Transactions

The `transact` function and `with-transaction` macro were briefly mentioned in the Getting Started section but we'll go into more detail here.

Although `(transact transactable f)` is available, it is expected that you will mostly use `(with-transaction [tx transactable] body...)` when you want to execute multiple SQL operations in the context of a single transaction so that is what this section focuses on.

Connection-level Control

By default, all connections that `next.jdbc` creates are automatically committable, i.e., as each operation is performed, the effect is committed to the database directly before the next operation is performed. Any exceptions only cause the current operation to be aborted -- any prior operations have already been committed.

It is possible to tell `next.jdbc` to create connections that do not automatically commit operations: pass `{:auto-commit false}` as part of the options map to anything that creates a connection (including `get-connection` itself). You can then decide when to commit or rollback by calling `.commit` or `.rollback` on the connection object itself. You can also create save points (`(.setSavepoint con)` , `(.setSavepoint con name)`) and rollback to them (`(.rollback con save-point)`). You can also change the auto-commit state of an open connection at any time (`(.setAutoCommit con on-off)`).

Automatic Commit & Rollback

`next.jdbc`'s transaction handling provides a convenient baseline for either committing a group of operations if they all succeed or rolling them all back if any of them fails, by throwing an exception. You can either do this on an existing connection -- and `next.jdbc` will try to restore the state of the connection after the transaction completes -- or by providing a datasource and letting `with-transaction` create and manage its own connection:

```
(jdbc/with-transaction [tx my-datasource]
  (jdbc/execute! tx ...)
  (jdbc/execute! tx ...)) ; will commit, unless exception thrown

(jdbc/with-transaction [tx my-datasource]
  (jdbc/execute! tx ...)
  (when ... (throw ...)) ; will rollback
  (jdbc/execute! tx ...))
```

If `with-transaction` is given a datasource, it will create and close the connection for you. If you pass in an existing connection, `with-transaction` will set up a transaction on that connection and, after either committing or rolling back the transaction, will restore the state of the connection and leave it open.

You can also provide an options map as the third element of the binding vector (or the third argument to the `transact` function). The following options are supported:

- `:isolation` -- the isolation level for this transaction (see All The Options for specifics),
- `:read-only` -- set the transaction into read-only mode (if `true`),
- `:rollback-only` -- set the transaction to always rollback, even on success (if `true`).

whether a "transaction" is in progress or not, and you can call `next.jdbc/active-tx?` to determine that, in your own code, in case you want to write code that behaves differently inside or outside a transaction.

> Note: `active-tx?` only knows about `next.jdbc` transactions -- it cannot track any transactions that you create yourself using the underlying JDBC `Connection`. In addition, this is a per-thread "global" setting and not related to just a single connection, so you can't use this setting if you are working with multiple databases in the same dynamic thread context (`binding`).

Manual Rollback Inside a Transaction

Instead of throwing an exception (which will propagate through `with-transaction` and therefore provide no result), you can also explicitly rollback if you want to return a result in that case:

```
(jdbc/with-transaction [tx my-datasource]
  (let [result (jdbc/execute! tx ...)]
    (if ...
      (do
        (.rollback tx)
        result)
      (jdbc/execute! tx ...))))
```

Save Points Inside a Transaction

In general, transactions are per-connection and do not nest in JDBC. If you nest calls to `with-transaction` using a `DataSource` argument (or a db-spec) then you will get separate connections inside each invocation

The outer call will then commit (or rollback) any additional operations within its scope. This will be confusing at best and most likely buggy behavior! See below for ways to exercise more control over this behavior.

If you want the ability to selectively roll back certain groups of operations inside a transaction, you can use named or unnamed save points:

```
(jdbc/with-transaction [tx my-datasource]
  (let [result (jdbc/execute! tx ...) ; op A
        sp1    (.setSavepoint tx)] ; unnamed save point

    (jdbc/execute! tx ...) ; op B

    (when ... (.rollback tx sp1)) ; just rolls back op B

    (let [sp2 (.setSavepoint tx "two")] ; named save point

      (jdbc/execute! tx ...) ; op C

      (when ... (.rollback tx sp2))) ; just rolls back op C

    result)) ; returns this and will commit op A
    ;; (and ops B & C if they weren't rolled back above)
```

Nesting Transactions

As noted above, transactions do not nest in JDBC and `next.jdbc`'s default behavior is to allow you to overlap transactions (i.e., nested calls to `with-transaction`) and assume you know what you are doing,

different way, but could be convenient if you wanted to override any transaction behavior in called code, as you might wish to do with a test fixture that set up and rolled back a transaction at the top-level -- you would just silently lose the effects of any (nested) transactions in the code under test.

`next.jdbc` provides a way to control the behavior via a public, dynamic Var:

- `next.jdbc.transaction/*nested-tx*` is initially set to `:allow` which allows nested calls but makes them overlap (as described above),
- `(binding [next.jdbc.transaction/*nested-tx* :ignore] ...)` provides the same behavior as `clojure.java.jdbc` where nested calls are essentially ignored and only the outermost transaction takes effect,
- `(binding [next.jdbc.transaction/*nested-tx* :prohibit] ...)` will cause any attempt to start a nested transaction to throw an exception instead; this could be a useful way to detect the potentially buggy behavior described above (for either `:allow` or `:ignore`).

> Note: this is a per-thread "global" setting and not related to just a single connection, so you can't use this setting if you are working with multiple databases in the same dynamic thread context (`binding`).

`with-options`

If you are using `with-options` to produce wrapped connectables / transactables, it's important to be aware that `with-transaction` produces a bare Java `java.sql.Connection` object that cannot have options -- but does allow direct interop. If you want to use `with-options` with `with-transaction`, you must either rewrap the `Connection` with a nested call to `with-options` or, as of 1.3.894, you can use `with-transaction+options` which will automatically rewrap the `Connection` in a new connectable along with the options from the original transactable. Be aware that you cannot use Java interop on this wrapped

Can you improve this documentation? These fine people already did:
Sean Corfield, Simon Robson & jet

Edit on GitHub