Hibernate Transaction

"Code with Passion!"

Topics

- Session and transaction scopes
- Transaction demarcations
  - JTA-based
  - JDBC-based
  - Interceptor (AOP)-based
- Locking
  - Optimistic locking
  - Pessimistic locking

Session Scope Patterns

Session as a Unit of Work

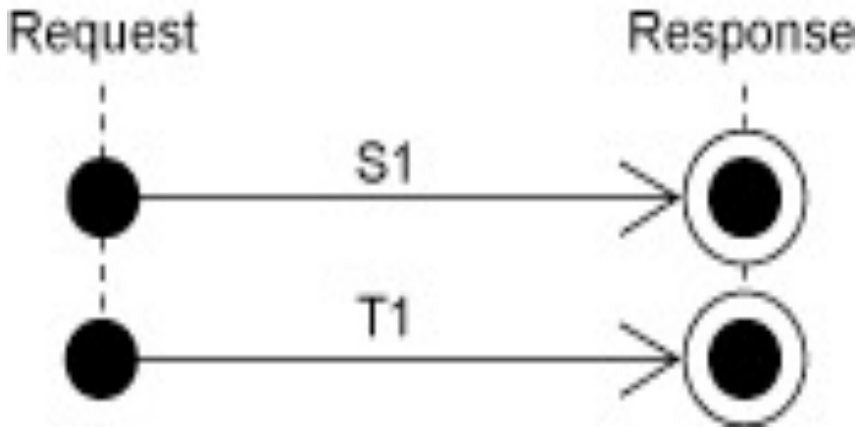
- A unit of work is considered as “grouping data access operations”.
- We usually refer to a Hibernate Session as a unit of work because the scope of a Session is exactly that
  - To begin a unit of work, you open a Session
  - To end a unit of work you close a Session
- Session scope patterns
  - Session per operation pattern
  - Session per transaction pattern
  - Session per transaction with detached objects pattern
  - Session for long conversation pattern

Session Per Operation pattern

- In a “Session per operation” pattern, a single Hibernate Session has the same scope as a single database operation
  - You create a new Session for every database operation
- Strongly discouraged

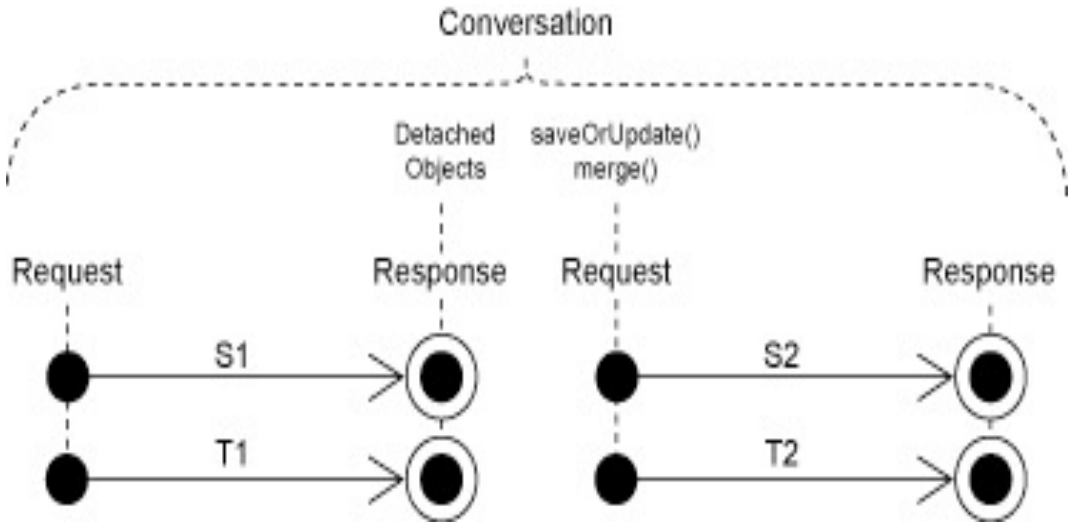
Session Per Transaction pattern

- A single Hibernate Session has the same scope as a single database transaction
- A single Session and a single database transaction implement the processing of a particular request event (for example, a Http request in a web application).
- Most common pattern for multi-user environment



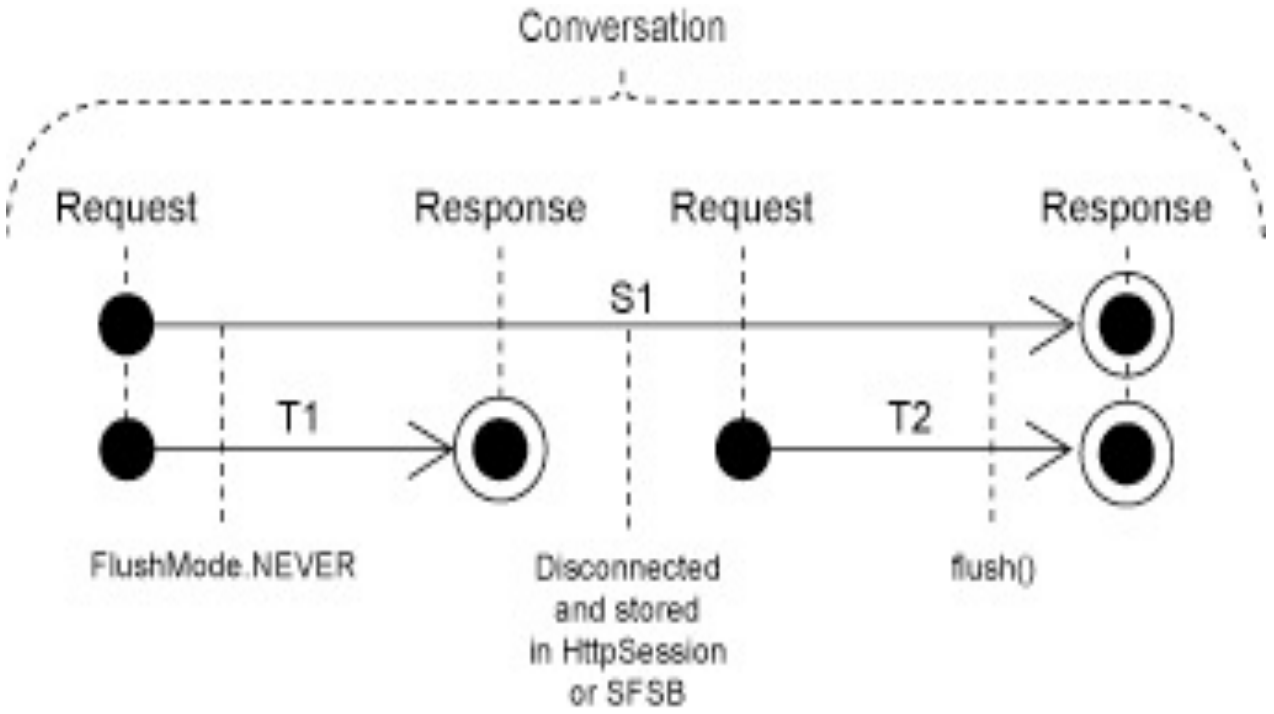
Session Per Transaction with Detached Objects pattern

- Long Conversations, e.g. an application that implements a multi-step dialog, for example a wizard dialog, to interact with the user in several request/response cycles.
- Once persistent objects are considered detached during user think-time and have to be reattached to a new Session after they have been modified.



Session Per Long Conversation pattern

- A single Session has a bigger scope than a single database transaction and it might span several database transactions.



Lab:

Exercise 1: Session Scope Patterns
3523_hibernate_transaction.zip

Transaction Demarcation

Transaction is always on

- No communication with the database can occur outside of a database transaction
  - All Hibernate life-cycle operations occur within a transaction
- Always use clear transaction boundaries, even for read-only operations.

Hibernate Application Running Modes

- A Hibernate application runs in one of two modes
  - Non-managed (i.e., standalone, simple Web- or Swing applications)
  - Managed (by container) environments
- Non-managed environment
  - The application developer has to manually set transaction boundaries (begin, commit, or rollback database transactions) themselves
- Managed environment
  - A managed environment usually provides container-managed transactions (CMT), with the transaction assembly defined declaratively

Portability of Applications

- It is often desirable to keep your persistence layer portable between non-managed resource-local environments, and systems that can rely on JTA but use BMT instead of CMT
  - In both cases, programmatic transaction demarcation is used
- Hibernate offers a wrapper API called *Transaction* that translates into the native transaction system of your deployment environment.
  - This API is actually optional, but we strongly encourage its use unless you are in a CMT session bean.

Ending a Session

- Ending a Session usually involves four distinct phases:
  - Flush the session
  - Commit or rollb the transaction
  - Close the session
  - Handle the exception

Non-Managed Environment

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    // You do not have to flush() the Session explicitly: the call to commit()
    // automatically triggers the synchronization depending on the FlushMode
    // for the session.
    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Managed JTA BMT Environment - Option1

```
// Managed JTA environment idiom  - Note this is the same code as previous slide.
// This is portable code for non-Managed and Managed JTA BMT
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    // You do not have to flush() the Session explicitly: the call to commit()
    // automatically triggers the synchronization depending on the FlushMode
    // for the session.
    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Managed JTA BMT Environment - Option2

```
// BMT idiom with explicit transaction demarcation
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
}
```

Transaction API

Transaction Java Interface

- The core of transactions in Hibernate is the *Transaction* interface
  - This interface is implemented to provide three of the in-built classes that do the real work(communication and transaction management at database/JTA level)
- A transaction object is always associated with a session object - To instantiate a transaction object, *beginTransaction()* is called on the session object
  - *Transaction tcx=session.beginTransaction();*

Implementation Classes of Transaction Interface

- Hibernate provides 3 implementation classes
  - *JDBCTransaction* (default implementation)
  - *JTATransaction*
  - *CMTTransaction*
- The choice of Transaction implementation class can be specified in the *hibernate.cfg.xml*

```
<property
 name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory
</property>
```

Methods of Transaction Interface

- begin()
- commit()
- isactive()
- rollback()
- setTimeout(int seconds)
- wasCommitted()
- wasRolledBack()

Lab:

Exercise 2: Transaction API
3523_hibernate_transaction.zip

Types of Transaction Demarcation

Types of Transaction Demarcation

- Non-Managed
  - Transaction demarcation with JDBC
- Managed
  - Transaction demarcation with JTA BMT
  - Transaction demarcation with EJB/CMT (Container Managed Transaction)
- Transaction Interceptors

Transaction Demarcation with JDBC

JDBC-based Transaction Demarcation

- Use it when you don't have JTA support
  - Mostly used in standalone application
- Two types
  1. You manage session
  2. Hibernate manages session

Transaction Demarcation with JDBC - You manage a session

JDBC-based Transaction Demarcation: You manage session

- Default setting in the *hibernate.cfg.xml* - not setting it as the same as following

```
<property
 name="transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory
</property>
```

- You get a session by calling *openSession()* method of the SessionFactory
  - *Session session = sessionFactory.openSession();*
- You have to call close() method of the session object
  - *session.close();*

JDBC-based Transaction Demarcation: You manage a session

```
// Create SessionFactory and Session object
SessionFactory sessionFactory =
      new Configuration().configure().buildSessionFactory();
Session session = sessionFactory.openSession();

// Perform life-cycle operations under a transaction
Transaction tx = null;
try {
    tx = session.beginTransaction();

    // Create a Person object and save it
    Person p1 = new Person();
    p1.setName("Sang Shin");
    session.save(p1);
} catch ( HibernateException e ) {
    if ( tx != null ) tx.rollback();
    e.printStackTrace();
 } finally {
    session.close();
 }
```

Transaction Demarcation with JDBC - Hibernate manage a session

JDBC-based Transaction Demarcation: Hibernate manages session

- A new session is opened when *getCurrentSession()* is called for the first time, but in a "proxied" state that doesn't allow you to do anything except start a transaction.
- When the transaction ends, either through commit or roll back, the "current" Session is closed automatically.
- The next call to *getCurrentSession()* starts a new proxied Session, and so on.

JDBC-based Transaction Demarcation: Hibernate manages session

```
try {
   // The first time you call getCurrentSession(), you have to
   // call beginTransaction() on the returned Session object.
   factory.getCurrentSession().beginTransaction();

    // Do some work
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

Configuration

- To enable the thread-bound strategy in your Hibernate configuration:
  - Set *hibernate.transaction.factory_class* to *org.hibernate.transaction.JDBCTransactionFactory*
  - Set *hibernate.current_session_context_class* to *thread*

```
<property name="hibernate.transaction.factory_class">org.hibernate.transaction.JDBCTransactionFactory
</property>
<property name="hibernate.current_session_context_class">thread
</property>
```

Transaction Demarcation with JTA

What is JTA?

- Used when transaction needs to be performed against multiple transactional resources, such as a database and a message queue
- JTA support is built-in in all Java EE app. servers
  - A Datasource you use in such a container is automatically handled by a JTA TransactionManager.

JTA and Hibernate

- Hibernate automatically binds the "current" Session to the current JTA transaction.
  - This enables an easy implementation of the session-per-request strategy with the *getCurrentSession()* method on your *SessionFactory*
  - All you have to do is start your JTA transaction, and *sessionFactory.getCurrentSession()* will return the Hibernate session associated with that JTA transaction.
  - As long as the JTA transaction is still pending, every call to *getCurrentSession()* will return the same session that is associated with the transaction.

JTA-based Explicit Transaction Demarcation

```
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                        .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

Hibernate Configuration

- Set *hibernate.transaction.manager_lookup_class* to a lookup strategy for your Java EE container
- Set *hibernate.transaction.factory_class* to *org.hibernate.transaction.JTATransactionFactory*

Lab:

Exercise 3: Transaction Demarcation
3523_hibernate_transaction.zip

Optimistic Concurrency Control

What is Optimistic Concurrency Control (OCC)?

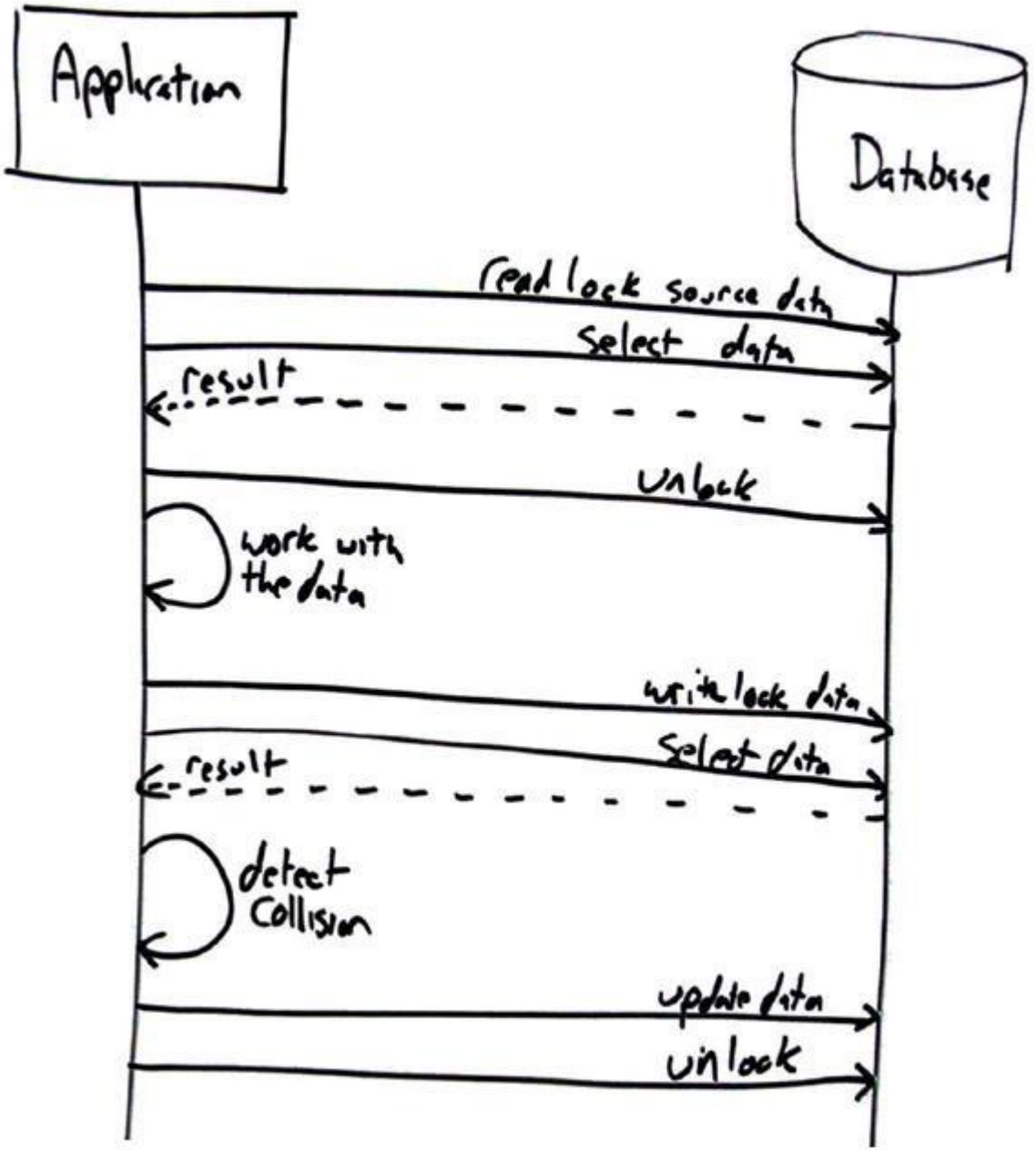
- An approach to concurrency in which an item is locked only during the time that it is accessed in the persistence mechanism
  - For example, if a customer object is edited, a lock is placed on it during the time that it takes to read it in memory and then it is immediately removed.
  - The object is edited and then when it needs to be saved, it is locked again, written out, then unlocked.

What Is the Issue of Optimistic Concurrency Control?

- OCC allows many people to work with an object simultaneously, but also presents the opportunity for people to overwrite the work of others.

When to use Optimistic Concurrency Control?

- Optimistic Concurrency Control (OCC) is based on the assumption that most database transactions don't conflict with other transactions, allowing OCC to be as permissive as possible in allowing transactions to execute.
- When conflicts are rare, validation can be done efficiently, leading to higher throughput



How Optimistic Concurrency Control Work

Optimistic Concurrency Control Scheme

- Optimistic concurrency control with versioning provides high concurrency and high scalability.
- Version checking uses version numbers, or timestamps, to detect conflicting updates (and to prevent lost updates).
- Hibernate provides for several possible approaches to writing application code that uses optimistic concurrency
  - Application version checking
  - Automatic versioning
  - Customizing automatic versioning

Optimistic Locking Configuration

```
<hibernate-mapping>

   <class name="Person" table="person" optimistic-lock = "version">
      <id name="id" type="int">
         <generator class="increment"/>
      </id>
      <version name="version" column="VERSION" type="int"/>
      <property name="name" column="cname" type="string"/>
   </class>

</hibernate-mapping>
```

Pessimistic Concurrency Control

What Is Pessimistic Concurrency Control?

- An approach to concurrency in which an item is locked in the persistence mechanism for the entire time that it is in memory
  - For example, when a customer object is edited a lock is placed on the object in the persistence mechanism, the object is brought into memory and edited, and then eventually the object is written back to the persistence mechanism and the object is unlocked
- This approach guarantees that an item won't be updated in the persistence mechanism while the item is in memory, but at the same time is disallows others to work with it while someone else does.

When to Use Pessimistic Concurrency Control?

- Pessimistic locking is ideal for batch jobs that need to ensure consistency in the data that they write.

Lab:

Exercise 4: Optimistic Locking & Pessimistic Locking
3523_hibernate_transaction.zip

Code with Passion!