**How does Spring @Transactional Really Work?**

In this post we will do a deep dive into Spring transaction management. We will go over on how does @Transactional really works under the hood. Other upcoming posts will include:

* how to use features like propagation and isolation
* what are the main pitfalls and how to avoid them

**JPA and Transaction Management**

It's important to notice that JPA on itself does not provide any type of declarative transaction management. When using JPA outside of a dependency injection container, transactions need to be handled programatically by the developer:

[?](http://blog.jhades.org/how-does-spring-transactional-really-work/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | UserTransaction utx = entityManager.getTransaction();    try {      utx.begin();        businessLogic();        utx.commit();  } catch(Exception ex) {      utx.rollback();      throw ex;  } |

This way of managing transactions makes the scope of the transaction very clear in the code, but it has several disavantages:

* it's repetitive and error prone
* any error can have a very high impact
* errors are hard to debug and reproduce
* this decreases the readability of the code base
* What if this method calls another transactional method?

**Using Spring @Transactional**

With Spring @Transactional, the above code gets reduced to simply this:

[?](http://blog.jhades.org/how-does-spring-transactional-really-work/)

|  |  |
| --- | --- |
| 1  2  3  4 | @Transactional  public void businessLogic() {      ... use entity manager inside a transaction ...  } |

This is much more convenient and readable, and is currently the recommended way to handle transactions in Spring.

By using @Transactional, many important aspects such as transaction propagation are handled automatically. In this case if another transactional method is called by businessLogic(), that method will have the option of joining the ongoing transaction.

One potential downside is that this powerful mechanism hides what is going on under the hood, making it hard to debug when things don't work.

**What does @Transactional mean?**

One of the key points about @Transactional is that there are two separate concepts to consider, each with it's own scope and life cycle:

* the persistence context
* the database transaction

The transactional annotation itself defines the scope of a single database transaction. The database transaction happens inside the scope of a*persistence context*.

The persistence context is in JPA the EntityManager, implemented internally using an Hibernate Session (when using Hibernate as the persistence provider).

The persistence context is just a synchronizer object that tracks the state of a limited set of Java objects and makes sure that changes on those objects are eventually persisted back into the database.

This is a very different notion than the one of a database transaction. One Entity Manager **can be used across several database transactions**, and it actually often is.

**When does an EntityManager span multiple database transactions?**

The most frequent case is when the application is using the Open Session In View pattern to deal with lazy initialization exceptions, see this previous blog post for it's [pros and cons](http://blog.jhades.org/open-session-in-view-pattern-pros-and-cons/).

In such case the queries that run in the view layer are in separate database transactions than the one used for the business logic, but they are made via the same entity manager.

Another case is when the persistence context is marked by the developer as PersistenceContextType.EXTENDED, which means that it can survive multiple requests.

**What defines the EntityManager vs Transaction relation?**

This is actually a choice of the application developer, but the most frequent way to use the JPA Entity Manager is with the   
"Entity Manager per application transaction" pattern. This is the most common way to inject an entity manager:

[?](http://blog.jhades.org/how-does-spring-transactional-really-work/)

|  |  |
| --- | --- |
| 1  2 | @PersistenceContext  private EntityManager em; |

Here we are by default in "Entity Manager per transaction" mode. In this mode, if we use this Entity Manager inside a @Transactional method, then the method will run in a single database transaction.

**How does @PersistenceContext work?**

One question that comes to mind is, how can @PersistenceContext inject an entity manager only once at container startup time, given that entity managers are so short lived, and that there are usually multiple per request.

The answer is that it can't: EntityManager is an interface, and what gets injected in the spring bean is not the entity manager itself but *a context aware proxy* that will delegate to a concrete entity manager at runtime.

Usually the concrete class used for the proxy is   
SharedEntityManagerInvocationHandler, this can be confirmed with the help a debugger.

**How does @Transactional work then?**

The persistence context proxy that implements EntityManager is not the only component needed for making declarative transaction management work. Actually three separate components are needed:

* The EntityManager Proxy itself
* The Transactional Aspect
* The Transaction Manager

Let's go over each one and see how they interact.

**The Transactional Aspect**

The Transactional Aspect is an 'around' aspect that gets called both before and after the annotated business method. The concrete class for implementing the aspect is TransactionInterceptor.

The Transactional Aspect has two main responsibilities:

* At the 'before' moment, the aspect provides a hook point for determining if the business method about to be called should run in the scope of an ongoing database transaction, or if a new separate transaction should be started.
* At the 'after' moment, the aspect needs to decide if the transaction should be committed, rolled back or left running.

At the 'before' moment the Transactional Aspect itself does not contain any decision logic, the decision to start a new transaction if needed is delegated to the Transaction Manager.

**The Transaction Manager**

The transaction manager needs to provide an answer to two questions:

* should a new Entity Manager be created?
* should a new database transaction be started?

This needs to be decided at the moment the Transactional Aspect 'before' logic is called. The transaction manager will decide based on:

* the fact that one transaction is already ongoing or not
* the propagation attribute of the transactional method (for example REQUIRES\_NEW always starts a new transaction)

If the transaction manager decides to create a new transaction, then it will:

* create a new entity manager
* bind the entity manager to the current thread
* grab a connection from the DB connection pool
* bind the connection to the current thread

The entity manager and the connection are both bound to the current thread using [ThreadLocal](http://docs.oracle.com/javase/6/docs/api/java/lang/ThreadLocal.html) variables.

They are stored in the thread while the transaction is running, and it's up to the Transaction Manager to clean them up when no longer needed.

Any parts of the program that need the current entity manager or connection can retrieve them from the thread. One program component that does exactly that is the EntityManager proxy.

**The EntityManager proxy**

The EntityManager proxy (that we have introduced before) is the last piece of the puzzle. When the business method calls for example   
entityManager.persist(), this call is not invoking the entity manager directly.

Instead the business method calls the proxy, which retrieves the current entity manager from the thread, where the Transaction Manager put it.

Knowing now what are the moving parts of the @Transactionalmechanism, let's go over the usual Spring configuration needed to make this work.

**Putting It All Together**

Let's go over how to setup the three components needed to make the transactional annotation work correctly. We start by defining the entity manager factory.

This will allow the injection of Entity Manager proxies via the persistence context annotation:

[?](http://blog.jhades.org/how-does-spring-transactional-really-work/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | @Configuration  public class EntityManagerFactoriesConfiguration {      @Autowired      private DataSource dataSource;        @Bean(name = "entityManagerFactory")      public LocalContainerEntityManagerFactoryBean emf() {          LocalContainerEntityManagerFactoryBean emf = ...          emf.setDataSource(dataSource);          emf.setPackagesToScan(              new String[] {"your.package"});          emf.setJpaVendorAdapter(              new HibernateJpaVendorAdapter());          return emf;      }  } |

The next step is to configure the Transaction Manager and to apply the Transactional Aspect in @Transactional annotated classes:

[?](http://blog.jhades.org/how-does-spring-transactional-really-work/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | @Configuration  @EnableTransactionManagement  public class TransactionManagersConfig {      @Autowired      EntityManagerFactory emf;      @Autowired      private DataSource dataSource;        @Bean(name = "transactionManager")      public PlatformTransactionManager transactionManager() {          JpaTransactionManager tm =              new JpaTransactionManager();              tm.setEntityManagerFactory(emf);              tm.setDataSource(dataSource);          return tm;      }  } |

The annotation @EnableTransactionManagement tells Spring that classes with the @Transactional annotation should be wrapped with the Transactional Aspect. With this the @Transactional is now ready to be used.

**Conclusion**

The Spring declarative transaction management mechanism is very powerful, but it can be misused or wrongly configured easily.

Understanding how it works internally is helpful when troubleshooting situations when the mechanism is not at all working or is working in an unexpected way.

The most important thing to bear in mind is that there are really two concepts to take into account: the database transaction and the persistence context, each with it's own not readily apparent life cycle.

A future post will go over the most frequent pitfalls of the transactional annotation and how to avoid them.

# Performance Tuning of Spring/Hibernate Applications

For most typical Spring/Hibernate enterprise applications, the application performance depends almost entirely on the performance of it's persistence layer.

This post will go over how to confirm that we are in presence of a 'database-bound' application, and then walk through 7 frequently used 'quick-win' tips that can help improve application performance.

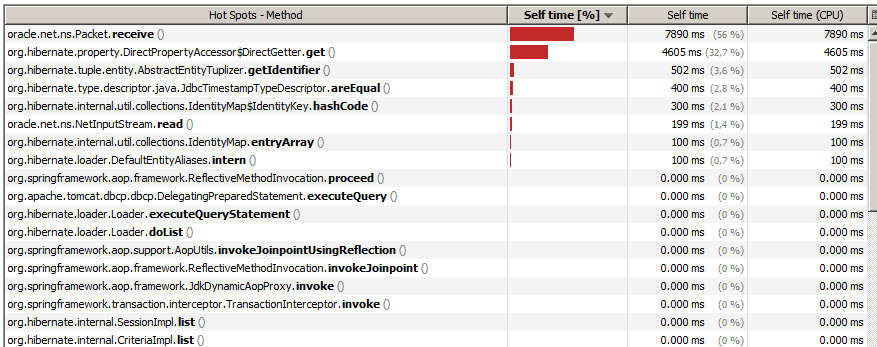
#### How to confirm that an application is 'database-bound'

To confirm that an application is 'database-bound', start by doing a typical run in some development environment, using [VisualVM](http://visualvm.java.net/) for monitoring. VisualVM is a Java profiler shipped with the JDK and launchable via the command line by calling jvisualvm.

After launching Visual VM, try the following steps:

* double click on your running application
* Select Sampler
* click on Settings checkbox
* Choose Profile only packages, and type in the following packages:
  + your.application.packages.\*
  + org.hibernate.\*
  + org.springframework.\*
  + your.database.driver.package, for example oracle.\*
  + Click Sample CPU

The CPU profiling of a typical 'database-bound' application should look something like this:



We can see that the client Java process spends 56% of it's time waiting for the database to return results over the network.

This is a good sign that the queries on the database are what's keeping the application slow. The 32.7% in Hibernate reflection calls is normal and nothing much can be done about it.

#### First step for tuning - obtaining a baseline run

The first step to do tuning is to define a baseline run for the program. We need to identify a set of functionally valid input data that makes the program go through a typical execution similar to the production run.

The main difference is that the baseline run should run in a much shorter period of time, as a guideline an execution time of around 5 to 10 minutes is a good target.

#### What makes a good baseline?

A good baseline should have the following characteristics:

* it's functionally correct
* the input data is similar to production in it's variety
* it completes in a short amount of time
* optimizations in the baseline run can be extrapolated to a full run

*Getting a good baseline is solving half of the problem.*

#### What makes a bad baseline?

For example, in a batch run for processing call data records in a telecommunications system, taking the first 10 000 records could be the**wrong** approach.

The reason being, the first 10 000 might be mostly voice calls, but the unknown performance problem is in the processing of SMS traffic. Taking the first records of a large run would lead us to a bad baseline, from which wrong conclusions would be taken.

#### Collecting SQL logs and query timings

The SQL queries executed with their execution time can be collected using for example [log4jdbc](https://code.google.com/p/log4jdbc/). See this blog post for how to collect SQL queries using log4jdbc - [Spring/Hibernate improved SQL logging with log4jdbc](http://blog.jhades.org/logging-the-actualreal-sql-queries-of-a-springhibernate-application/).

The query execution time is measured from the Java client side, and it includes the network round-trip to the database. The SQL query logs look like this:

16 avr. 2014 11:13:48 | SQL\_QUERY */\* insert your.package.YourEntity \*/* **insert** **into** YOUR\_TABLE (...) **values** (...) {executed **in** 13 msec}

The prepared statements themselves are also a good source of information - they allow to easily identify frequent query types. They can be logged by following this blog post - [Why and where is Hibernate doing this SQL query?](http://blog.jhades.org/how-to-find-out-why-hibernate-is-doing-a-certain-sql-query/)

#### What metrics can be extracted from SQL logs

The SQL logs can give the answer these questions:

* What are slowest queries being executed?
* What are the most frequent queries?
* What is the amount of time spent generating primary keys?
* Is there some data that could benefit from caching ?

#### How to parse the SQL logs

Probably the only viable option for large log volumes is to use command line tools. This approach has the advantage of being very flexible.

At the expense of writing a small script or command, we can extract mostly any metric needed. Any command line tool will work as long as you are comfortable with it.

If you are used to the Unix command line, bash might be a good option. Bash can be used also in Windows workstations, using for example[Cygwin](http://www.cygwin.com/), or [Git](http://git-scm.com/downloads) that includes a bash command line.

#### Frequently applied Quick-Wins

The quick-wins bellow identify common performance problems in Spring/Hibernate applications, and their corresponding solutions.

#### Quick-win Tip 1 - Reduce primary key generation overhead

In processes that are 'insert-intensive', the choice of a primary key generation strategy can matter a lot. One common way to generate id's is to use database sequences, usually one per table to avoid contention between inserts on different tables.

The problem is that if 50 records are inserted, we want to avoid that 50 network round-trips are made to the database in order to obtain 50 id's, leaving the Java process hanging most of the time.

##### How does Hibernate usually handle this?

Hibernate provides new optimized ID generators that avoid this problem. Namely for sequences, a HiLo id generator is used by default. This is how the HiLo sequence generator it works:

* call a sequence once and get 1000 (the High value)
* calculate 50 id's like this:
  + 1000 \* 50 + 0 = 50000
  + 1000 \* 50 + 1 = 50001
  + ...
  + 1000 \* 50 + 49 = 50049, Low value (50) reached
  + call sequence for new High value 1001 ... etc ...

So from a single sequence call, 50 keys where generated, reducing the overhead caused my inumerous network round-trips.

These new optimized key generators are on by default in Hibernate 4, and can even be turned off if needed by setting hibernate.id.new\_generator\_mappings to false.

##### Why can primary key generation still be a problem?

The problem is, if you declared the key generation strategy as AUTO, the optimized generators are **still** off, and your application will end up with a huge amount of sequence calls.

In order to make sure the new optimized generators are on, make sure to use the SEQUENCE strategy instead of AUTO:

@Id

@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "your\_key\_generator")

**private** Long id;

With this simple change, an improvement in the range of 10%-20% can be measured in 'insert-intensive' applications, with basically no code changes.

#### Quick-win Tip 2 - Use JDBC batch inserts/updates

For batch programs, JDBC drivers usually provide an optimization for reducing network round-trips named 'JDBC batch inserts/updates'. When these are used, inserts/updates are queued at the driver level before being sent to the database.

When a threshold is reached, then the whole batch of queued statements is sent to the database in one go. This prevents the driver from sending the statements one by one, which would waist multiple network round-trips.

This is the entity manager factory configuration needed to active batch inserts/updates:

<prop key="hibernate.jdbc.batch\_size">100</prop>

<prop key="hibernate.order\_inserts">true</prop>

<prop key="hibernate.order\_updates">true</prop>

Setting only the JDBC batch size **won't** work. This is because the JDBC driver will batch the inserts only when receiving insert/updates for the exact same table.

If an insert to a new table is received, then the JDBC driver will first flush the batched statements on the previous table, before starting to batch statements on the new table.

A similar functionality is implicitly used if using Spring Batch. This optimization can easily buy you 30% to 40% to 'insert intensive' programs, without changing a single line of code.

#### Quick-win Tip 3 - Periodically flush and clear the Hibernate session

When adding/modifying data in the database, Hibernate keeps in the session a version of the entities already persisted, just in case they are modified again before the session is closed.

But many times we can safely discard entities once the corresponding inserts where done in the database. This releases memory in the Java client process, preventing performance problems caused by long running Hibernate sessions.

Such long-running sessions **should** be avoided as much as possible, but if by some reason they are needed, this is how to contain memory consumption:

entityManager.flush();

entityManager.clear();

The flush will trigger the inserts from new entities to be sent to the database. The clear releases the new entities from the session.

#### Quick-win Tip 4 - Reduce Hibernate dirty-checking overhead

Hibernate uses internally a mechanism to keep track of modified entities called dirty-checking. This mechanism is **not** based on the equals and hashcode methods of the entity classes.

Hibernate does it's most to keep the performance cost of dirty-checking to a minimum, and to dirty-check only when it needs to, but the mechanism does have a cost, which is more noticeable in tables with a large number of columns.

Before applying any optimization, the most important is to measure the cost of dirty-checking using VisualVM.

##### How to avoid dirty-checking?

In Spring business methods that we know are read-only, dirty-checking can be turned off like this:

@Transactional(readOnly=**true**)

**public** **void** **someBusinessMethod**() {

....

}

An alternative to avoid dirty-checking is to use the Hibernate Stateless Session, which is detailed in the [documentation](http://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch04.html#d0e1932).

#### Quick-win Tip 5 - Search for 'bad' query plans

Check the queries in the slowest queries list to see if they have good query plans. The most usual 'bad' query plans are:

* Full table scans: they happen when the table is being fully scanned due to usually a missing index or outdated table statistics.
* Full cartesian joins: This means that the full cartesian product of several tables is being computed. Check for missing join conditions, or if this can be avoided by splitting a step into several.

#### Quick-win Tip 6 - check for wrong commit intervals

If you are doing batch processing, the commit interval can make a large difference in the performance results, as in 10 to 100 times faster.

Confirm that the commit interval is the one expected (usually around 100-1000 for Spring Batch jobs). It happens often that this parameter is not correctly configured.

#### Quick-win Tip 7 - Use the second-level and query caches

If some data is identified as being eligible for caching, then have a look at this blog post for how to setup the Hibernate caching: [Pitfalls of the Hibernate Second-Level / Query Caches](http://blog.jhades.org/setup-and-gotchas-of-the-hibernate-second-level-and-query-caches/)

#### Conclusions

To solve application performance problems, the most important action to take is to collect some metrics that allow to find what the current bottleneck is.

Without some metrics it is often not possible to guess in useful time what the correct problem cause is.

Also, many but not all of the typical performance pitfalls of a 'database-driven' application can be avoided in the first place by using the Spring Batch framework.

**Open Session In View Design Tradeoffs**

The Open Session in View (OSIV) pattern gives rise to different opinions in the Java development community. Let's go over OSIV and some of the pros and cons of this pattern.

**The problem**

The problem that OSIV solves is a mismatch between the Hibernate concept of session and it's lifecycle and the way that many server-side view technologies work.

In a typical Java frontend application the service layer starts by querying some of the data needed to build the view. The remaining data needed can be lazy-loaded later, with the condition that the Hibernate session remains open - and there lies the problem.

Between the moment that the service layer method finishes it's execution and the moment that the view is rendered, Hibernate has already committed the transaction and closed the session.

When the view tries to lazy load the extra data that it needs, if finds the Hibernate session closed, causing a LazyInitializationException.

**The OSIV solution**

OSIV tackles this problem by ensuring that the Hibernate session is kept open all the way up to the rendering of the view - hence the name of the pattern.

Because the session is kept open, no more LazyInitializationExceptions occur. The session or entity manager is kept open by means of a filter that is added to the request processing chain.

In the case of JPA the OpenEntityManagerInViewFilter will create an entity manager at the beginning of the request, and then bind it to the request thread.

The service layer will then be executed and the business transaction committed or rolled back, but the transaction manager will not remove the entity manager from the thread after the commit.

When the view rendering starts, the transaction manager will then check if there is already an entity manager binded to the thread, and if so use it instead of creating a new one.

After the request is processed, the filter will then unbind the entity manager from the thread.

The end result is that the same entity manager used to commit the business transaction was kept around in the request thread, allowing the view rendering code to lazy load the needed data.

**Going back to the original problem**

Let's step back a moment and go back to the initial problem: the LazyInitializationException. Is this exception really a problem? This exception can also be seen as a warning sign of a wrongly written query in the service layer.

When building a view and it's backing services, the developer knows upfront what data is needed, and can make sure that the needed data is loaded before the rendering starts.

Several relation types such as one-to-many use lazy-loading by default, but that default setting can be overridden if needed *at query time* using the following syntax:

**select** p **FROM** Person p **left** **join** **fetch** p.invoices

This means that the lazy loading can be turned off on a case by case basis depending on the data needed by the view.

**OSIV in projects I've worked**

In projects I have worked that used OSIV, we could see via query logging that the database was getting hit with a high number of SQL queries, sometimes to the point that developers had to turn off the Hibernate SQL logging.

The performance of these application was impacted, but it was kept manageable using second-level caches, and due to the fact that these where intranet-based applications with a limited number of users.

**Pros of OSIV**

The main advantage of OSIV is that it makes working with ORM and the database more transparent:

* Less queries need to be manually written
* Less awareness is required about the Hibernate session and how to solve LazyInitializationExceptions.

**Cons of OSIV**

OSIV seems to be easy to misuse and can accidentally introduce N+1 performance problems in the application. On projects I've worked OSIV did not work out well in the long-term.

The alternative of writing custom queries that eager fetch data depending on the use case is manageable and turned out well in other projects I've worked.

**Alternatives to OSIV**

Besides the application-level solution of writing custom queries to pre-fetch the needed data, there are other framework-level aproaches to OSIV.

The [Seam Framework](http://blog.jhades.org/open-session-in-view-pattern-pros-and-cons/www.seamframework.org/%E2%80%8E) was built by some of the same developers as Hibernate , and solves the problem by introducing the notion of[conversation](http://docs.jboss.org/seam/2.1.0.SP1/reference/en-US/html/conversations.html#d0e5740).

# Spring/Hibernate improved SQL logging with log4jdbc

Hibernate provides SQL logging out of the box, but such logging only shows prepared statements, and not the actual SQL queries sent to the database.

It also does not log the execution time of each query, which is useful for performance troubleshooting. This blog post will go over how to setup Hibernate query logging, and then compare it to the logging that can be obtained with [log4jdbc](https://code.google.com/p/log4jdbc-remix/).

#### The Hibernate query logging functionality

Hibernate does not log the real SQL queries sent to the database. This is because Hibernate interacts with the database via the JDBC driver, to which it sends prepared statements but not the actual queries.

So Hibernate can only log the prepared statements and the values of their binding parameters, but not the actual SQL queries themselves.

This is how a query looks like when logged by Hibernate:

**select** /\* **load** your.package.Employee \*/ this\_.code, ...

**from** employee this\_

**where** this\_.employee\_id=?

TRACE 12-04-2014@16:06:02 BasicBinder - binding parameter [1] **as** [**NUMBER**] - 1000

See this post [Why and where is Hibernate doing this SQL query?](http://blog.jhades.org/how-to-find-out-why-hibernate-is-doing-a-certain-sql-query) for how to setup this type of logging.

### Using log4jdbc

For a developer it's useful to be able to copy paste a query from the log and be able to execute the query directly in an SQL client, but the variable placeholders ? make that unfeasible.

Log4jdbc in an open source tool that allows to do just that, and more. Log4jdbc is a spy driver that will wrap itself around the real JDBC driver, logging queries as they go through it.

The version linked from this post provides Spring integration, unlike several other log4jdbc forks.

#### Setting up log4jdbc

First include the log4jdbc-remix library in your pom.xml. This library is a fork of the original log4jdbc:

<dependency>

<groupId>**org**.lazyluke</groupId>

<artifactId>**log4jdbc**-remix</artifactId

<version>**0**.2.7</version>

</dependency>

Next, find in the Spring configuration the definition of the data source. As an example, when using the JNDI lookup element this is how the data source looks like:

<jee:jndi-lookup id="dataSource"

jndi-name="java:comp/env/jdbc/some-db" />

After finding the data source definition, rename it to the following name:

<jee:jndi-lookup id="dataSourceSpied"

jndi-name="java:comp/env/jdbc/some-db" />

Then define a new log4jdbc data source that wraps the real data source, and give it the original name:

<bean id="dataSource" class="net.sf.log4jdbc.Log4jdbcProxyDataSource" >

<constructor-arg ref="dataSourceSpied" />

<property name="logFormatter">

<bean class="net.sf.log4jdbc.tools.Log4JdbcCustomFormatter" >

<property name="loggingType" value="SINGLE\_LINE" />

<property name="margin" value="19" />

<property name="sqlPrefix" value="SQL:::" />

</bean>

</property>

</bean >

With this configuration, the query logging should already be working. It's possible to customize the logging level of the several log4jdbc loggers available.

The original [log4jdbc](https://code.google.com/p/log4jdbc/) documentation provides more information on the available loggers:

* jdbc.sqlonly: Logs only SQL
* jdbc.sqltiming: Logs the SQL, post-execution, including timing execution statistics
* jdbc.audit: Logs ALL JDBC calls except for ResultSets
* jdbc.resultset: all calls to ResultSet objects are logged
* jdbc.connection: Logs connection open and close events

The jdbc.audit logger is especially useful to validate the scope of transactions, as it logs the begin/commit/rollback events of a database transaction.

This is the proposed log4j configuration that will print only the SQL queries together with their execution time:

<logger name="jdbc.sqltiming" additivity ="false">

<level value="info" />

</logger>

<logger name="jdbc.resultset" additivity ="false">

<level value="error" />

</logger>

<logger name="jdbc.audit" additivity ="false">

<level value="error" />

</logger>

<logger name="jdbc.sqlonly" additivity ="false">

<level value="error" />

</logger>

<logger name="jdbc.resultsettable" additivity ="false">

<level value="error" />

</logger>

<logger name="jdbc.connection" additivity ="false">

<level value="error" />

</logger>

<logger name="jdbc.resultsettable" additivity ="false">

<level value="error" />

</logger>

#### Conclusion

Using log4jdbc does imply some initial setup, but once it's in place it's really convenient to have. Having a true query log is also useful for performance troubleshooting, as will be described in an upcoming post.

**Hibernate Debugging - Finding the origin of a Query**

It's not always immediate why and in which part of the program is Hibernate generating a given SQL query, especially if we are dealing with code that we did not write ourselves.

This post will go over how to configure Hibernate query logging, and use that together with other tricks to find out why and where in the program a given query is being executed.

**What does the Hibernate query log look like**

Hibernate has built-in query logging that looks like this:

**select** /\* **load** your.package.Employee \*/ this\_.code, ...

**from** employee this\_

**where** this\_.employee\_id=?

TRACE 12-04-2014@16:06:02 BasicBinder - binding parameter [1] **as** [**NUMBER**] - 1000

**Why can't Hibernate log the actual query ?**

Notice that what is logged by Hibernate is the prepared statement sent by Hibernate to the JDBC driver plus it's parameters. The prepared statement has ? in the place of the query parameters, the parameter values themselves are logged just bellow the prepared statement.

This is not the same as the actual query sent to the database, as there is no way for Hibernate to log the actual query. The reason for this is that Hibernate only knows about the prepared statements and the parameters that it sends to the JDBC driver, and it's the driver that will build the actual queries and then send them to the database.

In order to produce a log with the real queries, a tool like [log4jdbc](https://code.google.com/p/log4jdbc-remix/) is needed, which will be the subject of another post.

**How to find out the origin of the query**

The logged query above contains a comment that allows to identify in most cases the origin of the query: if the query is due to a load by ID the comment is /\* load your.entity.Name \*/, if it's a named query then the comment will contain the name of the query.

If it's a one to many lazy initialization the comment will contain the name of the class and the property that triggered it, etc.

**Setting up the Hibernate query log**

In order to obtain a query log, the following flags need to be set in the configuration of the session factory:

<bean id= "entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean" >

...

<property name="jpaProperties" >

<props>

<prop key="hibernate.show\_sql" >true</ prop>

<prop key="hibernate.format\_sql" >true</ prop>

<prop key="hibernate.use\_sql\_comments">true</prop>

</props>

</property>

The example above is for Spring configuration of an entity manager factory. This is the meaning of the flags:

* show\_sql enables query logging
* format\_sql pretty prints the SQL
* use\_sql\_comments adds an explanatory comment

In order to log the query parameters, the following log4j or equivalent information is needed:

<logger name="org.hibernate.type">

<level value="trace" />

</logger >

**If everything else fails**

In many cases the comment created by use\_sql\_comments is enough to identify the origin of the query. If this is not sufficient, then we can start by identifying the entity returned by the query based on the table names involved, and put a breakpoint in the constructor of the returned entity.

If the entity does not have a constructor, then we can create one and put the breakpoint in the call to super():

@Entity

**public** **class** **Employee** {

**public** **Employee**() {

**super**(); *// put the breakpoint here*

}

...

}

When the breakpoint is hit, go to the IDE debug view containing the stack call of the program and go through it from top to bottom. The place where the query was made in the program will be there in the call stack.

# Pitfalls of the Hibernate Second-Level / Query Caches

This post will go through how to setup the Hibernate Second-Level and Query caches, how they work and what are their most common pitfalls.

The Hibernate second level cache is an application level cache for storing entity data. The query cache is a separate cache that stores query results only.

The two caches really go together, as there are not many cases where we would like to use one without the other. When well used these caches provide improved performance in a transparent way, by reducing the number of SQL statements that hit the database.

#### How does the second level-cache work?

The second level cache stores the entity data, but **NOT** the entities themselves. The data is stored in a 'dehydrated' format which looks like a hash map where the key is the entity Id, and the value is a list of primitive values.

Here is an example on how the contents of the second-level cache look:

\*-----------------------------------------\*

| Person Data Cache |

|-----------------------------------------|

| 1 -> [ "John" , "Q" , "Public" , null ] |

| 2 -> [ "Joey" , "D" , "Public" , 1 ] |

| 3 -> [ "Sara" , "N" , "Public" , 1 ] |

\*-----------------------------------------\*

The second level cache gets populated when an object is loaded by Id from the database, using for example entityManager.find(), or when traversing lazy initialized relations.

#### How does the query cache work?

The query cache looks conceptually like an hash map where the key is composed by the query text and the parameter values, and the value is a list of entity Id's that match the query:

\*----------------------------------------------------------\*

| Query Cache |

|----------------------------------------------------------|

| ["from Person where firstName=?", ["Joey"] ] -> [1, 2] ] |

\*----------------------------------------------------------\*

Some queries don't return entities, instead they return only primitive values. In those cases the values themselves will be stored in the query cache. The query cache gets populated when a cacheable JPQL/HQL query gets executed.

#### What is the relation between the two caches?

If a query under execution has previously cached results, then no SQL statement is sent to the database. Instead the query results are retrieved from the query cache, and then the cached entity identifiers are used to access the second level cache.

If the second level cache contains data for a given Id, it re-hydrates the entity and returns it. If the second level cache does not contain the results for that particular Id, then an SQL query is issued to load the entity from the database.

#### How to setup the two caches in an application

The first step is to include the hibernate-ehcache jar in the classpath:

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>SOME-HIBERNATE-VERSION</version>

</dependency>

The following parameters need to be added to the configuration of your EntityManagerFactory or SessionFactory:

<prop key="hibernate.cache.use\_second\_level\_cache">true</prop>

<prop key="hibernate.cache.use\_query\_cache">true</prop>

<prop key="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</prop>

<prop key="net.sf.ehcache.configurationResourceName">/your-cache-config.xml</prop>

Prefer using EhCacheRegionFactory instead of SingletonEhCacheRegionFactory. Using EhCacheRegionFactory means that Hibernate will create separate cache regions for Hibernate caching, instead of trying to reuse cache regions defined elsewhere in the application.

The next step is to configure the cache regions settings, in file your-cache-config.xml:

**<?xml version="1.0" ?>**

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

updateCheck="false"

xsi:noNamespaceSchemaLocation="ehcache.xsd" name="yourCacheManager">

<diskStore path="java.io.tmpdir"/>

<cache name="yourEntityCache"

maxEntriesLocalHeap="10000"

eternal="false"

overflowToDisk="false"

timeToLiveSeconds="86400" />

<cache name="org.hibernate.cache.internal.StandardQueryCache"

maxElementsInMemory="10000"

eternal="false

timeToLiveSeconds="86400"

overflowToDisk="false"

memoryStoreEvictionPolicy="LRU" />

<defaultCache

maxElementsInMemory="10000"

eternal="false"

timeToLiveSeconds="86400"

overflowToDisk="false"

memoryStoreEvictionPolicy="LRU" />

</ehcache>

If no cache settings are specified, default settings are taken, but this is probably best avoided. Make sure to give the cache a name by filling in the name attribute in the ehcache element.

Giving the cache a name prevents it from using the default name, which might already be used somewhere else on the application.

#### Using the second level cache

The second level cache is now ready to be used. In order to cache entities, annotate them with the @org.hibernate.annotations.Cacheannotation:

@Entity

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY,

region="yourEntityCache")

**public** **class** **SomeEntity** {

...

}

Associations can also be cached by the second level cache, but by default this is not done. In order to enable caching of an association, we need to apply @Cache to the association itself:

@Entity

**public** **class** **SomeEntity** {

@OneToMany

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY,

region="yourCollectionRegion")

**private** Set<OtherEntity> other;

}

#### Using the query cache

After configuring the query cache, by default no queries are cached yet. Queries need to be marked as cached explicitly, this is for example how a named query can be marked as cached:

@NamedQuery(name="account.queryName",

query="select acct from Account ...",

hints={

@QueryHint(name="org.hibernate.cacheable",

value="true")

}

})

And this is how to mark a criteria query as cached:

**List** cats = session.createCriteria(Cat.class)

.setCacheable(**true**)

.**list**();

The next section goes over some pitfalls that you might run into while trying to setup these two caches. These are behaviors that work as designed but still can be surprising.

#### Pitfall 1 - Query cache worsens performance causing a high volume of queries

There is an harmful side-effect of how the two caches work, that occurs if the cached query results are configured to expire more frequently than the cached entities returned by the query.

If a query has cached results, it returns a list of entity Id's, that is then resolved against the second level cache. If the entities with those Ids where not configured as cacheable or if they have expired, then a select will hit the database per entity Id.

For example if a cached query returned 1000 entity Ids, and non of those entities where cached in the second level cache, then 1000 selects by Id will be issued against the database.

The solution to this problem is to configure query results expiration to be aligned with the expiration of the entities returned by the query.

#### Pitfall 2 - Cache limitations when used in conjunction with @Inheritance

It is currently not possible to specify different caching policies for different subclasses of the same parent entity.

For example this will not work:

@Entity

@Inheritance

@Cache(CacheConcurrencyStrategy.READ\_ONLY)

**public** **class** **BaseEntity** {

...

}

@Entity

@Cache(CacheConcurrencyStrategy.READ\_WRITE)

**public** **class** **SomeReadWriteEntity** **extends** **BaseEntity** {

...

}

@Entity

@Cache(CacheConcurrencyStrategy.TRANSACTIONAL)

**public** **class** **SomeTransactionalEntity** **extends** **BaseEntity** {

...

}

In this case only the @Cache annotation of the parent class is considered, and all concrete entities have READ\_ONLY concurrency strategy.

#### Pitfall 3 - Cache settings get ignored when using a singleton based cache

It is advised to configure the cache region factory as a EhCacheRegionFactory, and specify an ehcache configuration via net.sf.ehcache.configurationResourceName.

There is an alternative to this region factory which is SingletonEhCacheRegionFactory. With this region factory the cache regions are stored in a singleton using the cache name as a lookup key.

The problem with the singleton region factory is that if another part of the application had already registered a cache with the default name in the singleton, this causes the ehcache configuration file passed via net.sf.ehcache.configurationResourceName to be ignored.

#### Conclusion

The second level and query caches are very useful if set up correctly, but there are some pitfalls to bear in mind in order to avoid unexpected behaviors. All in all it's a feature that works transparently and that if well used can increase significantly the performance of an application.

Please let us know in the comments bellow your own experience and pitfalls you have encountered. Thanks for reading.

#### Useful Links

This blog post is a well-known reference to the inner details of the Hibernate second level and query caches - [Truly Understanding the Second-Level and Query Caches](http://www.javalobby.org/java/forums/t48846.html)

Spring Transaction Attributes

**What are transaction attributes?**

Spring transactions allow setting up the propagation behavior, isolation, timeout and read only settings of a transaction. Before we delve into the details, here are some points that need to be kept in mind

* Isolation level and timeout settings get applied only after the transaction starts.
* Not all transaction managers specify all values and may throw exception with some non default values

**Propagation**

PROPAGATION\_REQUIRED   
This attribute tells that the code needs to be run in a transactional context. If a transaction already exists then the code will use it otherwise a new transaction is created. This is the default and mostly widely used transaction setting.

PROPAGATION\_SUPPORTS   
If a transaction exists then the code will use it, but the code does not require a new one. As an example, consider a ticket reservation system. A query to get total seats available can be executed non-transactionally. However, if used within a transaction context it will deduct tickets already selected and reduce them from the total count, and hence may give a better picture. This attribute should be used with care especially when PROPAGATION\_REQUIRED or PROPAGATION\_REQUIRES\_NEW is used within a PROPAGATION\_SUPPORTS context.

PROPAGATION\_MANDATORY   
Participates in an existing transaction, however if no transaction context is present then it throws a TransactionRequiredException

PROPAGATION\_REQUIRES\_NEW   
Creates a new transaction and if an existing transaction is present then it is suspended. In other words a new transaction is always started. When the new transaction is complete then the original transaction resumes. This transaction type is useful when a sub activity needs to be completed irrespective of the containing transaction. The best example of this is logging. Even if a transaction roll backs you still want to preserve the log statements. Transaction suspension may not work out of the box with all transaction managers, so make sure that the transaction manager supports transaction suspension

PROPAGATION\_NOT\_SUPPORTED   
This attribute says that transaction is not supported. In other words the activity needs to be performed non-transactionally. If an existing transaction is present then it is suspended till the activity finishes.

PROPAGATION\_NEVER   
This attributes says that the code cannot be invoked within a transaction. However, unlike PROPAGATION\_NOT\_SUPPORTED, if an existing transaction is present then an exception will be thrown

PROPAGATION\_NESTED   
The code is executed within a nested transaction if existing transaction is present, if no transaction is present then a new transaction is created. Nested transaction is supported out of the box on only certain transaction managers.

**Isolation**

Isolation is a property of a transaction that determines what effect a transaction has on other concurrent transactions. To completely isolate the transaction the database may apply locks to rows or tables. Before we go through the transaction levels, let us look at some problems that occur when transaction 1 reads data that is being modified by transaction 2.

* *Dirty Reads*- Dirty reads occur when transaction 2 reads data that has been modified by transaction 1 but not committed. The problem occurs when transaction 1 rollbacks the transaction, in which case the data read by transaction 2 will be invalid.
* *Non Repeatable Reads*- Nonrepeatable reads happen when a transaction fires the same query multiple times but receives different data each time for the same query. This may happen when another transaction has modified the rows while this query is in progress.
* *Phantom Reads* - Phantom reads occur when the collection of rows returned is different when a same query is executed multiple times in a transaction. Phantom reads occur when transaction 2 adds rows to a table between the multiple queries of transaction 1.

The following isolation levels are supported by spring

ISOLATION\_DEFAULT   
Use the isolation level of the underlying database.

ISOLATION\_READ\_UNCOMMITTED   
This is the lowest level of isolation and says that a transaction is allowed to read rows that have been added but not committed by another transaction. This level allows dirty reads, phantom reads and non repeatable reads.

ISOLATION\_READ\_COMMITTED   
This level allows multiple transactions on the same data but does not allow uncommited transaction of one transaction to be read by another. This level, therefore, prevents dirty reads but allows phantom reads and nonrepeatable reads. This is the default isolation setting for most database and is supported by most databases.

ISOLATION\_REPEATABLE\_READ   
This level ensures that the data set read during a transaction remains constant even if another transaction modifies and commits changes to the data. Therefore if transaction 1 reads 4 rows of data and transaction 2 modifies and commits the fourth row and then transaction 1 reads the four rows again then it does not see the modifications made by transaction 2. (It does not see the changes made in the fourth row by the second transaction). This level prevents dirty reads and non repeatable reads but allows phantom reads.

ISOLATION\_SERIALIZABLE   
This is the highest isolation level. It prevents dirty reads, non repeatable reads and phantom reads. This level prevents the situation when transaction 1 performs a query with a certain where clause and retrieves say four rows, transaction 2 inserts a row that forms part of the same where clause and then transaction 1 reruns the query with the same where clause but still sees only four rows (does not see the row added by the second transaction)

**Read Only**

The read only attribute specifies that the transaction is only going to read data from a database. The advantage is that the database may apply certain optimization to the transaction when it is declared to be read only. Since read only attribute comes in action as soon as the transaction starts, it may be applied to only those propagation settings that start a transaction. i.e. PROPAGATION\_REQUIRED,PROPAGATION\_REQUIRES\_NEW and PROPAGATION\_NESTED.

**Timeout**

Timeout specifies the maximum time allowed for a transaction to run. This may be required since transactions that run for a very long time may unnecessarily hold locks for a long time. When a transaction reaches the timeout period, it is rolled back. Timeout needs to be specified only on propagation settings that start a new transaction

**Rollback Rules**

It is also possible to specify that transactions roll back on certain exceptions and do not rollback on other exceptions by specifying the rollback rules.

# Spring transaction isolation level tutorial

30 January 2013

By [Gonçalo Marques](https://plus.google.com/107771575694787223844?rel=author)

[java](http://www.byteslounge.com/tag/java)[spring](http://www.byteslounge.com/tag/spring)[tx](http://www.byteslounge.com/tag/tx)

In this tutorial you will learn about the transaction isolation level provided by the Spring framework.

## Introduction

Transaction isolation level is a concept that is not exclusive to the Spring framework. It is applied to transactions in general and is directly related with the ACID transaction properties. Isolation level defines how the changes made to some data repository by one transaction affect other simultaneous concurrent transactions, and also how and when that changed data becomes available to other transactions. When we define a transaction using the Spring framework we are also able to configure in which isolation level that same transaction will be executed.

## Usage example

Using the **@Transactional** annotation we can define the isolation level of a Spring managed bean transactional method. This means that the transaction in which this method is executed will run with that isolation level:

Isolation level in a transactional method

@Autowired

private TestDAO testDAO;

@Transactional(isolation=Isolation.READ\_COMMITTED)

public void someTransactionalMethod(User user) {

// Interact with testDAO

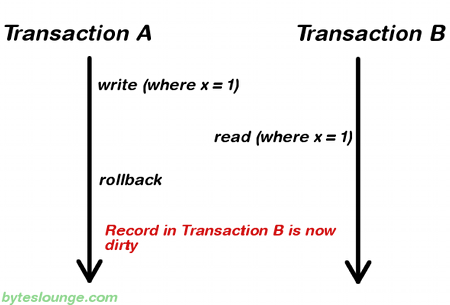
}

We are defining this method to be executed in a transaction which isolation level is **READ\_COMMITTED**. We will see each isolation level in detail in the next sections.

## READ\_UNCOMMITTED

**READ\_UNCOMMITTED** isolation level states that a transaction **may** read data that is still **uncommitted** by other transactions. This constraint is very relaxed in what matters to transactional concurrency but it may lead to some issues like **dirty reads**. Let's see the following image:

Dirty read



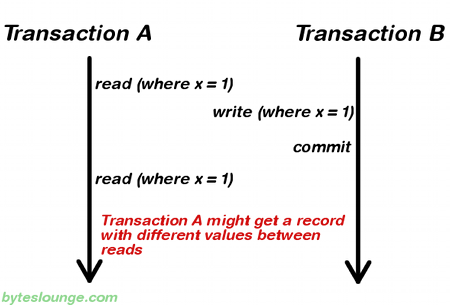
In this example **Transaction A** writes a record. Meanwhile **Transaction B** reads that same record before **Transaction A** commits. Later **Transaction A** decides to rollback and now we have changes in **Transaction B** that are inconsistent. This is a **dirty read**. **Transaction B** was running in **READ\_UNCOMMITTED** isolation level so it was able to read **Transaction A** changes before a commit occurred.

**Note:** READ\_UNCOMMITTED is also vulnerable to **non-repeatable reads** and **phantom reads**. We will also see these cases in detail in the next sections.

## READ\_COMMITTED

**READ\_COMMITTED** isolation level states that a transaction can't read data that is **not** yet committed by other transactions. This means that the **dirty read** is no longer an issue, but even this way other issues may occur. Let's see the following image:

Non-repeatable read



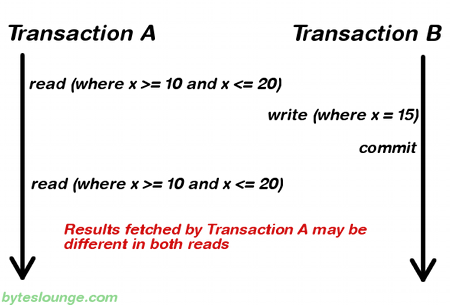
In this example **Transaction A** reads some record. Then **Transaction B** writes that same record and commits. Later **Transaction A** reads that same record again and may get different values because **Transaction B** made changes to that record and committed. This is a **non-repeatable read**.

**Note:** READ\_COMMITTED is also vulnerable to **phantom reads**. We will also see this case in detail in the next section.

## REPEATABLE\_READ

**REPEATABLE\_READ** isolation level states that if a transaction reads one record from the database multiple times the result of all those reading operations must always be the same. This eliminates both the **dirty read** and the **non-repeatable read** issues, but even this way other issues may occur. Let's see the following image:

Phantom read



In this example **Transaction A** reads a **range** of records. Meanwhile **Transaction B** inserts a new record in the same range that **Transaction A** initially fetched and commits. Later **Transaction A** reads the same range again and will also get the record that **Transaction B** just inserted. This is a **phantom read**: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

## SERIALIZABLE

**SERIALIZABLE** isolation level is the most restrictive of all isolation levels. Transactions are executed with locking at all levels (**read**, **range** and **write** locking) so they appear as if they were executed in a serialized way. This leads to a scenario where **none** of the issues mentioned above may occur, but in the other way we don't allow transaction concurrency and consequently introduce a performance penalty.

## DEFAULT

**DEFAULT** isolation level, as the name states, uses the default isolation level of the datastore we are actually connecting from our application.

## Summary

To summarize, the existing relationship between isolation level and read phenomena may be expressed in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **dirty reads** | **non-repeatable reads** | **phantom reads** |
| **READ\_UNCOMMITTED** | yes | yes | yes |
| **READ\_COMMITTED** | no | yes | yes |
| **REPEATABLE\_READ** | no | no | yes |
| **SERIALIZABLE** | no | no | no |

## JPA

If you are using Spring with JPA you may come across the following exception when you use an isolation level that is different the default:

InvalidIsolationLevelException: Standard JPA does not support custom isolation levels - use a special JpaDialect for your JPA implementation  
at org.springframework.orm.jpa.DefaultJpaDialect.beginTransaction(DefaultJpaDialect.java:67)  
at org.springframework.orm.jpa.JpaTransactionManager.doBegin(JpaTransactionManager.java:378)  
at org.springframework.transaction.support.AbstractPlatformTransactionManager.getTransaction(AbstractPlatformTransactionManager.java:372)  
at org.springframework.transaction.interceptor.TransactionAspectSupport.createTransactionIfNecessary(TransactionAspectSupport.java:417)  
at org.springframework.transaction.interceptor.TransactionAspectSupport.invokeWithinTransaction(TransactionAspectSupport.java:255)  
at org.springframework.transaction.interceptor.TransactionInterceptor.invoke(TransactionInterceptor.java:94)  
at org.springframework.aop.framework.ReflectiveMethodInvocation.proceed(ReflectiveMethodInvocation.java:172)  
at org.springframework.aop.framework.JdkDynamicAopProxy.invoke(JdkDynamicAopProxy.java:204)

To solve this problem you must implement a custom JPA dialect which is explained in detail in the following article: [**Spring - Change transaction isolation level example**](http://www.byteslounge.com/tutorials/spring-change-transaction-isolation-level-example).

# Spring transaction propagation tutorial

26 January 2013

By [Gonçalo Marques](https://plus.google.com/107771575694787223844?rel=author)

[java](http://www.byteslounge.com/tag/java)[spring](http://www.byteslounge.com/tag/spring)[tx](http://www.byteslounge.com/tag/tx)

In this tutorial you will learn about the transaction propagation behaviors provided by the Spring framework.

## Introduction

While dealing with Spring managed transactions the developer is able to specify how the transactions should behave in terms of propagation. In other words the developer has the ability to decide how the business methods should be encapsulated in both logical or physical transactions. Methods from distinct Spring beans may be executed in the same transaction scope or actually being spanned across multiple nested transactions. This may lead to details like how does the inner transaction outcome result affects the outer transactions. We will see the different propagation behaviors provided by Spring in the next sections.

This tutorial will only focus in transaction propagation behavior. For other details about Spring transactions you may refer to other tutorials on this website or the official Spring documentation at SpringSource website.

The full source code used in this tutorial is available for download at the bottom of this page. We will only show the relevant parts for a good understanding of the distinct transaction propagation behaviors in Spring.   
  
The full source code uses Hibernate to implement the persistence layer ([Spring with Hibernate persistence and transactions example](http://www.byteslounge.com/tutorials/spring-with-hibernate-persistence-and-transactions-example)).

## REQUIRED behavior

Spring **REQUIRED** behavior means that the same transaction will be used if there is an already opened transaction in the current bean method execution context. If there is no existing transaction the Spring container will create a new one. If multiple methods configured as **REQUIRED** behavior are called in a nested way they will be assigned **distinct logical transactions** but they will all share the **same physical transaction**. In short this means that if an inner method causes a transaction to rollback, the outer method will fail to commit and will also rollback the transaction. Let's see an example:

Outer bean

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequired();

} catch(RuntimeException e){

// handle exception

}

}

Inner bean

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequired() {

throw new RuntimeException("Rollback this transaction!");

}

Note that the inner method throws a **RuntimeException** and is annotated with **REQUIRED** behavior. This means that it will use the **same** transaction as the outer bean, so the outer transaction will fail to commit and will also rollback.

**Note:** The only exceptions that set a transaction to rollback state by default are the unchecked exceptions (like **RuntimeException**). If you want checked exceptions to also set transactions to rollback you must configure them to do so, but this will not be covered in this tutorial.   
  
**Note 2:** When using declarative transactions, ie by using only annotations, and calling methods from the same bean directly (self-invocation), the **@Transactional** annotation will be ignored by the container. If you want to enable transaction management in self-invocations you must configure the transactions using **aspectj**, but this will not be covered in this tutorial.

## REQUIRES\_NEW behavior

**REQUIRES\_NEW** behavior means that a new physical transaction will always be created by the container. In other words the inner transaction may commit or rollback independently of the outer transaction, i.e. the outer transaction will not be affected by the inner transaction result: they will run in **distinct physical transactions**.

Outer bean

@Autowired

private TestDAO testDAO;

@Autowired

private InnerBean innerBean;

@Override

@Transactional(propagation=Propagation.REQUIRED)

public void testRequiresNew(User user) {

testDAO.insertUser(user);

try{

innerBean.testRequiresNew();

} catch(RuntimeException e){

// handle exception

}

}

Inner bean

@Override

@Transactional(propagation=Propagation.REQUIRES\_NEW)

public void testRequiresNew() {

throw new RuntimeException("Rollback this transaction!");

}

The inner method is annotated with **REQUIRES\_NEW** and throws a **RuntimeException** so it will set its transaction to rollback but will **not** affect the outer transaction. The outer transaction is paused when the inner transaction starts and then resumes after the inner transaction is concluded. They run independently of each other so the outer transaction may commit successfully.

## NESTED behavior

The **NESTED** behavior makes nested Spring transactions to use the same physical transaction but sets savepoints between nested invocations so inner transactions may also rollback independently of outer transactions. This may be familiar to JDBC aware developers as the savepoints are achieved with JDBC savepoints, so this behavior should only be used with Spring JDBC managed transactions ([**Spring JDBC transactions example**](http://www.byteslounge.com/tutorials/spring-jdbc-transactions-example)).

## MANDATORY behavior

The **MANDATORY** behavior states that an existing opened transaction must already exist. If not an exception will be thrown by the container.

## NEVER behavior

The **NEVER** behavior states that an existing opened transaction must **not** already exist. If a transaction exists an exception will be thrown by the container.

## NOT\_SUPPORTED behavior

The **NOT\_SUPPORTED** behavior will execute outside of the scope of any transaction. If an opened transaction already exists it will be paused.

## SUPPORTS behavior

The **SUPPORTS** behavior will execute in the scope of a transaction if an opened transaction already exists. If there isn't an already opened transaction the method will execute anyway but in a non-transactional way.

## Quick note about the full source code

The full source code available at the end of this page considers the following MySQL table:

MySQL table used in full source code

CREATE TABLE USER (

ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

USERNAME VARCHAR (32) NOT NULL,

NAME VARCHAR (64) NOT NULL,

UNIQUE (USERNAME)

);

## Download source code from this article

**Download link:** [**spring-transaction-propagation-tutorial.zip**](http://www.byteslounge.com/repository/spring-transaction-propagation-tutorial.zip)

# Spring JTA multiple resource transactions in Tomcat with Atomikos example

17 June 2013

By [Gonçalo Marques](https://plus.google.com/107771575694787223844?rel=author)

[atomikos](http://www.byteslounge.com/tag/atomikos)[hibernate](http://www.byteslounge.com/tag/hibernate)[java](http://www.byteslounge.com/tag/java)[jpa](http://www.byteslounge.com/tag/jpa)[jta](http://www.byteslounge.com/tag/jta)[spring](http://www.byteslounge.com/tag/spring)[tomcat](http://www.byteslounge.com/tag/tomcat)[tx](http://www.byteslounge.com/tag/tx)[xa](http://www.byteslounge.com/tag/xa)

In this tutorial you will learn how to configure a JTA transaction manager outside an enterprise container - using the Spring framework and Atomikos, both deployed in Tomcat - in order to implement distributed multiple resource (or XA) transactions.

## Introduction

Distributed multiple resource transactions in Java are usually accomplished by resorting to the Java Transaction API (JTA). One usually delegates the task of distributed transaction coordination to an entity called the Transaction Manager.  
  
The transaction manager is then responsible for coordinating the distributed transaction by interacting with each resource's own Resource Manager. Fully fledged enterprise containers include a JTA implementation but what if we need JTA outside an enterprise container, ie. a servlet container like Tomcat?  
  
In this case we must use a 3rd party JTA implementation. In this tutorial we will use Atomikos, a quality JTA implementation which is also available as an open source distribution, along with the Spring framework in order to implement JTA transactions.

This tutorial considers the following environment:

1. Ubuntu 12.04
2. JDK 1.7.0.21
3. Spring 3.2.3
4. Atomikos 3.8.0
5. Tomcat 7.0.35

The following Maven dependencies are required:

Required Maven dependencies

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<spring.version>3.2.3.RELEASE</spring.version>

<hibernate.version>4.1.9.Final</hibernate.version>

<atomikos.version>3.8.0</atomikos.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-web</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-webmvc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>${hibernate.version}</version>

<exclusions>

<exclusion>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

</exclusion>

<exclusion>

<groupId>dom4j</groupId>

<artifactId>dom4j</artifactId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.0.1</version>

<scope>provided</scope>

</dependency>

<dependency>

<groupId>com.atomikos</groupId>

<artifactId>transactions-jta</artifactId>

<version>${atomikos.version}</version>

</dependency>

<dependency>

<groupId>com.atomikos</groupId>

<artifactId>transactions-jdbc</artifactId>

<version>${atomikos.version}</version>

</dependency>

<dependency>

<groupId>com.atomikos</groupId>

<artifactId>transactions-hibernate3</artifactId>

<version>${atomikos.version}</version>

<exclusions>

<exclusion>

<artifactId>hibernate</artifactId>

<groupId>org.hibernate</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>dom4j</groupId>

<artifactId>dom4j</artifactId>

<version>1.6.1</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.16</version>

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.25</version>

</dependency>

</dependencies>

## Datasources

In this tutorial we will use two datasources each one of them referencing a distinct MySQL database. We could have also included a JMS message queue to participate in the distributed transaction as this is also a very common scenario, but for simplicity we will keep up with the two MySQL datasources:

The datasources used in this example

**Datasource 1**  
**Database:** DATABASE1  
**Username:** user1  
**Database:** passwd1  
  
**Datasource 2**  
**Database:** DATABASE2  
**Username:** user2  
**Database:** passwd2

We will use a couple of tables in this example. **Datasource 1** will contain **TABLE\_ONE** and **Datasource 2** will contain **TABLE\_TWO**:

Tables used in this example:

**Datasource 1**  
CREATE TABLE TABLE\_ONE (  
TABLE\_ONE\_ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,  
VALUE VARCHAR(32) NOT NULL  
);   
  
**Datasource 2**  
CREATE TABLE TABLE\_TWO (  
TABLE\_TWO\_ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,  
VALUE VARCHAR(32) NOT NULL  
);

## JPA entities

Now we define a couple of JPA entities to map the two tables mentioned in the previous section.

TableOne.java

package com.byteslounge.spring.tx.entity;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "TABLE\_ONE")

public class TableOne {

@Id

@Column(name = "TABLE\_ONE\_ID", nullable = false)

@GeneratedValue(strategy = GenerationType.AUTO)

private int tableOneId;

@Column(name = "VALUE", nullable = false)

private String value;

public int getTableOneId() {

return tableOneId;

}

public void setTableOneId(int tableOneId) {

this.tableOneId = tableOneId;

}

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

}

TableTwo.java

package com.byteslounge.spring.tx.entity;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name = "TABLE\_TWO")

public class TableTwo {

@Id

@Column(name = "TABLE\_TWO\_ID", nullable = false)

@GeneratedValue(strategy = GenerationType.AUTO)

private int tableTwoId;

@Column(name = "VALUE", nullable = false)

private String value;

public int getTableTwoId() {

return tableTwoId;

}

public void setTableTwoId(int tableTwoId) {

this.tableTwoId = tableTwoId;

}

public String getValue() {

return value;

}

public void setValue(String value) {

this.value = value;

}

}

## DAO definition

We will also use a couple of Spring services as DAO's where each one is used to interact with the respective datasource. Following next are the DAO's interfaces and implementations:

TableOneDao.java

package com.byteslounge.spring.tx.dao;

import com.byteslounge.spring.tx.entity.TableOne;

public interface TableOneDao {

void save(TableOne tableOne);

}

TableOneDaoImpl.java

package com.byteslounge.spring.tx.dao.impl;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Service;

import com.byteslounge.spring.tx.dao.TableOneDao;

import com.byteslounge.spring.tx.entity.TableOne;

@Service

public class TableOneDaoImpl implements TableOneDao {

private EntityManager entityManager;

@PersistenceContext(unitName="PersistenceUnit1")

public void setEntityManager(EntityManager entityManager) {

this.entityManager = entityManager;

}

@Override

public void save(TableOne tableOne) {

entityManager.persist(tableOne);

}

}

TableTwoDao.java

package com.byteslounge.spring.tx.dao;

import com.byteslounge.spring.tx.entity.TableTwo;

public interface TableTwoDao {

void save(TableTwo tableTwo) throws Exception;

}

TableTwoDaoImpl.java

package com.byteslounge.spring.tx.dao.impl;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Service;

import com.byteslounge.spring.tx.dao.TableTwoDao;

import com.byteslounge.spring.tx.entity.TableTwo;

@Service

public class TableTwoDaoImpl implements TableTwoDao {

private EntityManager entityManager;

@PersistenceContext(unitName="PersistenceUnit2")

public void setEntityManager(EntityManager entityManager) {

this.entityManager = entityManager;

}

@Override

public void save(TableTwo tableTwo) throws Exception {

entityManager.persist(tableTwo);

throw new Exception("Force transaction rollback");

}

}

There are a couple of things to note in this DAO's. The first one is that each DAO is associated with a distinct **PersistenceUnit** (we will see how to configure them later in this tutorial).  
  
The second one is that the **save** method of **TableTwoDaoImpl** is explicitly throwing an exception. This exception will be used to force the global transaction to rollback. We will also see this in detail in the following tutorial sections.

## The transaction service

Finally we need a service to implement our global container managed transaction. Following next is a possible service interface and implementation:

TransactionalService.java

package com.byteslounge.spring.tx.service;

import com.byteslounge.spring.tx.entity.TableOne;

import com.byteslounge.spring.tx.entity.TableTwo;

public interface TransactionalService {

void persist(TableOne tableOne, TableTwo tableTwo) throws Exception;

}

TransactionalServiceImpl.java

package com.byteslounge.spring.tx.service.impl;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.byteslounge.spring.tx.dao.TableOneDao;

import com.byteslounge.spring.tx.dao.TableTwoDao;

import com.byteslounge.spring.tx.entity.TableOne;

import com.byteslounge.spring.tx.entity.TableTwo;

import com.byteslounge.spring.tx.service.TransactionalService;

@Service

public class TransactionalServiceImpl

implements TransactionalService {

@Autowired

private TableOneDao tableOneDao;

@Autowired

private TableTwoDao tableTwoDao;

@Override

@Transactional(rollbackFor=Exception.class)

public void persist(TableOne tableOne, TableTwo tableTwo)

throws Exception {

tableOneDao.save(tableOne);

tableTwoDao.save(tableTwo);

}

}

Thing to note in this service implementation: Method **persist** is annotated with **@Transactional** so the method will be executed in a transactional fashion by the Spring container. This method will call the data persistence methods for both DAO's we defined earlier.  
  
**Transactional** annotation has the **rollbackFor** attribute defined with **Exception.class** value. This means that the transaction will rollback if an exception of type **Exception** occurs inside persist method execution.  
  
You can refine the exception types that cause a transaction rollback but for the simplicity of this example we will keep with **Exception**.

## Persistence Context configuration

Now the Persistence Context configuration (persistence.xml):

persistence.xml

<?xml version="1.0" encoding="UTF-8"?>

<persistence version="2.0"

xmlns="http://java.sun.com/xml/ns/persistence"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence

http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd">

<persistence-unit name="PersistenceUnit1" transaction-type="JTA">

<class>com.byteslounge.spring.tx.entity.TableOne</class>

<properties>

<property name="hibernate.transaction.manager\_lookup\_class"

value="com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup" />

<property name="hibernate.transaction.factory\_class"

value="org.hibernate.transaction.CMTTransactionFactory" />

</properties>

</persistence-unit>

<persistence-unit name="PersistenceUnit2" transaction-type="JTA">

<class>com.byteslounge.spring.tx.entity.TableTwo</class>

<properties>

<property name="hibernate.transaction.manager\_lookup\_class"

value="com.atomikos.icatch.jta.hibernate3.TransactionManagerLookup" />

<property name="hibernate.transaction.factory\_class"

value="org.hibernate.transaction.CMTTransactionFactory" />

</properties>

</persistence-unit>

</persistence>

Here we configure the Persistence Units we are injecting in the DAO's we defined earlier. Note that **transaction-type** attribute of each persistence unit is defined as **JTA**.  
  
We also define where Hibernate should look for a Transaction Manager. Since we are using Atomikos we will define it as the Hibernate TransactionManagerLookup provided by Atomikos (we are using Hibernate as JPA implementation).

## Spring configuration

Now we define the necessary Spring beans and also the Spring container configuration:

spring.xml

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:p="http://www.springframework.org/schema/p"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<tx:annotation-driven />

<tx:jta-transaction-manager />

<context:component-scan

base-package="com.byteslounge.spring.tx.dao.impl" />

<context:component-scan

base-package="com.byteslounge.spring.tx.service.impl" />

<context:component-scan

base-package="com.byteslounge.spring.tx.servlet" />

<bean id="entityManagerFactory1"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="persistenceUnitName" value="PersistenceUnit1" />

<property name="dataSource" ref="dataSource1" />

<property name="jpaVendorAdapter">

<bean

class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="showSql" value="true" />

<property name="databasePlatform"

value="org.hibernate.dialect.MySQL5InnoDBDialect" />

</bean>

</property>

</bean>

<bean id="entityManagerFactory2"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">

<property name="persistenceUnitName" value="PersistenceUnit2" />

<property name="dataSource" ref="dataSource2" />

<property name="jpaVendorAdapter">

<bean

class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<property name="showSql" value="true" />

<property name="databasePlatform"

value="org.hibernate.dialect.MySQL5InnoDBDialect" />

</bean>

</property>

</bean>

<bean id="dataSource1"

class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="DataSource1" />

<property name="xaDataSource" ref="dataBase1" />

<property name="poolSize" value="3" />

</bean>

<bean id="dataBase1"

class="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"

lazy-init="true">

<property name="pinGlobalTxToPhysicalConnection" value="true" />

<property name="user" value="user1" />

<property name="password" value="passwd1" />

<property name="url" value="jdbc:mysql://localhost:3306/DATABASE1" />

</bean>

<bean id="dataSource2"

class="com.atomikos.jdbc.AtomikosDataSourceBean"

init-method="init" destroy-method="close">

<property name="uniqueResourceName" value="DataSource2" />

<property name="xaDataSource" ref="dataBase2" />

<property name="poolSize" value="3" />

</bean>

<bean id="dataBase2"

class="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"

lazy-init="true">

<property name="pinGlobalTxToPhysicalConnection" value="true" />

<property name="user" value="user2" />

<property name="password" value="passwd2" />

<property name="url" value="jdbc:mysql://localhost:3306/DATABASE2" />

</bean>

<bean id="atomikosTransactionManager"

class="com.atomikos.icatch.jta.UserTransactionManager"

init-method="init" destroy-method="close">

<property name="forceShutdown" value="false" />

</bean>

<bean id="atomikosUserTransaction"

class="com.atomikos.icatch.jta.J2eeUserTransaction">

<property name="transactionTimeout" value="300" />

</bean>

<bean id="transactionManager"

class="org.springframework.transaction.jta.JtaTransactionManager"

depends-on="atomikosTransactionManager,atomikosUserTransaction">

<property name="transactionManager"

ref="atomikosTransactionManager" />

<property name="userTransaction"

ref="atomikosUserTransaction" />

<property name="allowCustomIsolationLevels" value="true" />

</bean>

</beans>

Things to note in Spring configuration:  
  
We are instructing Spring to use a JTA transaction manager (**jta-transaction-manager** configuration element).  
  
We define a couple of beans to represent the both database connections (beans **dataBase1** and **dataBase2**). The datasource connections are using **com.mysql.jdbc.jdbc2.optional.MysqlXADataSource** class.  
  
We are defining two Atomikos datasources (**dataSource1** and **dataSource2**) that will be coupled with the respective database connections we just defined.  
  
We are also defining a couple of Entity Manager factories each one associated with the respective Atomikos datasource and the Persistence Units we defined in the previous section.  
  
Finally we define the Atomikos JTA Transaction Manager and the Atomikos JTA User Transaction that will be both used by The Spring JTA Transaction Manager.  
  
If you need more information on the JTA Transaction Manager and JTA User Transaction roles in a JTA distributed transaction you should search the Internet for additional documentation as there is plenty available.

## Testing the example

Now we define a simple servlet to test the configuration:

Testing servlet

package com.byteslounge.spring.tx.servlet;

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import org.springframework.web.HttpRequestHandler;

import com.byteslounge.spring.tx.entity.TableOne;

import com.byteslounge.spring.tx.entity.TableTwo;

import com.byteslounge.spring.tx.service.TransactionalService;

@Component("testServlet")

public class TestServlet implements HttpRequestHandler {

@Autowired

private TransactionalService transactionalService;

@Override

public void handleRequest(

HttpServletRequest req,

HttpServletResponse resp)

throws ServletException, IOException {

TableOne tableOne = new TableOne();

tableOne.setValue("value1");

TableTwo tableTwo = new TableTwo();

tableTwo.setValue("value2");

try {

transactionalService.persist(tableOne, tableTwo);

} catch (Exception e) {

e.printStackTrace();

}

}

}

Note that we are implementing Spring **HttpRequestHandler** interface so we are able to use Spring dependency injection in the servlet itself, but we will not detail this subject in the current tutorial.

After the servlet execution we will observe that **neither** of the records are inserted in **TABLE\_ONE** and **TABLE\_TWO**. This is because the second transaction is explicitly throwing an Exception as we configured it in the previous sections and so the global transaction is rollback.

## Make the transaction commit

In order to make the transaction to successfully commit we just need to remove the explicit exception throwing in **Datasource 2** DAO:

Modified TableTwoDaoImpl.java

package com.byteslounge.spring.tx.dao.impl;

import javax.persistence.EntityManager;

import javax.persistence.PersistenceContext;

import org.springframework.stereotype.Service;

import com.byteslounge.spring.tx.dao.TableTwoDao;

import com.byteslounge.spring.tx.entity.TableTwo;

@Service

public class TableTwoDaoImpl implements TableTwoDao {

private EntityManager entityManager;

@PersistenceContext(unitName="PersistenceUnit2")

public void setEntityManager(EntityManager entityManager) {

this.entityManager = entityManager;

}

@Override

public void save(TableTwo tableTwo) throws Exception {

// Exception throwing is removed

entityManager.persist(tableTwo);

}

}

## Downloadable sample

You may find the complete example source code as a downloadable resource at the end of this page. The downloadable sample **is explicitly throwing the exception** so you may observe the behaviour of global transaction rollback. Change it in order to suit your needs.

## Download source code from this article

**Download link:** [**spring-jta-multiple-resource-transactions-in-tomcat-with-atomikos-example.zip**](http://www.byteslounge.com/repository/spring-jta-multiple-resource-transactions-in-tomcat-with-atomikos-example.zip)

# Spring JDBC transactions example

29 December 2012

By [Gonçalo Marques](https://plus.google.com/107771575694787223844?rel=author)

[java](http://www.byteslounge.com/tag/java)[jdbc](http://www.byteslounge.com/tag/jdbc)[spring](http://www.byteslounge.com/tag/spring)[tx](http://www.byteslounge.com/tag/tx)

In this tutorial you will learn how to implement JDBC transactions using the Spring framework.

## Introduction

In this tutorial we will explore one of the most powerful Spring features: The transaction management. At the end of this tutorial we will have used the JDBC Transaction Manager in a declarative way so Spring manages all transaction related boilerplate synchronization for us.

Comprehensive documentation on this subject can be found at the official Spring website.

This tutorial considers the following software and environment:

1. Ubuntu 12.04
2. Maven 3.0.4
3. JDK 1.7.0.09
4. Spring 3.2.0
5. MySQL 5.5.28

## Configuration

Configure Maven to get the required Spring dependencies:

Maven pom.xml file referencing required dependencies

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.byteslounge.spring.tx</groupId>

<artifactId>com-byteslounge-spring-tx</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>com-byteslounge-spring-tx</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- Define Spring version as a constant -->

<spring.version>3.2.0.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

</dependencies>

</project>

Now place yourself in the project directory and issue the following command to prepare your project for Eclipse:

**mvn eclipse:eclipse**

After conclusion you can import the project into Eclipse.

This tutorial will not focus on how to configure a MySQL instance or database but will consider the following table:

MySQL table used in this example

CREATE TABLE USER (

ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

USERNAME VARCHAR (32) NOT NULL,

NAME VARCHAR (64) NOT NULL,

UNIQUE (USERNAME)

);

## Model and DAO

We will need a simple Java class to represent **USER** table information. This class will be the model for this example.

User.java class

package com.byteslounge.spring.tx.model;

public class User {

private int id;

private String username;

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Now we define our DAO interface and implementation:

DAO interface

package com.byteslounge.spring.tx.dao;

import java.util.List;

import com.byteslounge.spring.tx.model.User;

public interface UserDAO {

void insertUser(User user);

User getUser(String username);

List<User> getUsers();

}

DAO implementation

package com.byteslounge.spring.tx.dao.impl;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.jdbc.core.RowMapper;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

import org.springframework.stereotype.Service;

import com.byteslounge.spring.tx.dao.UserDAO;

import com.byteslounge.spring.tx.model.User;

@Service

public class UserDAOImpl extends JdbcDaoSupport implements UserDAO {

@Autowired

public UserDAOImpl(DataSource dataSource) {

setDataSource(dataSource);

}

@Override

public void insertUser(User user) {

getJdbcTemplate().update(

"INSERT INTO USER (USERNAME, NAME) VALUES (?, ?)",

new Object[] {

user.getUsername(),

user.getName()

}

);

}

@Override

public User getUser(String username) {

User user = getJdbcTemplate().

queryForObject("SELECT \* FROM USER WHERE USERNAME = ?",

new Object[] { username },

new UserMapper()

);

return user;

}

@Override

public List<User> getUsers() {

List<User> users = getJdbcTemplate().

query("SELECT \* FROM USER",

new UserMapper()

);

return users;

}

private class UserMapper implements RowMapper<User>{

@Override

public User mapRow(ResultSet rs, int rowNum)

throws SQLException {

User user = new User();

user.setId(rs.getInt("ID"));

user.setUsername(rs.getString("USERNAME"));

user.setName(rs.getString("NAME"));

return user;

}

}

}

We are basically defining three operations that will be executed over our example **USER** table: Insert a new user, fetching a user by its username and fetching all users. There are a couple of things to note here. The first is the **@Service** annotation. This DAO will be injected by the Spring container into another managed bean. The second is that we are extending **JdbcDaoSupport**. This Spring class represents an abstraction layer around JDBC so we don't need to implement the JDBC boilerplate code ourselves but we delegate this wiring to Spring instead. **JdbcDaoSupport** needs a **Datasource** so we also inject it using **@Autowired** annotation at the constructor level (the Datasource will be configured later as a managed bean in Spring configuration file).

## The Service bean

Now we need to define the actual service bean that will make use of the DAO we previously defined. You usually implement your business logic in this layer: The service layer. Since this is a very simple example the service layer will just make use of the DAO to interact with the Database and return the results directly to the caller.

Service interface

package com.byteslounge.spring.tx.user;

import java.util.List;

import com.byteslounge.spring.tx.model.User;

public interface UserManager {

void insertUser(User user);

User getUser(String username);

List<User> getUsers();

}

Service implementation

package com.byteslounge.spring.tx.user.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.byteslounge.spring.tx.dao.UserDAO;

import com.byteslounge.spring.tx.model.User;

import com.byteslounge.spring.tx.user.UserManager;

@Service

public class UserManagerImpl implements UserManager {

@Autowired

private UserDAO userDAO;

@Override

@Transactional

public void insertUser(User user) {

userDAO.insertUser(user);

}

@Override

public User getUser(String username) {

return userDAO.getUser(username);

}

@Override

public List<User> getUsers() {

return userDAO.getUsers();

}

}

As we have already stated before it should be in this service layer that the business logic would be implemented. In this simple example we are just using the DAO to interact with the Database and return the results to the caller. Things to note in this class: The service implementation is annotated with **@Service** which means that this will be a bean managed by Spring. **UserDAO** is annotated with **@Autowired** so it will be injected by the Spring container. **insertUser(User user)** method is annotated with **@Transactional** so every operations that occur inside this method will be executed in a transactional way by Spring JDBC Transaction Manager.

## Spring configuration file

Now we define the configuration file used for this example:

Spring XML configuration file

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:tx="http://www.springframework.org/schema/tx"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<tx:annotation-driven />

<context:component-scan

base-package="com.byteslounge.spring.tx.dao.impl" />

<context:component-scan

base-package="com.byteslounge.spring.tx.user.impl" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/TEST" />

<property name="username" value="testuser" />

<property name="password" value="testpasswd" />

</bean>

<bean id="transactionManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

</beans>

Important things to note in the configuration file: We define a **datasource** bean pointing to our MySQL instance. This **datasource** bean will be used in our DAO as we have seen previously. There is also a **transactionManager** bean. This bean is the Spring JDBC transaction manager that will handle transaction related boilerplate code and wiring for us. **tx:annotation-driven** element defines that we are declaring transactions using annotations in our classes (remember **@Transactional** annotations in our service layer?). Finally we define the packages where Spring should look for beans using **context:component-scan** elements.

**Note:** In this example we used MySQL as the data repository so we need to specify the correct MySQL Driver in the **dataSource** bean. This Driver must be in the application classpath when you run your application. Drivers can be usually found in the respective vendor websites. In our case we got it from MySQL website.

## Testing the application

Let's create a simple class to test our example:

Simple Main testing class

package com.byteslounge.spring.tx;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.byteslounge.spring.tx.model.User;

import com.byteslounge.spring.tx.user.UserManager;

public class Main

{

public static void main( String[] args ) {

ApplicationContext ctx =

new ClassPathXmlApplicationContext("spring.xml");

UserManager userManager =

(UserManager) ctx.getBean("userManagerImpl");

User user = new User();

user.setUsername("johndoe");

user.setName("John Doe");

userManager.insertUser(user);

System.out.println("User inserted!");

user = userManager.getUser("johndoe");

System.out.println("\nUser fetched!"

+ "\nId: " + user.getId()

+ "\nUsername: " + user.getUsername()

+ "\nName: " + user.getName());

List<User> users = userManager.getUsers();

System.out.println("\nUser list fetched!"

+ "\nUser count: " + users.size());

}

}

When we run our test the following output will be generated:

**User inserted!   
  
User fetched!   
Id: 1   
Username: johndoe   
Name: John Doe   
  
User list fetched!   
User count: 1**

**Remember that the Driver should be in the application classpath.**

This tutorial source code can be found at the end of this page.

## Download source code from this article

**Download link:** [**spring-jdbc-transactions-example.zip**](http://www.byteslounge.com/repository/spring-jdbc-transactions-example.zip)

# Spring with Hibernate persistence and transactions example

02 January 2013

By [Gonçalo Marques](https://plus.google.com/107771575694787223844?rel=author)

[hibernate](http://www.byteslounge.com/tag/hibernate)[java](http://www.byteslounge.com/tag/java)[jpa](http://www.byteslounge.com/tag/jpa)[spring](http://www.byteslounge.com/tag/spring)[tx](http://www.byteslounge.com/tag/tx)

In this tutorial you will learn how to implement Hibernate persistence using the Spring framework in a transactional fashion.

## Introduction

After reading this tutorial you will be able to implement Hibernate persistence using Spring framework. Additionally you will also use Spring Hibernate transaction manager to manage all the transactional boilerplate code and wiring for you.

This tutorial considers the following software and environment:

1. Ubuntu 12.04
2. Maven 3.0.4
3. JDK 1.7.0.09
4. Spring 3.2.0
5. Hibernate 4.1.9
6. MySQL 5.5.28

## Configuration

Configure Maven to get the required Spring dependencies:

Maven pom.xml file referencing required dependencies

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.byteslounge.spring.tx</groupId>

<artifactId>com-byteslounge-spring-tx</artifactId>

<version>1.0-SNAPSHOT</version>

<packaging>jar</packaging>

<name>com-byteslounge-spring-tx</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<!-- Define Spring version as a constant -->

<spring.version>3.2.0.RELEASE</spring.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-tx</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>${spring.version}</version>

</dependency>

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.2.2</version>

</dependency>

<dependency>

<groupId>javax.persistence</groupId>

<artifactId>persistence-api</artifactId>

<version>1.0</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>4.1.9.Final</version>

</dependency>

</dependencies>

</project>

Now place yourself in the project directory and issue the following command to prepare your project for Eclipse:

**mvn eclipse:eclipse**

After conclusion you can import the project into Eclipse.

This tutorial will not focus on how to configure a MySQL instance or database but will consider the following table:

MySQL table used in this example

CREATE TABLE USER (

ID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

USERNAME VARCHAR (32) NOT NULL,

NAME VARCHAR (64) NOT NULL,

UNIQUE (USERNAME)

);

## Entity and DAO

We will need a simple Java class to represent **USER** table information in the form of a JPA Entity. This class will be the model for this example.

User.java class

package com.byteslounge.spring.tx.model;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="USER")

public class User {

@Id

@GeneratedValue(strategy = GenerationType.AUTO)

@Column(name="ID", nullable = false)

private int id;

@Column(name="USERNAME", nullable = false)

private String username;

@Column(name="NAME", nullable = false)

private String name;

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getUsername() {

return username;

}

public void setUsername(String username) {

this.username = username;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

Basically we are defining our model object and the mappings of the **User** object to the **USER** table. If you are familiar with JPA this will be straight forward.

Now we define our DAO interface and implementation:

DAO interface

package com.byteslounge.spring.tx.dao;

import java.util.List;

import com.byteslounge.spring.tx.model.User;

public interface UserDAO {

void insertUser(User user);

User getUserById(int userId);

User getUser(String username);

List<User> getUsers();

}

DAO implementation

package com.byteslounge.spring.tx.dao.impl;

import java.util.List;

import org.hibernate.Criteria;

import org.hibernate.Query;

import org.hibernate.SessionFactory;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import com.byteslounge.spring.tx.dao.UserDAO;

import com.byteslounge.spring.tx.model.User;

@Service

public class UserDAOImpl implements UserDAO {

@Autowired

private SessionFactory sessionFactory;

@Override

public void insertUser(User user) {

sessionFactory.getCurrentSession().save(user);

}

@Override

public User getUserById(int userId) {

return (User) sessionFactory.

getCurrentSession().

get(User.class, userId);

}

@Override

public User getUser(String username) {

Query query = sessionFactory.

getCurrentSession().

createQuery("from User where username = :username");

query.setParameter("username", username);

return (User) query.list().get(0);

}

@Override

@SuppressWarnings("unchecked")

public List<User> getUsers() {

Criteria criteria = sessionFactory.

getCurrentSession().

createCriteria(User.class);

return criteria.list();

}

}

We are basically defining some operations that will be executed over the **USER** table. Insert a new user, get a user by ID (Primary Key), get a user by it's username and fetching all users.  
  
We use the **@Autowired** in the **SessionFactory** property so it gets injected by the Spring container during bean initialization. We will see later how we configure our Hibernate session factory.

## The Service bean

Now we need to define the actual service bean that will make use of the DAO we previously defined. You usually implement your business logic in this layer: The service layer. Since this is a very simple example the service layer will just make use of the DAO to interact with the Database and return the results directly to the caller.

Service interface

package com.byteslounge.spring.tx.user;

import java.util.List;

import com.byteslounge.spring.tx.model.User;

public interface UserManager {

void insertUser(User user);

User getUserById(int userId);

User getUser(String username);

List<User> getUsers();

}

Service implementation

package com.byteslounge.spring.tx.user.impl;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Service;

import org.springframework.transaction.annotation.Transactional;

import com.byteslounge.spring.tx.dao.UserDAO;

import com.byteslounge.spring.tx.model.User;

import com.byteslounge.spring.tx.user.UserManager;

@Service

public class UserManagerImpl implements UserManager {

@Autowired

private UserDAO userDAO;

@Override

@Transactional

public void insertUser(User user) {

userDAO.insertUser(user);

}

@Override

@Transactional

public User getUserById(int userId) {

return userDAO.getUserById(userId);

}

@Override

@Transactional

public User getUser(String username) {

return userDAO.getUser(username);

}

@Override

@Transactional

public List<User> getUsers() {

return userDAO.getUsers();

}

}

As we have already stated before it should be in this service layer that the business logic would be implemented. In this simple example we are just using the DAO to interact with the Database and return the results to the caller.  
  
Things to note in this class: The service implementation is annotated with **@Service** which means that this will be a bean managed by Spring. **UserDAO** is annotated with **@Autowired** so it will be injected by the Spring container.  
  
Methods are annotated with **@Transactional** so the Spring Hibernate transaction manager creates the required transactions and the respective sessions.

## Spring configuration file

Now we define the configuration file used for this example:

Spring XML configuration file

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:tx="http://www.springframework.org/schema/tx"

xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-3.0.xsd

http://www.springframework.org/schema/tx

http://www.springframework.org/schema/tx/spring-tx.xsd">

<tx:annotation-driven />

<context:component-scan

base-package="com.byteslounge.spring.tx.dao.impl" />

<context:component-scan

base-package="com.byteslounge.spring.tx.user.impl" />

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy-method="close">

<property name="driverClassName" value="com.mysql.jdbc.Driver" />

<property name="url" value="jdbc:mysql://localhost:3306/TEST" />

<property name="username" value="testuser" />

<property name="password" value="testpasswd" />

</bean>

<bean id="sessionFactory" class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">

<property name="dataSource" ref="dataSource"></property>

<property name="hibernateProperties">

<props>

<prop

key="hibernate.dialect">org.hibernate.dialect.MySQL5Dialect</prop>

<prop key="hibernate.show\_sql">true</prop>

</props>

</property>

<property name="packagesToScan" value="com.byteslounge.spring.tx.model" />

</bean>

<bean id="transactionManager"

class="org.springframework.orm.hibernate4.HibernateTransactionManager"

p:sessionFactory-ref="sessionFactory">

</bean>

</beans>

Important things to note in the configuration file: We define a **datasource** bean pointing to our MySQL instance. The **sessionFactory** bean represents the Hibernate session factory that will create sessions to interact with the database.  
  
We needed to define the packages where the container should look for Entities. In our case it will look for entities in **com.byteslounge.spring.tx.model**. We also defined the datasource that the session factory will use (property **dataSource**).  
  
There is also a **transactionManager** bean. This bean is the Spring Hibernate transaction manager that will handle transaction related boilerplate code and wiring for us. We needed to define the session factory that the transaction manager will use to create sessions (attribute **sessionFactory-ref**).  
  
**tx:annotation-driven** element defines that we are declaring transactions using annotations in our classes (remember **@Transactional** annotations in our service layer?). Finally we define the packages where Spring should look for beans using **context:component-scan** elements.

**Note:** In this example we used MySQL as the data repository so we need to specify the correct MySQL Driver in the **dataSource** bean. This Driver must be in the application classpath when you run your application. Drivers can be usually found in the respective vendor websites. In our case we got it from MySQL website.

## Testing the application

Let's create a simple class to test our example:

Simple Main testing class

package com.byteslounge.spring.tx;

import java.util.List;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.byteslounge.spring.tx.model.User;

import com.byteslounge.spring.tx.user.UserManager;

public class Main

{

public static void main( String[] args )

{

ApplicationContext ctx =

new ClassPathXmlApplicationContext("spring.xml");

UserManager userManager =

(UserManager) ctx.getBean("userManagerImpl");

User user = new User();

user.setUsername("johndoe");

user.setName("John Doe");

userManager.insertUser(user);

System.out.println("User inserted!");

user = userManager.getUser("johndoe");

System.out.println("\nUser fetched by username!"

+ "\nId: " + user.getId()

+ "\nUsername: " + user.getUsername()

+ "\nName: " + user.getName());

user = userManager.getUserById(user.getId());

System.out.println("\nUser fetched by ID!"

+ "\nId: " + user.getId()

+ "\nUsername: " + user.getUsername()

+ "\nName: " + user.getName());

List<User> users = userManager.getUsers();

System.out.println("\nUser list fetched!"

+ "\nUser count: " + users.size());

}

}

When we run our test the following output will be generated:

**User inserted!   
  
User fetched by username!   
Id: 1   
Username: johndoe   
Name: John Doe   
  
User fetched by ID!   
Id: 1   
Username: johndoe   
Name: John Doe   
  
User list fetched!   
User count: 1**

You have successfully integrated Hibernate with Spring. To be more precise you used Spring Hibernate transaction manager to manage your Hibernate sessions and transactions. The source code of this tutorial can be found at the end of this page.

## Download source code from this article

**Download link:** [**spring-with-hibernate-persistence-and-transactions-example.zip**](http://www.byteslounge.com/repository/spring-with-hibernate-persistence-and-transactions-example.zip)