Migrating from `clojure.java.jdbc`

This page attempts to list all of the differences between clojure.java.jdbc and `next.jdbc`. Some of them are large and obvious, some of them are small and subtle -- all of them are deliberate design choices.

Conceptually

`clojure.java.jdbc` focuses heavily on a `db-spec` hash map to describe the various ways of interacting with the database and grew from very imperative origins that expose a lot of the JDBC API (multiple types of SQL execution, some operations returned hash maps, others update counts as integers, etc).

`next.jdbc` focuses on using protocols and native Java JDBC types where possible (for performance and simplicity) and strives to present a more modern Clojure API with namespace-qualified keywords in hash maps, reducible SQL operations as part of the primary API, and a streamlined set of SQL execution primitives. Execution always returns a hash map (for one result) or a vector of hash maps (for multiple results) -- even update counts are returned as if they were result sets.

Rows and Result Sets

`clojure.java.jdbc` returned result sets (and generated keys) as hash maps with simple, lower-case keys by default. `next.jdbc` returns result sets (and generated keys) as hash maps with qualified, as-is keys by default: each key is qualified by the name of table from which it is drawn, if known. The as-is default is chosen to a) improve performance and b) not mess with the data. Using a `:builder-fn` option of

> Note: `clojure.java.jdbc` would make column names unique by appending numeric suffixes, for example in a `JOIN` that produced columns `id` from multiple tables. `next.jdbc` does not do this: if you use *qualified* column names -- the default -- then you would get `:sometable/id` and `:othertable/id`, but with *unqualified* column names you would just get one `:id` in each row. It was always poor practice to rely on `clojure.java.jdbc`'s renaming behavior and it added quite an overhead to result set building, which is why `next.jdbc` does not support it -- use explicit column aliasing in your SQL instead if you want *unqualified* column names!

If you used `:as-arrays? true`, you will most likely want to use a `:builder-fn` option of `next.jdbc.result-set/as-unqualified-lower-arrays`.

> Note: When `next.jdbc` cannot obtain a `ResultSet` object and returns `{:next.jdbc/count N}` instead, these builder functions are not applied -- the `:builder-fn` option is not used in that situation.

Transactions

Although both libraries support transactions -- via `clojure.java.jdbc/with-db-transaction` and via `next.jdbc/with-transaction` -- there are some important considerations when you are migrating:

- `clojure.java.jdbc/with-db-transaction` allows nested calls to be present but it tracks the "depth" of such calls and "nested" calls are simply ignored (because transactions do not actually nest in JDBC).
- `next.jdbc/with-transaction` will attempt to set up a transaction on an existing `Connection` if that is what it is passed (otherwise a new `Connection` is created and a new transaction set up on that). That means that if you have nested calls, the inner transaction will commit (or rollback) all the way to the outermost transaction. `next.jdbc` "trusts" the programmer to know what they are doing. You can

use this setting if you are working with multiple databases in the same dynamic thread context (`binding`).

- Every operation in `clojure.java.jdbc` attempts to create its own transaction, which is a no-op inside an `with-db-transaction` so it is safe; transactions are *implicit* in `clojure.java.jdbc`. However, if you have migrated that `with-db-transaction` call over to `next.jdbc/with-transaction` then any `clojure.java.jdbc` operations invoked inside the body of that migrated transaction *will still try to create their own transactions* and `with-db-transaction` won't know about the outer `with-transaction` call. That means you will effectively get the "overlapping" behavior of `next.jdbc` since the `clojure.java.jdbc` operation will cause the outermost transaction to be committed or rolled back.
- None of the operations in `next.jdbc` try to create transactions -- exception `with-transaction`. All `Connection`s are auto-commit by default so it doesn't need the local transactions that `clojure.java.jdbc` tries to create; transactions are *explicit* in `next.jdbc`.

There are some strategies you can take to mitigate these differences:

1. Migrate code bottom-up so that you don't end up with calls to `clojure.java.jdbc` operations inside `next.jdbc/with-transaction` calls.
2. When you migrate a `with-db-transaction` call, think carefully about whether it could be a nested call (in which case simply remove it) or a conditionally nested call which you'll need to be much more careful about migrating.
3. You can bind `next.jdbc.transaction/*nested-tx*` to `:prohibit` which will throw exceptions if you accidentally nest calls to `next.jdbc/with-transaction`. Although you can bind it to `:ignore` in order to mimic the behavior of `clojure.java.jdbc`, that should be considered a last resort for dealing with complex conditional nesting of transaction calls. *Note that 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*

Because `clojure.java.jdbc` focuses on a hash map for the `db-spec` that is passed around, it can hold options that act as defaults for all operations on it. In addition, all operations in `clojure.java.jdbc` can accept a hash map of options and can pass those options down the call chain. In `next.jdbc`, `get-datasource`, `get-connection`, and `prepare` all produce Java objects that cannot have any extra options attached. On one hand, that means that it is harder to provide "default options", and on the other hand it means you need to be a bit more careful to ensure that you pass the appropriate options to the appropriate function, since they cannot be passed through the call chain via the `db-spec`. That's where `next.jdbc/with-options` can come in handy to wrap a connectable (generally a datasource or a connection) but be careful where you are managing connections and/or transactions directly, as mentioned in the Getting Started guide.

In All The Options, the appropriate options are shown for each function, as well as which options *will* get passed down the call chain, e.g., if a function can open a connection, it will accept options for `get-connection`; if a function can build a result set, it will accept `:builder-fn`. However, `get-datasource`, `get-connection`, and `prepare` cannot propagate options any further because they produce Java objects as their results -- in particular, `prepare` can't accept `:builder-fn` because it doesn't build result sets: only `plan`, `execute-one!`, and `execute!` can use `:builder-fn`.

In particular, this means that you can't globally override the default options (as you could with `clojure.java.jdbc` by adding your preferred defaults to the db-spec itself). If the default options do not suit your usage and you really don't want to override them in every call, it is recommended that you try to use `next.jdbc/with-options` first, and if that still doesn't satisfy you, write a wrapper namespace that implements the subset of the dozen API functions (from `next.jdbc` and `next.jdbc.sql`) that you want to use, overriding their `opts` argument with your defaults.

Primary API

`javax.sql.DataSource` as a starting point,

- `get-connection` -- overlaps with `clojure.java.jdbc` (and returns a `java.sql.Connection`) but accepts only a subset of the options (`:dbtype` / `:dbname` hash map, `String` JDBC URL); `clojure.java.jdbc/get-connection` accepts `{:datasource ds}` whereas `next.jdbc/get-connection` accepts the `javax.sql.DataSource` object directly,
- `prepare` -- somewhat similar to `clojure.java.jdbc/prepare-statement` but it accepts a vector of SQL and parameters (compared to just a raw SQL string),
- `plan` -- somewhat similar to `clojure.java.jdbc/reducible-query` but accepts arbitrary SQL statements for execution,
- `execute!` -- has no direct equivalent in `clojure.java.jdbc` (but it can replace most uses of both `query` and `db-do-commands`),
- `execute-one!` -- has no equivalent in `clojure.java.jdbc` (but it can replace most uses of `query` that currently use `:result-set-fn first`),
- `transact` -- similar to `clojure.java.jdbc/db-transaction*`,
- `with-transaction` -- similar to `clojure.java.jdbc/with-db-transaction`,
- `with-options` -- provides a way to specify "default options" over a group of operations, by wrapping the connectable (datasource or connection).

If you were using a bare `db-spec` hash map with `:dbtype` / `:dbname`, or a JDBC URL string everywhere, that should mostly work with `next.jdbc` since most functions accept a "connectable", but it would be better to create a datasource first, and then pass that around. Note that `clojure.java.jdbc` allowed the `jdbc:` prefix in a JDBC URL to be omitted but `next.jdbc` *requires that prefix!*

If you were already creating `db-spec` as a pooled connection datasource -- a `{:datasource ds}` hashmap

using `next.jdbc`'s `get-datasource` or the `->pool` function (in `next.jdbc.connection`) to create the a `javax.sql.DataSource` and then build a `db-spec` hash map with it (`{:datasource ds}`) and pass that around your program. `clojure.java.jdbc` calls can use that as-is, `next.jdbc` calls can use `(:datasource db-spec)`, so you don't have to adjust any of your call chains (assuming you're passing `db-spec` around) and you can migrate one function at a time.

If you were using other forms of the `db-spec` hash map, you'll need to adjust to one of the three modes above, since those are the only ones supported in `next.jdbc`.

The `next.jdbc.sql` namespace contains several functions with similarities to `clojure.java.jdbc`'s core API:

- `insert!` -- similar to `clojure.java.jdbc/insert!` but only supports inserting a single map,
- `insert-multi!` -- similar to `clojure.java.jdbc/insert-multi!` but only supports inserting columns and a vector of row values, or a sequence of hash maps *that all have the same keys* -- unlike `clojure.java.jdbc/insert-multi!`, you should always get a single multi-row insertion,
- `query` -- similar to `clojure.java.jdbc/query`,
- `find-by-keys` -- similar to `clojure.java.jdbc/find-by-keys` but will also accept a partial where clause (vector) instead of a hash map of column name/value pairs,
- `get-by-id` -- similar to `clojure.java.jdbc/get-by-id`,
- `update!` -- similar to `clojure.java.jdbc/update!` but will also accept a hash map of column name/value pairs instead of a partial where clause (vector),
- `delete!` -- similar to `clojure.java.jdbc/delete!` but will also accept a hash map of column name/value pairs instead of a partial where clause (vector).

If you were using `db-do-commands` in `clojure.java.jdbc` to execute DDL, the following is the equivalent

```
  (if (instance? java.sql.Connection connectable)
    (with-open [stmt (next.jdbc.prepare/statement connectable)]
      (run! #(.addBatch stmt %) commands)
      (into [] (.executeBatch stmt)))
    (with-open [conn (next.jdbc/get-connection connectable)]
      (do-commands conn commands))))
```

`:identifiers` and `:qualifier`

If you are using `:identifiers`, you will need to change to the appropriate `:builder-fn` option with one of `next.jdbc.result-set`'s `as-*` functions.

`clojure.java.jdbc`'s default is the equivalent of `as-unqualified-lower-maps` (with the caveat that conflicting column names are not made unique -- see the note above in **Rows and Result Sets**). If you specified `:identifiers identity`, you can use `as-unqualified-maps`. If you provided your own string transformation function, you probably want `as-unqualified-modified-maps` and also pass your transformation function as the `:label-fn` option.

If you used `:qualifier`, you can get the same effect with `as-modified-maps` by passing `:qualifier-fn (constantly "my_qualifier")` (and the appropriate `:label-fn` -- either `identity` or `clojure.string/lowercase`).

`:entities`

If you are using `:entities`, you will need to change to the appropriate `:table-fn` / `:column-fn` options. Table naming and column naming can be controlled separately in `next.jdbc`. Instead of the `quoted` function, there is the `next.jdbc.quoted` namespace which contains functions for the common quoting

If you are using `:result-set-fn` and/or `:row-fn`, you will need to change to explicit calls (to the result set function, or to `map` the row function), or to use the `plan` approach with `reduce` or various transducing functions.

> Note: this means that result sets are never exposed lazily in `next.jdbc` -- in `clojure.java.jdbc` you had to be careful that your `:result-set-fn` was eager, but in `next.jdbc` you either reduce the result set eagerly (via `plan`) or you get a fully-realized result set data structure back (from `execute!` and `execute-one!`). As with `clojure.java.jdbc` however, you can still stream result sets from the database and process them via reduction (was `reducible-query`, now `plan`). Remember that you can terminate a reduction early by using the `reduced` function to wrap the final value you produce.

Processing Database Metadata

There are no metadata-specific functions in `next.jdbc` but those in `clojure.java.jdbc` are only a very thin layer over the raw Java calls. Here's how metadata can be handled in `next.jdbc`:

```
(with-open [con (p/get-connection ds opts)]
  (-> (.getMetaData con) ; produces java.sql.DatabaseMetaData
      (.getTables nil nil nil (into-array ["TABLE" "VIEW"]))
      (rs/datafiable-result-set ds opts)))
```

Several methods on `DatabaseMetaData` return a `ResultSet` object. All of those can be handled similarly.

Further Minor differences

These are mostly drawn from Issue #5 although most of the bullets in that issue are described in more detail

- `with-db-connection` has been replaced by just `with-open` containing a call to `get-connection`,
- `with-transaction` can take a `:rollback-only` option, but there is no built-in way to change a transaction to rollback *dynamically*; either throw an exception (all transactions roll back on an exception) or call `.rollback` directly on the `java.sql.Connection` object (see Manual Rollback Inside a Transactions and the following section about save points),
- `clojure.java.jdbc` implicitly allowed transactions to nest and just silently ignored the inner, nested transactions (so you only really had the top-level, outermost transaction); `next.jdbc` by default assumes you know what you are doing and so an inner (nested) transaction will commit or rollback the work done so far in outer transaction (and then when that outer transaction ends, the remaining work is rolled back or committed); `next.jdbc.transaction/*nested-tx*` is a dynamic var that can be bound to `:ignore` to get similar behavior to `clojure.java.jdbc`.
- The extension points for setting parameters and reading columns are now `SettableParameter` and `ReadableColumn` protocols.

❮ datafy, nav, and :schema