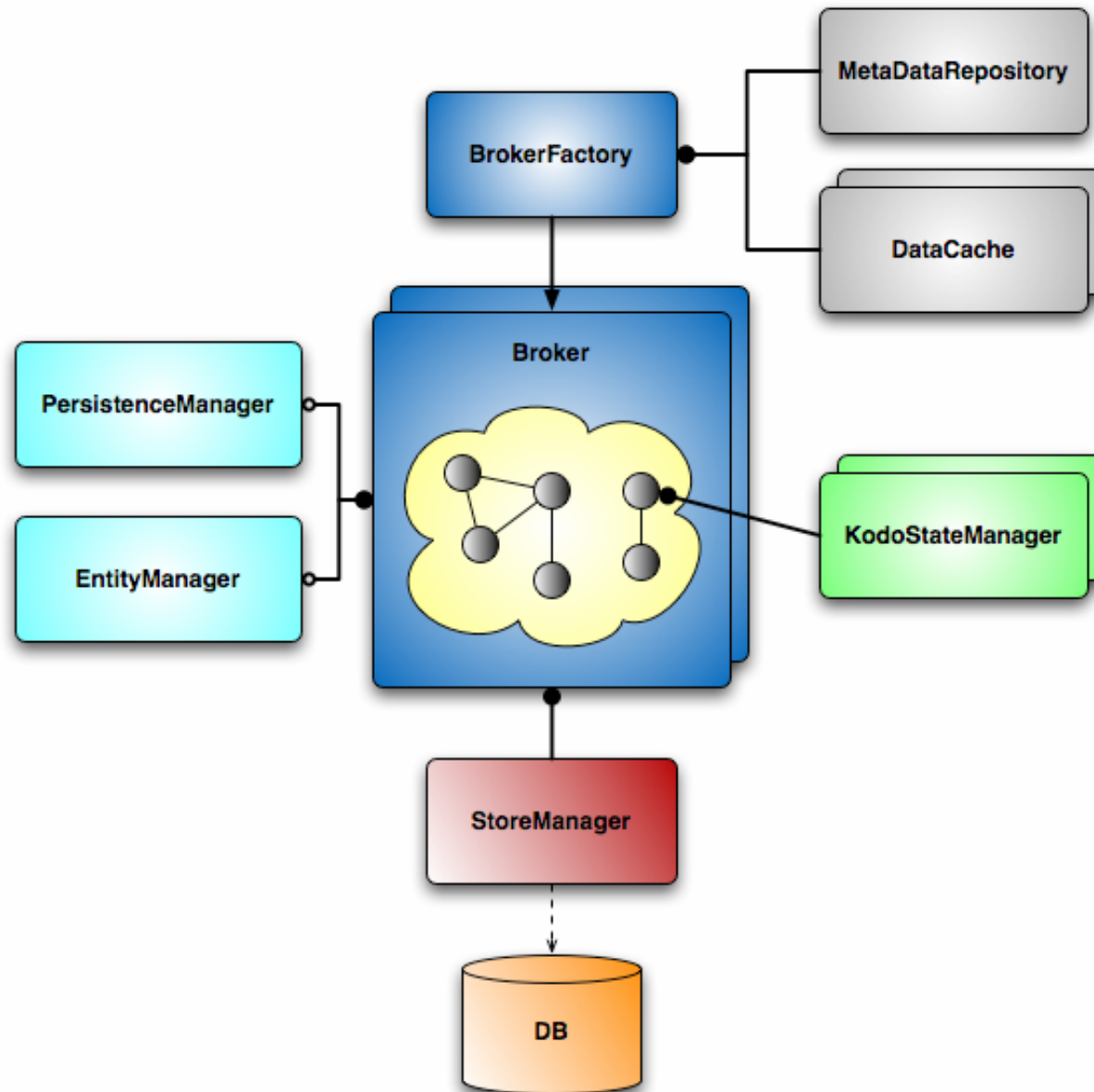
    @PersistenceContext private EntityManager em;

    public Order newOrderForProduct(long custId, long prodId) {
        Customer c = em.find(Customer.class, custId);
        Product p = em.find(Product.class, prodId);

        Order o = new Order(customer);
        em.persist(o);
        o.addLineItem(new Item(p));

        return o;
    }
}
```

OpenJPA Architecture

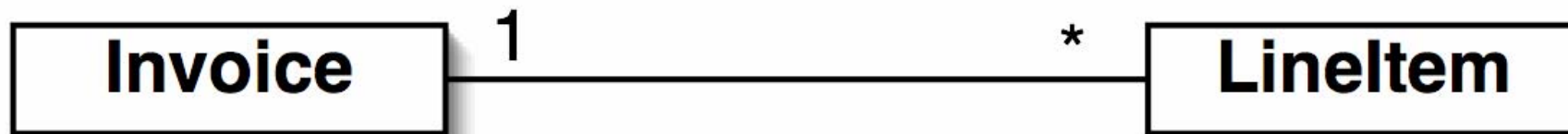


Dirty Tracking

- OpenJPA instruments bytecodes to actively track mutations
 - As you make changes to persistent objects, OpenJPA maintains change sets automatically
 - At flush time, OpenJPA immediately knows which objects are dirty and unflushed
 - After flush, those objects that were flushed change state, so next time the flush will ignore them
 - OpenJPA only holds hard references to dirty, unflushed data
 - OpenJPA does not hold a copy of persistent data*
-
- Results
 - ▶ Transactions can involve arbitrarily many objects
 - ▶ Transaction flush requires very little in-VM work

Large Relationships

- Usually, many-valued relations (Collection and Map field types) should be loaded all at once as soon as the relation is touched

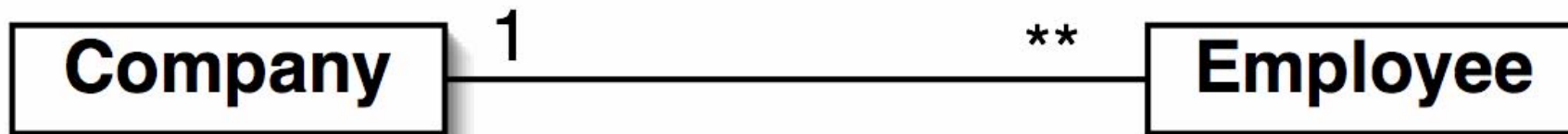


- Load all **LineItems** in relation when **Invoice.lineItems** is first accessed

Large Relationships

- For large relations, this is non-ideal

(** is for “high cardinality”)



- **Do not load** entire Employee relation when the Company.employees field is accessed
- Company.employees.contains(foo) should issue a check for containment in SQL
- Must be possible to mark which relations should be left in the database (the big ones) and which should be brought into memory (the small ones)

Large Relationships

@Entity

```
public class Company {
```

```
    ...
```

```
    @LRS
```

```
    @OneToMany
```

```
    private Collection employees;
```

```
}
```

Connection Retain Mode

- Connections to the database are expensive
- Lifecycle of a “business transaction” may be quite long
 - ▶ May span multiple network round-trips
- It’s undesirable to keep connections open for a long time

- Solution: `ConnectionRetainMode`
- Default setting: `on-demand`
- Other options: `transaction`, `always`
- Typically configured globally in `persistence.xml`

Data Caching

- OpenJPA caches object data and JPQL query results
- Look-aside cache
- Updated when data is loaded from database and after changes are successfully committed
- In a multi-VM environment, all members of the OpenJPA cache cluster are notified when changes are made
- Default notification causes mutated records to be evicted
- Can plug in more aggressive cache implementations, such as Tangosol's Coherence product

Data Cache Expiration

- As other non-OpenJPA applications access the database, the data cache may become stale
- OpenJPA supports several cache eviction strategies:

- ▶ Time-to-live settings for cache instances

```
@Entity @DataCache(timeout=5000)
public class Person { ... }
```

- ▶ cron-style eviction scheduling

```
<property name="openjpa.DataCache"
value="EvictionSchedule='5 0 * * 1'"/>
```

- ▶ Direct API usage

```
DataCache.remove(Object oid)
```

Self-healing DataCache Characteristics

- If changes to the database go undetected by the various cache eviction / notification mechanisms, OpenJPA will lazily heal the data cache
- When an optimistic lock exception is raised while flushing changes to the database, the data cache will check whether the data should have successfully committed based on the current data cache values
- If the data looks correct, the data cache is out-of-date. OpenJPA will automatically evict the questionable data from the cache
- When you repeat the transaction, the entry will not be in cache, so you will get fresh values from the database

Data Caching and Large Transactions

- As mentioned earlier, OpenJPA supports transactions involving potentially more data than JVM heap size
- The data cache must track changed records to keep remote VMs in sync
- Also must update local cache at the end of the transaction
- These features imply that changed records must be tracked
- OpenJPA provides two API calls to help reduce memory footprint for large transactions
 - ▶ `OpenJPAEntityManager.setLargeTransaction(true)`
 - ▶ `OpenJPAEntityManager.setPopulateDataCache(false)`



OpenJPAEntityManager.setLargeTransaction

- Invoke this method at the beginning of a transaction that will involve many objects
- OpenJPA will track classes that change instead of OIDs
- At commit time, OpenJPA will broadcast just the changed classes to other cache cluster members
- All instances of changed classes will be evicted from the local data cache

OpenJPAEntityManager.setPopulateDataCache

- Useful for transactions that visit objects that are very unlikely to be accessed by other transactions
 - ▶ For example an exhaustive report run only once a month
- Tells OpenJPA to not load data into cache as the records are loaded from the database
- Prevents the transaction from churning the cache with data that won't be reused
- ... but still keeps the cache consistent as changes are made

Query Compilation Caching

- When a query is executed, OpenJPA creates a parse tree for the JPQL statement
- This parsing process is relatively expensive
- To eliminate this overhead, OpenJPA caches these parse trees when the query is first executed
- Subsequent executions of the same JPQL statement will bypass the entire query parsing exercise

Fetch Plans

- The `FetchPlan` interface configures database accesses

<code>EagerFetchMode</code>	How to load relations. Options: <code>none</code> , <code>join</code> , <code>parallel</code>
<code>SubclassFetchMode</code>	How to load subclasses. Options: <code>none</code> , <code>join</code> , <code>parallel</code>
<code>MaxFetchDepth</code>	Upper limit on the number of relations to eagerly load
<code>FetchBatchSize</code>	The JDBC batch size to use for the statement
<code>LRSSize</code>	How to compute the size of large result set. Options: <code>query</code> , <code>last</code> , <code>unknown</code>
<code>QueryResultCache</code>	Whether or not query results should be enlisted in cache
<code>addFetchGroup</code> , <code>removeFetchGroup</code> , <code>addField</code> , <code>removeField</code> etc.	Configures which named fetch groups and fields should be eagerly loaded
<code>ReadLockMode</code> , <code>WriteLockMode</code>	Configures how locks should be obtains for reads and writes

Fetch Groups

```
@FetchGroup(name="detail" attributes={
    @FetchAttribute(name="dept"),
    @FetchAttribute(name="manager", recursionDepth="1") } )
@Entity
public class Employee {
    @Id private int id;
    private String firstName;
    private String lastName;
    @ManyToOne private Department dept;
    @OneToOne private Employee manager;
}
```

```
Query q = em.createQuery("select e from Employee e");
List l = q.getResultList();
==> SELECT e.id, e.first_name, e.last_name, e.dept_id, e.manager_id FROM Employee e
```

```
Query q = em.createQuery("select e from Employee e");
((OpenJPAQuery) q).getFetchPlan().addFetchGroup("list");
List l = q.getResultList();
==> SELECT e.id, e.first_name, e.last_name, d.id, d.name, m.id, m.first_name, m.last_name
FROM Employee e, Department d, Manager m
WHERE e.dept_id = d.id AND e.manager_id = m.id
```

Fetch Groups and detachment

- If the `openjpa.DetachState` property is set to `fgs`, then OpenJPA will:
 - ▶ Detach all of the fields in the EM's fetch group at detach time
 - ▶ Unload any other fields that were loaded earlier in the transaction
- Useful for security
 - ▶ Declaratively prevent secure data (SSNs, account numbers, etc.) from being available in other tiers



DetachState setting and detached field access

- By default, when detaching records and operating on them in other tiers, the user will get NullPointerExceptions (or default values) when attempting to access fields that were not detached
- This can lead to difficulties with debugging and programming in a detached environment
- OpenJPA can throw meaningful exceptions when you navigate detached objects
- To do this, OpenJPA must store bookkeeping information in detached objects, which changes their serialization footprint

Savepoints

```
String OpenJPAEntityManager.setSavepoint();  
void OpenJPAEntityManager.rollbackToSavepoint(String);  
void OpenJPAEntityManager.releaseSavepoint(String);
```

- Allows incremental rollbacks during the course of a larger transaction
- If the underlying database supports savepoints, that support is used
- Otherwise, savepoints are implemented in-memory

Externalization

- The JPA spec supports a limited number of “intrinsic”
 - ▶ `String`, `byte`, `boolean`, `char`, `double`, `float`, `int`, `long`, `Byte`, `Boolean`, `Character`, `Double`, `Float`, `Integer`, `Long`, `BigInteger`, `BigDecimal`, `Date`, `Calendar`, `Timestamp`, enums
- You might use other simple types in your domain model
 - ▶ `java.net.URL`, `java.io.File`, etc.
- To do this within the JPA spec, you must:
 - ▶ Create transient fields of the types that you want to use
 - ▶ Create persistent fields of the types to store in the database
 - ▶ Register an `EntityCallback` to transform the values as needed
- OpenJPA simplifies this: `@Externalizer` and `@Factory`

Externalization

```
@Entity public class Person {  
    ...  
    @Persistent  
    @Externalizer("toExternalForm")  
    private URL homePage;  
}
```

- This example will automatically use 'new URL(String)' to reverse the process
- Other options for generating the domain type are available via @Factory annotation
- The @Persistent annotation instructs OpenJPA to store the field even though it's not one of the standard JPA types.

External Values

- A special case of externalization is to simply apply a lookup table to transform data
- Often useful for supporting enumerated type (or even Java enum types)

```
@Entity
public class Magazine
{
    @ExternalValues({"true=1", "false=2"})
    @Type(int.class)
    private boolean isWeekly;
}
```

Programming Model

- Transaction listeners
- Plural methods
- Multi-threaded support
- Persistent Interfaces
- Attribute-level @ReadOnly
- Dynamically-generated proxies for Maps, Collections, mutable objects

Extended Object-Relational Mapping

- Partial key joins
- Custom Mappings
- Inheritance Options

Questions?

- plinskey@bea.com
- kodo@bea.com
- <http://incubator.apache.org/projects/openjpa>