# Software Architecture 5
# Object-Relation Mapping II

**Haopeng Chen**

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

# Overview



- O/R mapping
  - Basic O/R Mapping
  - Working with Objects
  - Transactions and Concurrency
  - Improving Performance

# Session and transaction scopes



- A SessionFactory is an expensive-to-create, thread-safe object, intended to be shared by all application threads.
  - It is created once, usually on application startup, from a Configuration instance.

- A Session is an inexpensive, non-thread-safe object that should be used once and then discarded for: a single request, a conversation or a single unit of work.
  - A Session will not obtain a JDBC Connection, or a Datasource, unless it is needed. It will not consume any resources until used.

# Unit of work



- A unit of work is a design pattern described by Martin Fowler as
  - "[maintaining] a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems. "
- In other words, its a series of operations we wish to carry out against the database together.
  - Basically, it is a transaction, though fulfilling a unit of work will often span multiple physical database transactions
- Do not use the *session-per-operation* anti-pattern:
  - do not open and close a Session for every simple database call in a single thread.

# Long conversations



- In web and enterprise applications, it is not acceptable for a database transaction to span a user interaction.

- Hibernate's features:
  - *Automatic Versioning*.
  - *Detached Objects*.
  - *Extended (or Long) Session*

- Session span
  - *session-per-operation*
  - *session-per-request*
  - *session-per-conversation*

# Considering object identity



- An application can concurrently access the same persistent state in two different Sessions.
  - However, an instance of a persistent class is never shared between two Session instances

- Database Identity
  - foo.getId().equals( bar.getId() )
- JVM Identity
  - foo==bar

- The developer has to override the equals() and hashCode() methods in persistent classes and implement their own notion of object equality.
  - There is one caveat: never use the database identifier to implement equality.

# Database transaction demarcation



- No communication with the database can occur outside of a database transaction
  - this seems to confuse many developers who are used to the auto-commit mode

- Ending a Session usually involves four distinct phases:
  - flush the session
  - commit the transaction
  - close the session
  - handle exceptions

# Non-managed environment



```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
  tx = sess.beginTransaction();
  // do some work ...
  tx.commit();
}
catch (RuntimeException e) {
  if (tx != null)
     tx.rollback();
  throw e; // or display error message
}
finally {
  sess.close();
}
```

# Non-managed environment



```
// Non-managed environment idiom with getCurrentSession()
try {
  factory.getCurrentSession().beginTransaction();
  // do some work ...
  factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
  factory.getCurrentSession().getTransaction().rollback();
  throw e;   // or display error message
}
```

# Using JTA



```
// BMT idiom
Transaction tx = null;
try {
  UserTransaction tx = (UserTransaction)new
                        InitialContext() .lookup("java:comp/UserTransaction");
 // do some work ...
 tx.commit();
}
catch (RuntimeException e) {
  if (tx != null) tx.rollback();
  throw e;
  // or display error message
}
finally {
  sess.close();
}
```

# Using JTA



```
// BMT idiom with getCurrentSession()
try {
  UserTransaction tx = (UserTransaction)new
                    InitialContext() .lookup("java:comp/UserTransaction");
  tx.begin();
  // Do some work on Session bound to transaction
  factory.getCurrentSession().load(...);
  factory.getCurrentSession().persist(...);
  tx.commit();
}
catch (RuntimeException e) {
  tx.rollback();
  throw e;
  // or display error message
}
```

# Using JTA



```
// CMT idiom
Session sess = factory.getCurrentSession();
// do some work ...

import static TransactionAttributeType.*;
    @Stateless
    @TransactionAttribute(NOT_SUPPORTED)
    public class TravelAgentBean implements TravelAgentRemote {
        public void setCustomer(Customer cust) {...}

        @TransactionAttribute(REQUIRED)
        public TicketDO bookPassage(CreditCardDO card, double price)
        { ... }
    }
```

# Transaction attributes defined

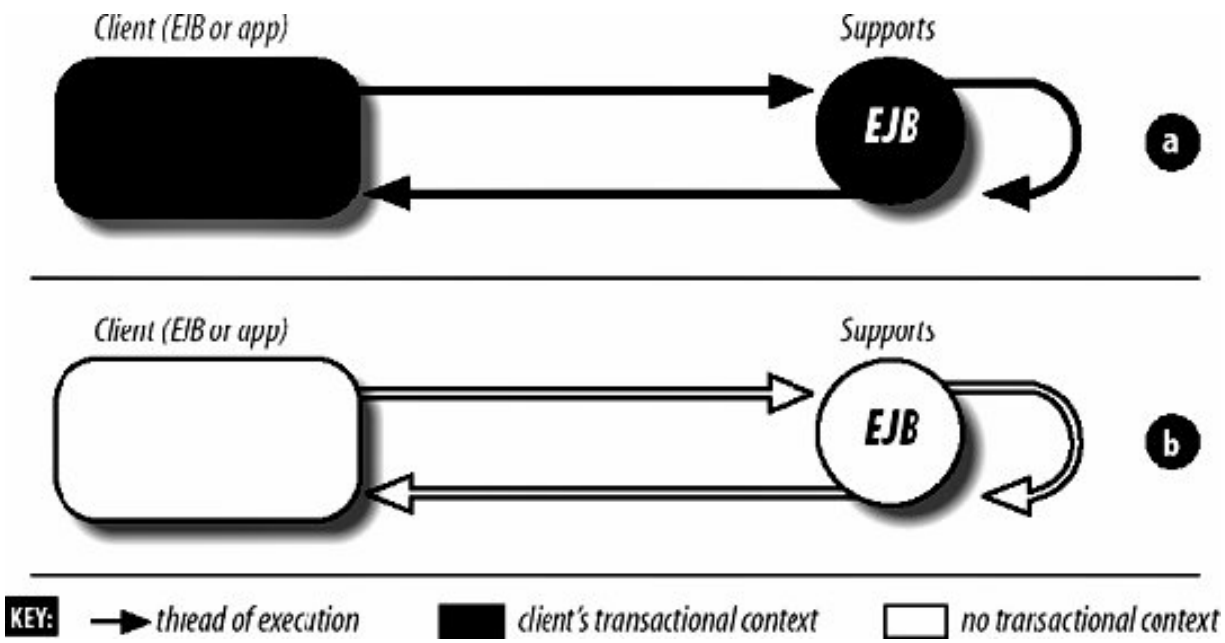


- NotSupported
  - Invoking a method on an EJB with this transaction attribute suspends the transaction until the method is completed.

# Transaction attributes defined



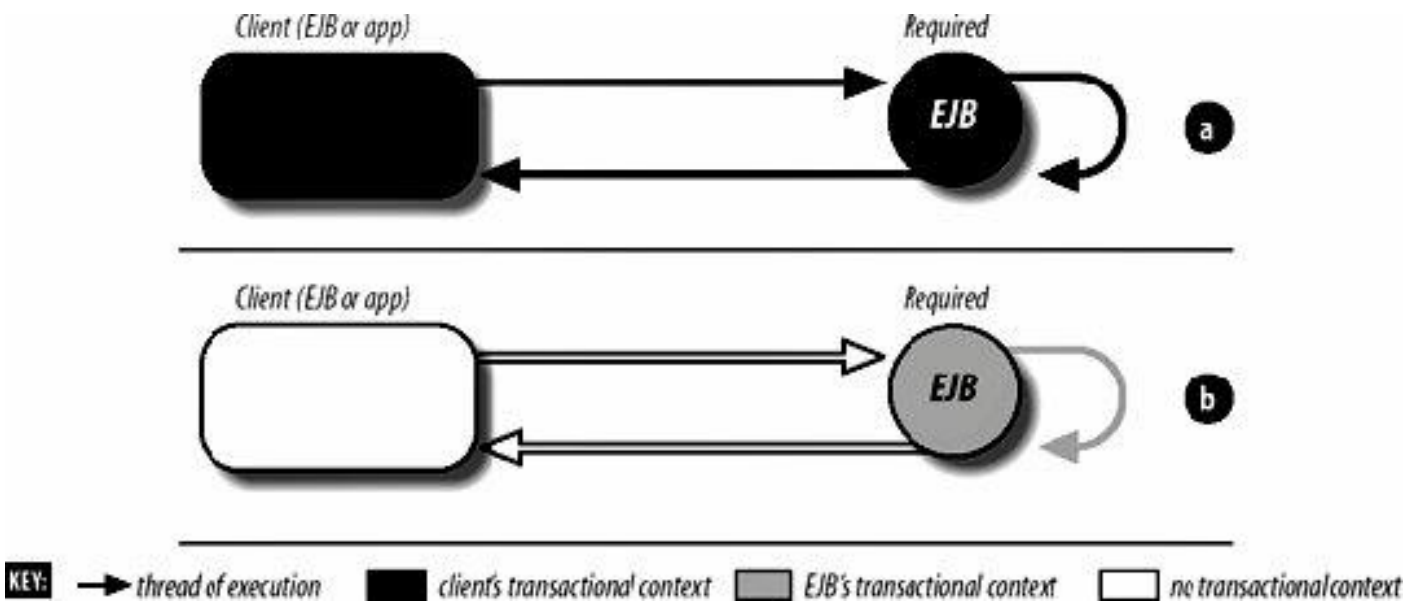
- Supports
  - This attribute means that the enterprise bean method will be included in the transaction scope if it is invoked within a transaction.



# Transaction attributes defined



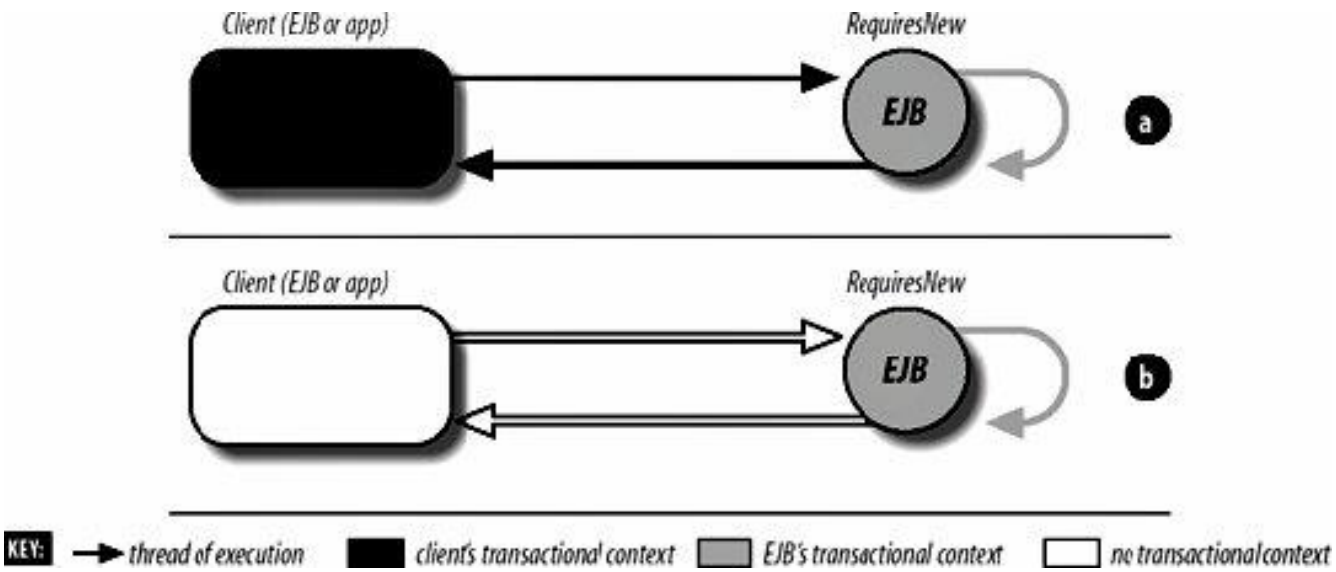
- Required
  - This attribute means that the enterprise bean method must be invoked within the scope of a transaction.



# Transaction attributes defined



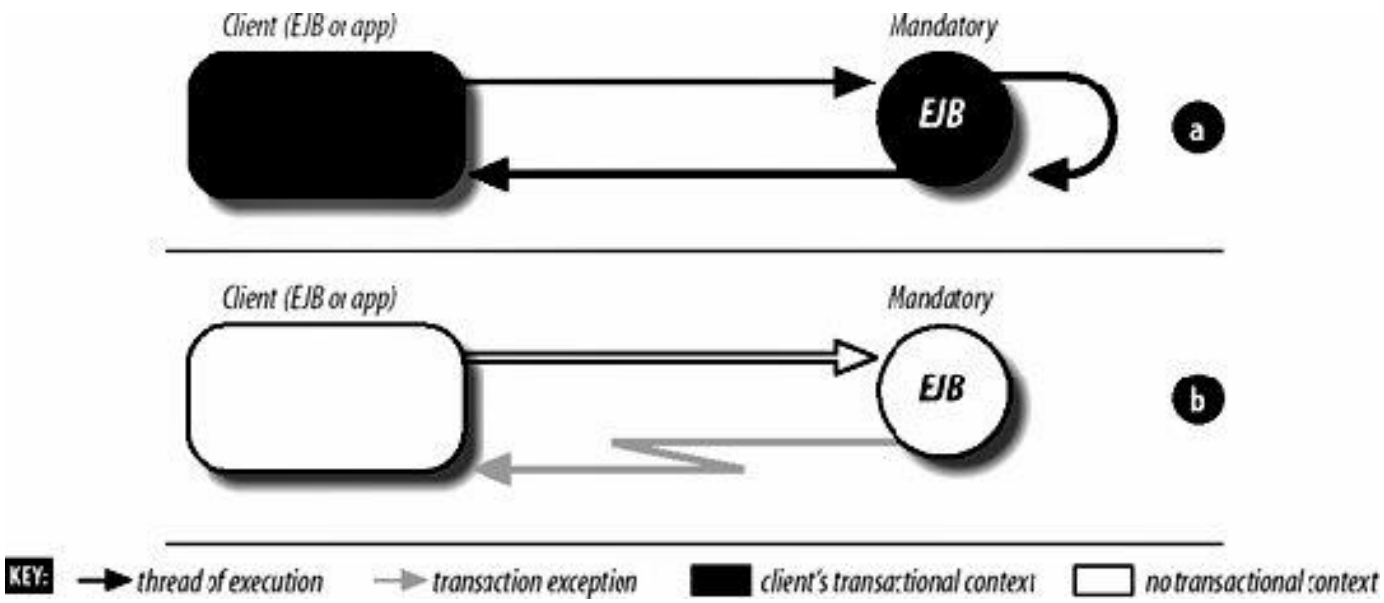
- RequiresNew
  - This attribute means that a new transaction is always started.



# Transaction attributes defined



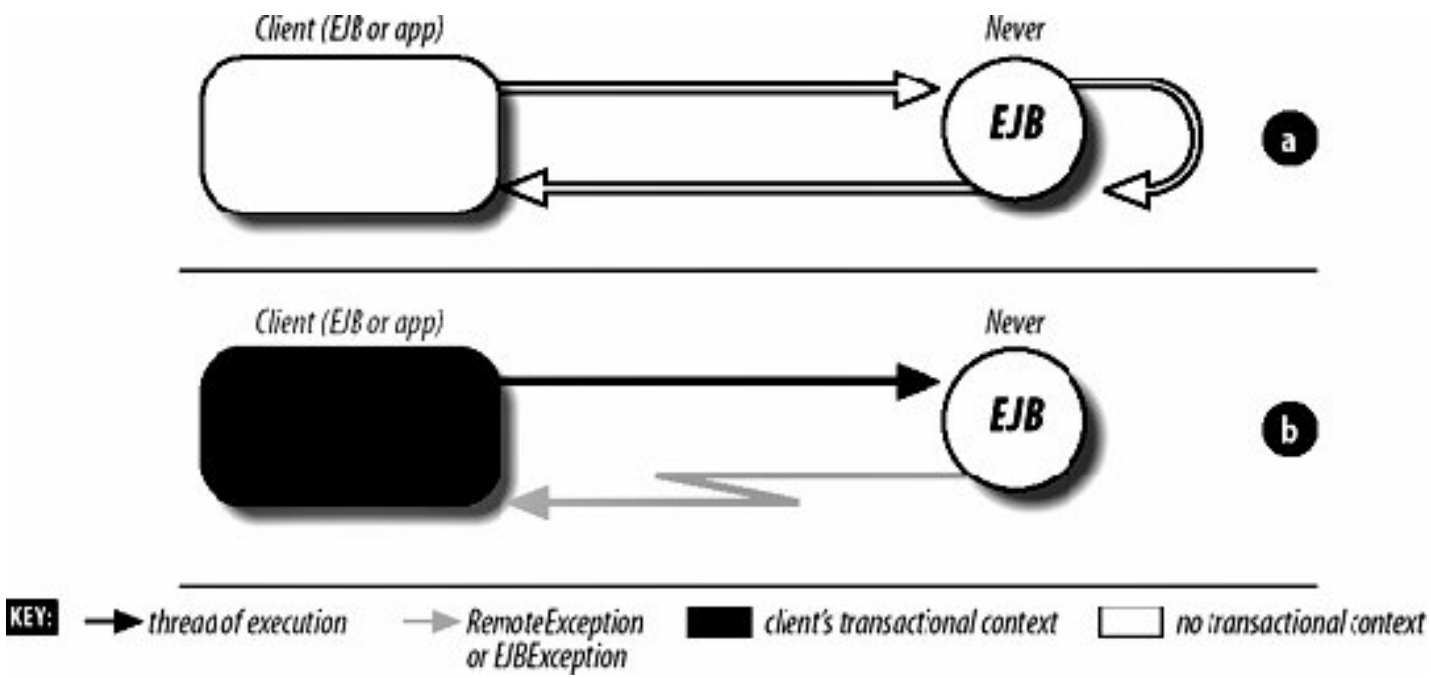
- Mandatory
  - This attribute means that the enterprise bean method must always be made part of the transaction scope of the calling client.



# Transaction attributes defined



- Never
  - This attribute means that the enterprise bean method must not be invoked within the scope of a transaction.



# Transaction attributes defined



| TRANSACTION ATTRIBUTE | CLIENT'S TRANSACTION | BEAN'S TRANSACTION |
|---|---|---|
| Required | None | T2 |
| | T1 | T1 |
| RequiresNew | None | T2 |
| | T1 | T2 |
| Supports | None | None |
| | T1 | T1 |
| Mandatory | None | Error |
| | T1 | T1 |
| NotSupported | none | None |
| | T1 | None |
| Never | none | None |
| | T1 | Error |

# Transaction Timeout



```
Session sess = factory.openSession();
try {
  //set transaction timeout to 3 seconds
  sess.getTransaction().setTimeout(3);
  sess.getTransaction().begin();
  // do some work ...
  sess.getTransaction().commit()
}
catch (RuntimeException e) {
  sess.getTransaction().rollback();
  throw e; // or display error message
}
finally {
  sess.close();
}
```

# Optimistic concurrency control



- The only approach that is consistent with high concurrency and high scalability, is optimistic concurrency control with versioning.
  - Version checking uses version numbers, or timestamps, to detect conflicting updates and to prevent lost updates.

# Application version checking



```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() )
   throw new StaleObjectStateException();
foo.setProperty("bar");
t.commit();
session.close();
```

# Extended session and automatic versioning



```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction();
// Obtain a new JDBC connection, start transaction

foo.setProperty("bar");
session.flush();  // Only for last transaction in conversation
t.commit();       // Also return JDBC connection
session.close(); // Only for last transaction in conversation
```

# Detached objects and automatic versioning



```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo);
// Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

# Pessimistic locking



- The LockMode class defines the different lock levels that can be acquired by Hibernate.
  - LockMode.WRITE
  - LockMode.UPGRADE
  - LockMode.UPGRADE_NOWAIT
  - LockMode.READ
  - LockMode.NONE

# Overview



- O/R mapping
  - Basic O/R Mapping
  - Working with Objects
  - Transactions and Concurrency
  - Improving Performance

# Fetching strategies



- Hibernate3 defines the following fetching strategies:
  - *Join fetching*
  - *Select fetching*
  - *Subselect fetching*
  - *Batch fetching*
- Hibernate also distinguishes between:
  - *Immediate fetching*
  - *Lazy collection fetching*
  - *"Extra-lazy" collection fetching*
  - *Proxy fetching*
  - *"No-proxy" fetching*
  - *Lazy attribute fetching*

# Working with lazy associations



- hibernate.default_batch_fetch_size

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();
User u = (User) s.createQuery("from User u where u.name=:userName")
  .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();
tx.commit();
s.close();
Integer accessLevel = (Integer) permissions.get("accounts");
// Error!
```

- Hibernate does not support lazy initialization for detached objects.

# The second level cache



- specify the name of a class that implements org.hibernate.cache.spi.CacheProvider using the property hibernate.cache.provider_class.

| Cache | Provider class | Type | Cluster Safe | Query Cache Supported |
|---|---|---|---|---|
| ConcurrentHashMap (only for testing purpose, in hibernate-testing module) | org.hibernate.testing.cache.CachingRegionFactory | memory | | yes |
| EHCache | org.hibernate.cache.ehcache.EhCacheRegionFactory | memory, disk, transactional, clustered | yes | yes |
| Infinispan | org.hibernate.cache.infinispan.InfinispanRegionFactory | clustered (ip multicast), transactional | yes (replication or invalidation) | yes (clock sync req.) |

# Cache mappings



- shared-cache-mode element in your persistence.xmlfile or the javax.persistence.sharedCache.mode property in your configuration:
  - ENABLE_SELECTIVE (Default and recommended value)
  - DISABLE_SELECTIVE
  - ALL
  - NONE

- hibernate.cache.default_cache_concurrency_strategy:
  - read-only
  - read-write
  - nonstrict-read-write
  - transactional

# Cache mappings



- **Definition of cache concurrency strategy via @Cache**

```
@Entity
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Forest { ... }
```

- **Caching collections using annotations**

```
@OneToMany(cascade=CascadeType.ALL, fetch=FetchType.EAGER)
@JoinColumn(name="CUST_ID")
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public SortedSet<Ticket> getTickets() {
   return tickets;
}
```

# Cache mappings



- **Cache Concurrency Strategy Support**

| Cache | read-only | nonstrict-read-write | read-write | transactional |
|---|---|---|---|---|
| ConcurrentHashMap (not intended for production use) | yes | yes | yes | |
| EHCache | yes | yes | yes | yes |
| Infinispan | yes | | | yes |

# Managing the caches



- **Explcitly evicting a cached instance from the first level cache using Session.evict()**

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll();
//a huge result set
while ( cats.next() ) {
  Cat cat = (Cat) cats.get(0);
  doSomethingWithACat(cat);
  sess.evict(cat);
}
```

- **Second-level cache eviction via SessionFactoty.evict() andSessionFacyory.evictCollection()**

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class); //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId);
//evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

# Managing the caches



- **Browsing the second-level cache entries via the Statistics API**

```
Map cacheEntries = sessionFactory.getStatistics()
    .getSecondLevelCacheStatistics(regionName)
    .getEntries();
```

- **Enabling Hibernate statistics**

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

# Query cache regions



- If you require fine-grained control over query cache expiration policies, you can specify a named cache region for a particular query by calling Query.setCacheRegion().

```
List blogs = sess.createQuery
  ("from Blog blog where blog.blogger = :blogger")
 .setEntity("blogger", blogger)
 .setMaxResults(15)
 .setCacheable(true)
 .setCacheRegion("frontpages")
 .list();
```

# Monitoring a SessionFactory



```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name
StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server

// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name
StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

# Metrics



```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();
double queryCacheHitCount = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio = queryCacheHitCount /
                    (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
stats.getEntityStatistics( Cat.class.getName() );
long changes =
   entityStats.getInsertCount()
   + entityStats.getUpdateCount()
   + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times" );
```

# References



- HIBERNATE - Relational Persistence for Idiomatic Java, http://docs.jboss.org/hibernate/orm/4.1/manual/en-US/html_single/#preface

# Thank You!