com.github.seancorfield/next.jdbc 1.3.967 CLJDOC

Tap for Articles & Namespaces

next.jdbc Release Version passing Develop & Snapshot passing Pull Request passing

The next generation of `clojure.java.jdbc`: a new low-level Clojure wrapper for JDBC-based access to databases.

Featured in Jacek Schae's Learn Reitit Pro online course!

TL;DR

The latest versions on Clojars and on cljdoc:

clojars com.github.seancorfield/next.jdbc 1.3.967 cljdoc 1.3.967 slack next.jdbc slack join clojurians

The documentation on cljdoc.org is for the current version of `next.jdbc`:

- Getting Started
- API Reference
- Migrating from `clojure.java.jdbc`
- Feedback via issues or in the `#sql` channel on the Clojurians Slack or the `#sql` stream on the Clojurians Zulip.

The documentation on GitHub is for **develop** since the 1.3.967 release -- see the CHANGELOG and then read the corresponding updated documentation on GitHub if you want. Older versions of `next.jdbc` were published under the `seancorfield` group ID and you can find older seancorfield/next.jdbc documentation changes endeavor to be non-breaking (by moving to new names rather than by breaking existing names). COMMITS is an ever-increasing counter of commits since the beginning of this repository.

> Note: every commit to the **develop** branch runs CI (GitHub Actions) and successful runs push a MAJOR.MINOR.999-SNAPSHOT build to Clojars so the very latest version of `next.jdbc` is always available either via that snapshot on Clojars or via a git dependency on the latest SHA.

Motivation

Why another JDBC library? Why a different API from `clojure.java.jdbc`?

- Performance: there's a surprising amount of overhead in how `ResultSet` objects are converted to sequences of hash maps in `clojure.java.jdbc` – which can be really noticeable for large result sets – so I wanted a better way to handle that. There's also quite a bit of overhead and complexity in all the conditional logic and parsing that is associated with `db-spec`-as-hash-map.
- A more modern API, based on using qualified keywords and transducers etc: `:qualifier` and `reducible-query` in recent `clojure.java.jdbc` versions were steps toward that but there's a lot of "legacy" API in the library and I want to present a more focused, more streamlined API so folks naturally use the `IReduceInit` / transducer approach from day one and benefit from qualified keywords.
- Simplicity: `clojure.java.jdbc` uses a variety of ways to execute SQL which can lead to inconsistencies and surprises – `query`, `execute!`, and `db-do-commands` are all different ways to execute different types of SQL statement so you have to remember which is which and you often have to watch out for restrictions in the underlying JDBC API.

Those were my three primary drivers. In addition, the `db-spec`-as-hash-map approach in performance overall). There's a much clearer path of `db-spec` -> `DataSource` -> `connection` now, which should steer people toward more connection reuse and better performing apps.

I also wanted `datafy`/`nav` support baked right in (it was added to `clojure.java.jdbc` back in December 2018 as an undocumented, experimental API in a separate namespace). It is the default behavior for `execute!` and `execute-one!`. The protocol-based function `next.jdbc.result-set/datafiable-row` can be used with `plan` if you need to add `datafy`/`nav` support to rows you are creating in your reduction.

As `next.jdbc` moved from alpha to beta, the last breaking change was made (renaming `reducible!` to `plan`) and the API should be considered stable. Only accretive and fixative changes will be made from now on.

After a month of alpha builds being available for testing, the first beta build was released on May 24th, 2019. A release candidate followed on June 4th and the "gold" (1.0.0) release was on June 12th. In addition to the small, core API in `next.jdbc`, there are "syntactic sugar" SQL functions (`insert!`, `query`, `update!`, and `delete!`) available in `next.jdbc.sql` that are similar to the main API in `clojure.java.jdbc`. See Migrating from `clojure.java.jdbc` for more detail about the differences.

Usage

The primary concepts behind `next.jdbc` are that you start by producing a `javax.sql.DataSource`. You can create a pooled datasource object using your preferred library (c3p0, hikari-cp, etc). You can use `next.jdbc`'s `get-datasource` function to create a `DataSource` from a `db-spec` hash map or from a JDBC URL (string). The underlying protocol, `Sourceable`, can be extended to allow more things to be turned into a `DataSource` (and can be extended via metadata on an object as well as via types).

From a `DataSource`, either you or `next.jdbc` can create a `java.sql.Connection` via the `get-`

- `plan` -- yields an `IReduceInit` that, when reduced with an initial value, executes the SQL statement and then reduces over the `ResultSet` with as little overhead as possible.
- `execute!` -- executes the SQL statement and produces a vector of realized hash maps, that use qualified keywords for the column names, of the form `:<table>/<column>`. If you join across multiple tables, the qualified keywords will reflect the originating tables for each of the columns. If the SQL produces named values that do not come from an associated table, a simple, unqualified keyword will be used. The realized hash maps returned by `execute!` are `Datafiable` and thus `Navigable` (see Clojure 1.10's `datafy` and `nav` functions, and tools like Portal, Reveal, and Cognitect's REBL). Alternatively, you can specify `{:builder-fn rs/as-arrays}` and produce a vector with column names followed by vectors of row values. `rs/as-maps` is the default for `:builder-fn` but there are also `rs/as-unqualified-maps` and `rs/as-unqualified-arrays` if you want unqualified `:<column>` column names (and there are also lower-case variants of all of these).
- `execute-one!` -- executes the SQL or DDL statement and produces a single realized hash map. The realized hash map returned by `execute-one!` is `Datafiable` and thus `Navigable`.

In addition, there are API functions to create `PreparedStatement`s (`prepare`) from `Connection`s, which can be passed to `plan`, `execute!`, or `execute-one!`, and to run code inside a transaction (the `transact` function and the `with-transaction` macro).

Since `next.jdbc` uses raw Java JDBC types, you can use `with-open` directly to reuse connections and ensure they are cleaned up correctly:

```clojure
(let [my-datasource (jdbc/get-datasource {:dbtype "..." :dbname "..." ...})]
  (with-open [connection (jdbc/get-connection my-datasource)]
    (jdbc/execute! connection [...])
```

Usage scenarios

There are three intended usage scenarios that have driven the design of the API:

- Execute a SQL statement and process it in a single eager operation, which may allow for the results to be streamed from the database (how to persuade JDBC to do that is database-specific!), and which cleans up resources before returning the result -- even if the reduction is short-circuited via `reduced`. This usage is supported by `plan`. This is likely to be the fastest approach and should be the first option you consider for SQL queries.
- Execute a SQL or DDL statement to obtain a single, fully-realized, `Datafiable` hash map that represents either the first row from a `ResultSet`, the first generated keys result (again, from a `ResultSet`), or the first result where neither of those are available (`next.jdbc` yields `{:next.jdbc/update-count N}` when it can only return an update count). This usage is supported by `execute-one!`. This is probably your best choice for most non-query operations.
- Execute a SQL statement to obtain a fully-realized, `Datafiable` result set -- a vector of hash maps. This usage is supported by `execute!`. You can also produce a vector of column names/row values (`next.jdbc.result-set/as-arrays`).

In addition, convenience functions -- "syntactic sugar" -- are provided to insert rows, run queries, update rows, and delete rows, using the same names as in `clojure.java.jdbc`. These are in `next.jdbc.sql` since they involve SQL creation -- they are not considered part of the core API.

More Detailed Documentation

- Getting Started
- Friendly SQL Functions

- Transactions
- All The Options
- `datafy`, `nav`, and `:schema`
- Migration from `clojure.java.jdbc`

License

Can you improve this documentation? These fine people already did:
Sean Corfield, Nate Smith & jreighley

Edit on GitHub