8. Transactions and concurrency controlIt is important to understand that the term transaction has many different yet related meanings in regards to persistence and Object/Relational Mapping. In most use-cases these definitions align, but that is not always the case. It might refer to the physical transaction with the database. It might refer to the logical notion of a transaction as related to a persistence context. It might refer to the application notion of a Unit-of-Work, as defined by the archetypal pattern. This documentation largely treats the physical and logic notions of a transaction as one-in-the-same. 8.1. Physical Transactions Hibernate uses the JDBC API for persistence. In the world of Java, there are two well-defined mechanisms for dealing with transactions in JDBC: JDBC itself and JTA. Hibernate supports both mechanisms for integrating with transactions and allowing applications to manage physical transactions. The transaction handling per Session is handled by the org.hibernate.resource.transaction.spi.TransactionCoordinator contract, which are built by the org.hibernate.resource.transaction.spi.TransactionCoordinatorBuilder service. TransactionCoordinatorBuilder represents a strategy for dealing with transactions whereas TransactionCoordinator represents one instance of that strategy related to a Session. Which TransactionCoordinatorBuilder implementation to use is defined by the hibernate.transaction.coordinator\_class setting. jdbc (the default for non-Jakarta Persistence applications) Manages transactions via calls to java.sql.Connection jta Manages transactions via JTA. See Java EE bootstrapping If a Jakarta Persistence application does not provide a setting for hibernate.transaction.coordinator\_class, Hibernate will automatically build the proper transaction coordinator based on the transaction type for the persistence unit. If a non-Jakarta Persistence application does not provide a setting for hibernate.transaction.coordinator\_class, Hibernate will use jdbc as the default. This default will cause problems if the application actually uses JTA-based transactions. A non-Jakarta Persistence application that uses JTA-based transactions should explicitly set hibernate.transaction.coordinator\_class=jta or provide a custom org.hibernate.resource.transaction.TransactionCoordinatorBuilder that builds a org.hibernate.resource.transaction.TransactionCoordinator that properly coordinates with JTA-based transactions. For details on implementing a custom TransactionCoordinatorBuilder, or simply better understanding how it works, see the Integration Guide . Hibernate uses JDBC connections and JTA resources directly, without adding any additional locking behavior. Hibernate does not lock objects in memory. The behavior defined by the isolation level of your database transactions does not change when you use Hibernate. The Hibernate Session acts as a transaction-scoped cache providing repeatable reads for lookup by identifier and queries that result in loading entities. To reduce lock contention in the database, the physical database transaction needs to be as short as possible. Long-running database transactions prevent your application from scaling to a highly-concurrent load. Do not hold a database transaction open during end-user-level work, but open it after the end-user-level work is finished. This concept is referred to as transactional write-behind. 8.2. JTA configuration Interaction with a JTA system is consolidated behind a single contract named org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform which exposes access to the javax.transaction.TransactionManager and javax.transaction.UserTransaction for that system as well as exposing the ability to register javax.transaction.Synchronization instances, check transaction status, etc. Generally, JtaPlatform will need access to JNDI to resolve the JTA TransactionManager, UserTransaction, etc. See JNDI chapter for details on configuring access to JNDI. Hibernate tries to discover the JtaPlatform it should use through the use of another service named org.hibernate.engine.transaction.jta.platform.spi.JtaPlatformResolver. If that resolution does not work, or if you wish to provide a custom implementation you will need to specify the hibernate.transaction.jta.platform setting. Hibernate provides many implementations of the JtaPlatform contract, all with short names: Atomikos JtaPlatform for Atomikos. Borland JtaPlatform for the Borland Enterprise Server. Bitronix JtaPlatform for Bitronix. JBossAS JtaPlatform for Arjuna/JBossTransactions/Narayana when used within the JBoss/WildFly Application Server. JBossTS JtaPlatform for Arjuna/JBossTransactions/Narayana when used standalone. JOnAS JtaPlatform for JOTM when used within JOnAS. JOTM JtaPlatform for JOTM when used standalone. JRun4 JtaPlatform for the JRun 4 Application Server. OC4J JtaPlatform for Oracle’s OC4J container. Orion JtaPlatform for the Orion Application Server. Resin JtaPlatform for the Resin Application Server. SapNetWeaver JtaPlatform for the SAP NetWeaver Application Server. SunOne JtaPlatform for the SunOne Application Server. Weblogic JtaPlatform for the Weblogic Application Server. WebSphere JtaPlatform for older versions of the WebSphere Application Server. WebSphereExtended JtaPlatform for newer versions of the WebSphere Application Server. 8.3. Hibernate Transaction API Hibernate provides an API for helping to isolate applications from the differences in the underlying physical transaction system in use. Based on the configured TransactionCoordinatorBuilder, Hibernate will simply do the right thing when this transaction API is used by the application. This allows your applications and components to be more portable to move around into different environments. To use this API, you would obtain the org.hibernate.Transaction from the Session. Transaction allows for all the normal operations you’d expect: begin, commit and rollback, and it even exposes some cool methods like: markRollbackOnly that works in both JTA and JDBC. getTimeout and setTimeout that again work in both JTA and JDBC. registerSynchronization that allows you to register JTA Synchronizations even in non-JTA environments. In fact, in both JTA and JDBC environments, these Synchronizations are kept locally by Hibernate. In JTA environments, Hibernate will only ever register one single Synchronization with the TransactionManager to avoid ordering problems. Additionally, it exposes a getStatus method that returns an org.hibernate.resource.transaction.spi.TransactionStatus enum. This method checks with the underlying transaction system if needed, so care should be taken to minimize its use; it can have a big performance impact in certain JTA setups. Let’s take a look at using the Transaction API in the various environments. Example 431. Using Transaction API in JDBC StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
// "jdbc" is the default, but for explicitness
.applySetting(AvailableSettings.TRANSACTION\_COORDINATOR\_STRATEGY, "jdbc")
.build();
Metadata metadata = new MetadataSources(serviceRegistry)
.addAnnotatedClass(Customer.class)
.getMetadataBuilder()
.build();
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
.build();
Session session = sessionFactory.openSession();
try {
// calls Connection#setAutoCommit(false) to
// signal start of transaction
session.getTransaction().begin();
session.createMutationQuery("UPDATE customer set NAME = 'Sir. '||NAME")
.executeUpdate();
// calls Connection#commit(), if an error
// happens we attempt a rollback
session.getTransaction().commit();
}
catch (Exception e) {
// we may need to rollback depending on
// where the exception happened
if (session.getTransaction().getStatus() == TransactionStatus.ACTIVE
|| session.getTransaction().getStatus() == TransactionStatus.MARKED\_ROLLBACK) {
session.getTransaction().rollback();
}
// handle the underlying error
}
finally {
session.close();
sessionFactory.close();
} Example 432. Using Transaction API in JTA (CMT) StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
// "jdbc" is the default, but for explicitness
.applySetting(AvailableSettings.TRANSACTION\_COORDINATOR\_STRATEGY, "jta")
.build();
Metadata metadata = new MetadataSources(serviceRegistry)
.addAnnotatedClass(Customer.class)
.getMetadataBuilder()
.build();
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
.build();
// Note: depending on the JtaPlatform used and some optional settings,
// the underlying transactions here will be controlled through either
// the JTA TransactionManager or UserTransaction
Session session = sessionFactory.openSession();
try {
// Since we are in CMT, a JTA transaction would
// already have been started. This call essentially
// no-ops
session.getTransaction().begin();
Number customerCount = (Number) session.createSelectionQuery("select count(c) from Customer c").uniqueResult();
// Since we did not start the transaction (CMT),
// we also will not end it. This call essentially
// no-ops in terms of transaction handling.
session.getTransaction().commit();
}
catch (Exception e) {
// again, the rollback call here would no-op (aside from
// marking the underlying CMT transaction for rollback only).
if (session.getTransaction().getStatus() == TransactionStatus.ACTIVE
|| session.getTransaction().getStatus() == TransactionStatus.MARKED\_ROLLBACK) {
session.getTransaction().rollback();
}
// handle the underlying error
}
finally {
session.close();
sessionFactory.close();
} Example 433. Using Transaction API in JTA (BMT) StandardServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder()
// "jdbc" is the default, but for explicitness
.applySetting(AvailableSettings.TRANSACTION\_COORDINATOR\_STRATEGY, "jta")
.build();
Metadata metadata = new MetadataSources(serviceRegistry)
.addAnnotatedClass(Customer.class)
.getMetadataBuilder()
.build();
SessionFactory sessionFactory = metadata.getSessionFactoryBuilder()
.build();
// Note: depending on the JtaPlatform used and some optional settings,
// the underlying transactions here will be controlled through either
// the JTA TransactionManager or UserTransaction
Session session = sessionFactory.openSession();
try {
// Assuming a JTA transaction is not already active,
// this call the TM/UT begin method. If a JTA
// transaction is already active, we remember that
// the Transaction associated with the Session did
// not "initiate" the JTA transaction and will later
// nop-op the commit and rollback calls...
session.getTransaction().begin();
session.persist(new Customer());
Customer customer = (Customer) session.createSelectionQuery("select c from Customer c").uniqueResult();
// calls TM/UT commit method, assuming we are initiator.
session.getTransaction().commit();
}
catch (Exception e) {
// we may need to rollback depending on
// where the exception happened
if (session.getTransaction().getStatus() == TransactionStatus.ACTIVE
|| session.getTransaction().getStatus() == TransactionStatus.MARKED\_ROLLBACK) {
// calls TM/UT commit method, assuming we are initiator;
// otherwise marks the JTA transaction for rollback only
session.getTransaction().rollback();
}
// handle the underlying error
}
finally {
session.close();
sessionFactory.close();
} In the CMT case, we really could have omitted all of the Transaction calls. But the point of the examples was to show that the Transaction API really does insulate your code from the underlying transaction mechanism. In fact, if you strip away the comments and the single configuration setting supplied at bootstrap, the code is exactly the same in all 3 examples. In other words, we could develop that code and drop it, as-is, in any of the 3 transaction environments. The Transaction API tries hard to make the experience consistent across all environments. To that end, it generally defers to the JTA specification when there are differences (for example automatically trying rollback on a failed commit). 8.4. Contextual sessions Most applications using Hibernate need some form of contextual session, where a given session is in effect throughout the scope of a given context. However, across applications the definition of what constitutes a context is typically different; different contexts define different scopes to the notion of current. Applications using Hibernate prior to version 3.0 tended to utilize either home-grown ThreadLocal-based contextual sessions, helper classes such as HibernateUtil, or utilized third-party frameworks, such as Spring or Pico, which provided proxy/interception-based contextual sessions. Starting with version 3.0.1, Hibernate added the SessionFactory.getCurrentSession() method. Initially, this assumed usage of JTA transactions, where the JTA transaction defined both the scope and context of a current session. Given the maturity of the numerous stand-alone JTA TransactionManager implementations, most, if not all, applications should be using JTA transaction management, whether or not they are deployed into a J2EE container. Based on that, the JTA-based contextual sessions are all you need to use. However, as of version 3.1, the processing behind SessionFactory.getCurrentSession() is now pluggable. To that end, a new extension interface, org.hibernate.context.spi.CurrentSessionContext, and a new configuration parameter, hibernate.current\_session\_context\_class, have been added to allow pluggability of the scope and context of defining current sessions. See the Javadocs for the org.hibernate.context.spi.CurrentSessionContext interface for a detailed discussion of its contract. It defines a single method, currentSession(), by which the implementation is responsible for tracking the current contextual session. Out-of-the-box, Hibernate comes with three implementations of this interface: org.hibernate.context.internal.JTASessionContext current sessions are tracked and scoped by a JTA transaction. The processing here is exactly the same as in the older JTA-only approach. org.hibernate.context.internal.ThreadLocalSessionContext current sessions are tracked by thread of execution. See the Javadocs for more details. org.hibernate.context.internal.ManagedSessionContext current sessions are tracked by thread of execution. However, you are responsible to bind and unbind a Session instance with static methods on this class; it does not open, flush, or close a Session. Typically, the value of this parameter would just name the implementation class to use. For the three out-of-the-box implementations, however, there are three corresponding short names: jta, thread, and managed. The first two implementations provide a one session - one database transaction programming model. This is also known and used as session-per-request. The beginning and end of a Hibernate session is defined by the duration of a database transaction. If you use programmatic transaction demarcation in plain Java SE without JTA, you are advised to use the Hibernate Transaction API to hide the underlying transaction system from your code. If you use JTA, you can utilize the JTA interfaces to demarcate transactions. If you execute in an EJB container that supports CMT, transaction boundaries are defined declaratively and you do not need any transaction or session demarcation operations in your code. The hibernate.current\_session\_context\_class configuration parameter defines which org.hibernate.context.spi.CurrentSessionContext implementation should be used. For backward compatibility, if this configuration parameter is not set but a org.hibernate.engine.transaction.jta.platform.spi.JtaPlatform is configured, Hibernate will use the org.hibernate.context.internal.JTASessionContext. 8.5. Transactional patterns (and anti-patterns) 8.5.1. Session-per-operation anti-pattern This is an anti-pattern of opening and closing a Session for each database call in a single thread. It is also an anti-pattern in terms of database transactions. Group your database calls into a planned sequence. In the same way, do not auto-commit after every SQL statement in your application. Hibernate disables or expects the application server to disable, auto-commit mode immediately. Database transactions are never optional. All communication with a database must be encapsulated by a transaction. Avoid auto-commit behavior for reading data because many small transactions are unlikely to perform better than one clearly-defined unit of work, and are more difficult to maintain and extend. Using auto-commit does not circumvent database transactions. Instead, when in auto-commit mode, JDBC drivers simply perform each call in an implicit transaction call. It is as if your application called commit after each and every JDBC call. 8.5.2. Session-per-request pattern This is the most common transaction pattern. The term request here relates to the concept of a system that reacts to a series of requests from a client/user. Web applications are a prime example of this type of system, though certainly not the only one. At the beginning of handling such a request, the application opens a Hibernate Session, starts a transaction, performs all data related work, ends the transaction and closes the Session. The crux of the pattern is the one-to-one relationship between the transaction and the Session. Within this pattern, there is a common technique of defining a current session to simplify the need of passing this Session around to all the application components that may need access to it. Hibernate provides support for this technique through the getCurrentSession method of the SessionFactory. The concept of a current session has to have a scope that defines the bounds in which the notion of current is valid. This is the purpose of the org.hibernate.context.spi.CurrentSessionContext contract. There are 2 reliable defining scopes: First is a JTA transaction because it allows a callback hook to know when it is ending, which gives Hibernate a chance to close the Session and clean up. This is represented by the org.hibernate.context.internal.JTASessionContext implementation of the org.hibernate.context.spi.CurrentSessionContext contract. Using this implementation, a Session will be opened the first time getCurrentSession is called within that transaction. Secondly is this application request cycle itself. This is best represented with the org.hibernate.context.internal.ManagedSessionContext implementation of the org.hibernate.context.spi.CurrentSessionContext contract. Here an external component is responsible for managing the lifecycle and scoping of a current session. At the start of such a scope, ManagedSessionContext#bind() method is called passing in the Session. In the end, its unbind() method is called. Some common examples of such external components include: javax.servlet.Filter implementation AOP interceptor with a pointcut on the service methods A proxy/interception container The getCurrentSession() method has one downside in a JTA environment. If you use it, after\_statement connection release mode is also used by default. Due to a limitation of the JTA specification, Hibernate cannot automatically clean up any unclosed ScrollableResults or Iterator instances returned by scroll() or iterate(). Release the underlying database cursor by calling ScrollableResults#close() or Hibernate.close(Iterator) explicitly from a finally block. 8.5.3. Conversations (application-level transactions) The session-per-request pattern is not the only valid way of designing units of work. Many business processes require a whole series of interactions with the user that are interleaved with database accesses. In web and enterprise applications, it is not acceptable for a database transaction to span a user interaction. Consider the following example: The first screen of a dialog opens. The data seen by the user is loaded in a particular Session and database transaction. The user is free to modify the objects. The user uses a UI element to save their work after five minutes of editing. The modifications are made persistent. The user also expects to have exclusive access to the data during the edit session. Even though we have multiple databases access here, from the point of view of the user, this series of steps represents a single unit of work. There are many ways to implement this in your application. A first naive implementation might keep the Session and database transaction open while the user is editing, using database-level locks to prevent other users from modifying the same data and to guarantee isolation and atomicity. This is an anti-pattern because lock contention is a bottleneck which will prevent scalability in the future. Several database transactions are used to implement the conversation. In this case, maintaining isolation of business processes becomes the partial responsibility of the application tier. A single conversation usually spans several database transactions. These multiple database accesses can only be atomic as a whole if only one of these database transactions (typically the last one) stores the updated data. All others only read data. A common way to receive this data is through a wizard-style dialog spanning several request/response cycles. Hibernate includes some features which make this easy to implement. Automatic Versioning Hibernate can perform automatic optimistic concurrency control for you. It can automatically detect (at the end of the conversation) if a concurrent modification occurred during user think time. Detached Objects If you decide to use the session-per-request pattern, all loaded instances will be in the detached state during user think time. Hibernate allows you to reattach the objects and persist the modifications. The pattern is called session-per-request-with-detached-objects. Automatic versioning is used to isolate concurrent modifications. Extended Session The Hibernate Session can be disconnected from the underlying JDBC connection after the database transaction has been committed and reconnected when a new client request occurs. This pattern is known as session-per-conversation and makes even reattachment unnecessary. Automatic versioning is used to isolate concurrent modifications, and the Session will not be allowed to flush automatically, only explicitly. Session-per-request-with-detached-objects and session-per-conversation each have advantages and disadvantages. 8.5.4. Session-per-application anti-pattern The session-per-application is also considered an anti-pattern. The Hibernate Session, like the Jakarta Persistence EntityManager, is not a thread-safe object and it is intended to be confined to a single thread at once. If the Session is shared among multiple threads, there will be race conditions as well as visibility issues, so beware of this. An exception thrown by Hibernate means you have to rollback your database transaction and close the Session immediately. If your Session is bound to the application, you have to stop the application. Rolling back the database transaction does not put your business objects back into the state they were at the start of the transaction. This means that the database state and the business objects will be out of sync. Usually, this is not a problem because exceptions are not recoverable and you will have to start over after rollback anyway. For more details, check out the exception handling section in Persistence Context chapter. The Session caches every object that is in a persistent state (watched and checked for dirty state by Hibernate). If you keep it open for a long time or simply load too much data, it will grow endlessly until you get an OutOfMemoryException. One solution is to call clear() and evict() to manage the Session cache, but you should consider a Stored Procedure if you need mass data operations. Some solutions are shown in the Batching chapter. Keeping a Session open for the duration of a user session also means a higher probability of stale data.