Getting Started with next.jdbc

The `next.jdbc` library provides a simpler, faster alternative to the `clojure.java.jdbc` Contrib library and is the next step in the evolution of that library.

It is designed to work with Clojure 1.10 or later, supports `datafy` / `nav`, and by default produces hash maps with automatically qualified keywords, indicating source tables and column names (labels), if your database supports that.

Installation

You must be using Clojure 1.10 or later. 1.12.0 is the most recent stable version of Clojure (as of March 15th, 2024).

You can add `next.jdbc` to your project with either:

```
com.github.seancorfield/next.jdbc {:mvn/version "1.3.967"}
```

for `deps.edn` or:

```
[com.github.seancorfield/next.jdbc "1.3.967"]
```

- MySQL: `com.mysql/mysql-connector-j {:mvn/version "8.1.0"}` (search for latest version)
- PostgreSQL: `org.postgresql/postgresql {:mvn/version "42.6.0"}` (search for latest version)
- Microsoft SQL Server: `com.microsoft.sqlserver/mssql-jdbc {:mvn/version "12.4.1.jre11"}` (search for latest version)
- Sqlite: `org.xerial/sqlite-jdbc {:mvn/version "3.43.0.0"}` (search for latest version)

> Note: these are the versions that `next.jdbc` is tested against but there may be more recent versions and those should generally work too -- click the "search for latest version" link to see all available versions of those drivers on Maven Central. You can see the full list of drivers and versions that `next.jdbc` is tested against in the project's `deps.edn` file, but many other JDBC drivers for other databases should also work (e.g., Oracle, Red Shift).

An Example REPL Session

To start using `next.jdbc`, you need to create a datasource (an instance of `javax.sql.DataSource`). You can use `next.jdbc/get-datasource` with either a "db-spec" -- a hash map describing the database you wish to connect to -- or a JDBC URL string. Or you can construct a datasource from one of the connection pooling libraries out there, such as HikariCP or c3p0 -- see Connection Pooling below.

For the examples in this documentation, we will use a local H2 database on disk, and we'll use the Clojure CLI tools and `deps.edn`:

```
;; deps.edn
{:deps {org.clojure/clojure {:mvn/version "1.12.0"}
        com.github.seancorfield/next.jdbc {:mvn/version "1.3.967"}
```

In this REPL session, we'll define an H2 datasource, create a database with a simple table, and then add some data and query it:

```
> clj
Clojure 1.12.0
user=> (require '[next.jdbc :as jdbc])
nil
user=> (def db {:dbtype "h2" :dbname "example"})
#'user/db
user=> (def ds (jdbc/get-datasource db))
#'user/ds
user=> (jdbc/execute! ds ["
create table address (
  id int auto_increment primary key,
  name varchar(32),
  email varchar(255)
)"])
[#:next.jdbc{:update-count 0}]
user=> (jdbc/execute! ds ["
insert into address(name,email)
  values('Sean Corfield','sean@corfield.org')"])
[#:next.jdbc{:update-count 1}]
user=> (jdbc/execute! ds ["select * from address"])
[#:ADDRESS{:ID 1, :NAME "Sean Corfield", :EMAIL "sean@corfield.org"}]
user=>
```

The "db-spec" hash map

tries to guess the correct port based on the `:dbtype`).

> Note: You can see the full list of `:dbtype` values supported in next.jdbc/get-datasource's docstring. If you need this programmatically, you can get it from the next.jdbc.connection/dbtypes hash map. If those lists differ, the hash map is the definitive list (and I'll need to fix the docstring!). The docstring of that Var explains how to tell `next.jdbc` about additional databases.

The hash map can contain arbitrary keys and values: any keys not specifically recognized by `next.jdbc` will be passed through to the JDBC driver as part of the connection string. For example, if you specify `:useSSL false`, then the connection string will have `&useSSL=false` appended to it.

If you already have a JDBC URL (string), you can use that as-is instead of the db-spec hash map. If you have a JDBC URL and still need additional options passed into the JDBC driver, you can use a hash map with the `:jdbcUrl` key specifying the string and whatever additional options you need.

`execute!` & `execute-one!`

We used `execute!` to create the `address` table, to insert a new row into it, and to query it. In all three cases, `execute!` returns a vector of hash maps with namespace-qualified keys, representing the result set from the operation, if available. If the result set contains no rows, `execute!` returns an empty vector `[]`. When no result set is available, `next.jdbc` returns a "result set" containing the "update count" from the operation (which is usually the number of rows affected; note that `:builder-fn` does not affect this fake "result set"). By default, H2 uses uppercase names and `next.jdbc` returns these as-is.

If you only want a single row back -- the first row of any result set, generated keys, or update counts -- you can use `execute-one!` instead. Continuing the REPL session, we'll insert another address and ask for the generated keys to be returned, and then we'll query for a single row:

```
"] {:return-keys true})
#:ADDRESS{:ID 2}
user=> (jdbc/execute-one! ds ["select * from address where id = ?" 2])
#:ADDRESS{:ID 2, :NAME "Someone Else", :EMAIL "some@elsewhere.com"}
user=>
```

Since we used `execute-one!`, we get just one row back (a hash map). This also shows how you provide parameters to SQL statements -- with `?` in the SQL and then the corresponding parameter values in the vector after the SQL string. If the result set contains no rows, `execute-one!` returns `nil`. When no result is available, and `next.jdbc` returns a fake "result set" containing the "update count", `execute-one!` returns just a single hash map with the key `next.jdbc/update-count` and the number of rows updated.

In the same way that you would use `execute-one!` if you only want one row or one update count, compared to `execute!` for multiple rows or a vector containing an update count, you can also ask `execute!` to return multiple result sets -- such as might be returned from a stored procedure call, or a T-SQL script (for SQL Server), or multiple statements (for MySQL) -- instead of just one. If you pass the `:multi-rs true` option to `execute!`, you will get back a vector of results sets, instead of just one result set: a vector of zero or more vectors. The result may well be a mix of vectors containing realized rows and vectors containing update counts, reflecting the results from specific SQL operations in the stored procedure or script.

> Note: In general, you should use `execute-one!` for DDL operations since you will only get back an update count. If you have a SQL statement that you know will only return an update count, `execute-one!` is the right choice. If you have a SQL statement that you know will only return a single row in the result set, you probably want to use `execute-one!`. If you use `execute-one!` for a SQL statement that would return multiple rows in a result set, even though you will only get the first row back (as a hash map), the full result set will still be retrieved from the database -- it does not limit the SQL in any

All functions in `next.jdbc` (except `get-datasource`) can accept, as the optional last argument, a hash map containing a variety of options that control the behavior of the `next.jdbc` functions.

We saw `:return-keys` provided as an option to the `execute-one!` function above and mentioned the `:builder-fn` option just above that. As noted, the default behavior is to return rows as hash maps with namespace-qualified keywords identifying the column names with the table name as the qualifier. There's a whole chapter on result set builders but here's a quick example showing how to get unqualified, lower case keywords instead:

```
user=> (require '[next.jdbc.result-set :as rs])
nil
user=> (jdbc/execute-one! ds ["
insert into address(name,email)
  values('Someone Else','some@elsewhere.com')
"] {:return-keys true :builder-fn rs/as-unqualified-lower-maps})
{:id 3}
user=> (jdbc/execute-one! ds ["select * from address where id = ?" 3]
                          {:builder-fn rs/as-unqualified-lower-maps})
{:id 3, :name "Someone Else", :email "some@elsewhere.com"}
user=>
```

Relying on the default result set builder -- and table-qualified column names -- is the recommended approach to take, if possible, with a few caveats:

- MS SQL Server produces unqualified column names by default (see **Tips & Tricks** for how to get table names back from MS SQL Server),
- Oracle's JDBC driver doesn't support `.getTableName()` so it will only produce unqualified column

- If your SQL query joins tables in a way that produces duplicate column names, and you use unqualified column names, then those duplicated column names will conflict and you will get only one of them in your result -- use aliases in SQL (`as`) to make the column names distinct,
- If your SQL query joins a table to itself under different aliases, the *qualified* column names will conflict because they are based on the underlying table name provided by the JDBC driver rather the alias you used in your query -- again, use aliases in SQL to make those column names distinct.

If you want to pass the same set of options into several operations, you can use `next.jdbc/with-options` to wrap your datasource (or connection) in a way that will pass "default options". Here's the example above rewritten with that:

```
user=> (require '[next.jdbc.result-set :as rs])
nil
user=> (def ds-opts (jdbc/with-options ds {:builder-fn rs/as-unqualified-lower-maps}))
#'user/ds-opts
user=> (jdbc/execute-one! ds-opts ["
insert into address(name,email)
  values('Someone Else','some@elsewhere.com')
"] {:return-keys true})
{:id 4}
user=> (jdbc/execute-one! ds-opts ["select * from address where id = ?" 4])
{:id 4, :name "Someone Else", :email "some@elsewhere.com"}
user=>
```

> Note: See the `next.jdbc/with-option` examples in the **Datasources, Connections & Transactions** below for some caveats around using this function.

`:builder-fn` that will convert Clojure identifiers in `:kebab-case` to SQL entities in `snake_case` and will produce result sets with qualified `:kebab-case` names from SQL entities that use `snake_case`,

- `unqualified-snake-kebab-opts` -- provides `:column-fn`, `:table-fn`, `:label-fn`, `:qualifier-fn`, and `:builder-fn` that will convert Clojure identifiers in `:kebab-case` to SQL entities in `snake_case` and will produce result sets with *unqualified* `:kebab-case` names from SQL entities that use `snake_case`.

You can `assoc` any additional options you need into these pre-built option hash maps and pass the combined options into any of this library's functions.

> Note: Using `camel-snake-kebab` might also be helpful if your database has `camelCase` table and column names, although you'll have to provide `:column-fn` and `:table-fn` yourself as `->camelCase` from that library. Either way, consider relying on the *default* result set builder first and avoid converting column and table names (see Advantages of 'snake case': portability and ubiquity for an interesting discussion on kebab-case vs snake_case -- I do not agree with all of the author's points in that article, particularly his position against qualified keywords, but his argument for retaining snake_case around system boundaries is compelling).

`plan` & Reducing Result Sets

While the `execute!` and `execute-one!` functions are fine for retrieving result sets as data, most of the time you want to process that data efficiently without necessarily converting the entire result set into a Clojure data structure, so `next.jdbc` provides a SQL execution function that works with `reduce` and with transducers to consume the result set without the intermediate overhead of creating Clojure data structures for every row.

```
user=> (jdbc/execute-one! ds ["
create table invoice (
  id int auto_increment primary key,
  product varchar(32),
  unit_price decimal(10,2),
  unit_count int unsigned,
  customer_id int unsigned
)"])
#:next.jdbc{:update-count 0}
user=> (jdbc/execute-one! ds ["
insert into invoice (product, unit_price, unit_count, customer_id)
values ('apple', 0.99, 6, 100),
       ('banana', 1.25, 3, 100),
       ('cucumber', 2.49, 2, 100)
"])
#:next.jdbc{:update-count 3}
user=> (reduce
         (fn [cost row]
           (+ cost (* (:unit_price row)
                      (:unit_count row))))
         0
         (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
14.67M
```

The call to `jdbc/plan` returns an `IReduceInit` object (a "reducible collection" that requires an initial value) but does not actually run the SQL. Only when the returned object is reduced is the connection obtained from the data source, the SQL executed, and the computation performed. The connection is closed automatically when the reduction is complete. The `row` in the reduction is an abstraction over the underlying (mutable) `ResultSet` object -- it is not a Clojure data structure. Because of that, you can simply access the columns

> to the `plan` call.

Here's the same computation rewritten using `transduce`:

```
user=> (transduce
         (map #(* (:unit_price %) (:unit_count %)))
         +
         0
         (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
14.67M
```

or composing the transforms:

```
user=> (transduce
         (comp (map (juxt :unit_price :unit_count))
               (map #(apply * %)))
         +
         0
         (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
14.67M
```

If you just wanted the total item count:

```
user=> (transduce
         (map :unit_count)
         +
         0
```

You can use other functions that perform reductions to process the result of `plan`, such as obtaining a set of unique products from an invoice:

```
user=> (into #{}
             (map :product)
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
#{"apple" "banana" "cucumber"}
```

If you want to process the rows purely for side-effects, without a result, you can use `run!`:

```
user=> (run! #(println (:product %))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
apple
banana
cucumber
nil
```

Any operation that can perform key-based lookup can be used here without creating hash maps from the rows: `get`, `contains?`, `find` (returns a `MapEntry` of whatever key you requested and the corresponding column value), or direct keyword access as shown above. Any operation that would require a Clojure hash map, such as `assoc` or anything that invokes `seq` (`keys`, `vals`), will cause the full row to be expanded into a hash map, such as produced by `execute!` or `execute-one!`, which implements `Datafiable` and `Navigable` and supports lazy navigation via foreign keys, explained in `datafy`, `nav`, and the `:schema` option.

This means that `select-keys` can be used to create regular Clojure hash map from (a subset of) columns

map, just as you passed into `plan`. Compare the difference in output between these four expressions (see below for a simpler way to do this):

```
;; selects specific keys (as simple keywords):
user=> (into []
             (map #(select-keys % [:id :product :unit_price :unit_count :customer_id]))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; selects specific keys (as qualified keywords):
user=> (into []
             (map #(select-keys % [:invoice/id :invoice/product
                                   :invoice/unit_price :invoice/unit_count
                                   :invoice/customer_id]))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; selects specific keys (as qualified keywords -- ignoring the table name):
user=> (into []
             (map #(select-keys % [:foo/id :bar/product
                                   :quux/unit_price :wibble/unit_count
                                   :blah/customer_id]))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; do not do this:
user=> (into []
             (map #(into {} %))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; do this if you just want realized rows with default qualified names:
user=> (into []
             (map #(rs/datafiable-row % ds {}))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
```

away. It has still called `rs/datafiable-row` to fully-realize a datafiable and navigable hash map but it has then "poured" that into a new, empty hash map, losing the metadata.

In addition to the hash map operations described above, the abstraction over the `ResultSet` can also respond to a couple of functions in `next.jdbc.result-set`:

- `next.jdbc.result-set/row-number` - returns the 1-based row number, by calling `.getRow()` on the `ResultSet`,
- `next.jdbc.result-set/column-names` - returns a vector of column names from the `ResultSet`, as created by the result set builder specified,
- `next.jdbc.result-set/metadata` - returns the `ResultSetMetaData` object, datafied (so the result will depend on whether you have required `next.jdbc.datafy`).

> Note: Apache Derby requires the following options to be provided in order to call `.getRow()` (and therefore `row-number`): `{:concurrency :read-only, :cursors :close, :result-type :scroll-insensitive}`

If you realize a row, by calling `datafiable-row` on the abstract row passed into the reducing function, you can still call `row-number` and `column-names` on that realized row. These functions are *not* available on the realized rows returned from `execute!` or `execute-one!`, only within reductions over `plan`.

The order of the column names returned by `column-names` matches SQL's natural order, based on the operation performed, and will also match the order of column values provided in the reduction when using an array-based result set builder (`plan` provides just the column values, one row at a time, when using an array-based builder, without the leading vector of column names that you would get from `execute!`: if you call `datafiable-row` on such a row, you will get a realized vector of column values).

- `next.jdbc.plan/select-one!` -- reduces over `plan` and returns part of just the first row,
- `next.jdbc.plan/select!` -- reduces over `plan` and returns a sequence of parts of each row.

> Note: in both those cases, an appropriate initial value is supplied to the `reduce` (since `plan` returns an `IReduceInit` object).

`select!` accepts a vector of column names to extract or a function to apply to each row. It is equivalent to the following:

```
;; select! with vector of column names:
user=> (into [] (map #(select-keys % cols)) (jdbc/plan ...))
;; select! with a function:
user=> (into [] (map f) (jdbc/plan ...))
```

The `:into` option lets you override the default of `[]` as the first argument to `into`.

`select-one!` performs the same transformation on just the first row returned from a reduction over `plan`, equivalent to the following:

```
;; select-one! with vector of column names:
user=> (reduce (fn [_ row] (reduced (select-keys row cols))) nil (jdbc/plan ...))
;; select-one! with a function:
user=> (reduce (fn [_ row] (reduced (f row))) nil (jdbc/plan ...))
```

For example:

```
{:n 3}
;; apply a function:
user=> (plan/select-one!
        ds :n ["select count(*) as n from invoice where customer_id = ?" 100])
3
```

Here are some of the above sequence-producing operations, showing their `select!` equivalent:

```
user=> (require '[next.jdbc.plan :as plan])
nil
user=> (into #{}
             (map :product)
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
#{"apple" "banana" "cucumber"}
;; or:
user=> (plan/select! ds
                     :product
                     ["select * from invoice where customer_id = ?" 100]
                     {:into #{}}) ; product a set, rather than a vector
#{"apple" "banana" "cucumber"}
;; selects specific keys (as simple keywords):
user=> (into []
             (map #(select-keys % [:id :product :unit_price :unit_count :customer_id]))
             (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; or:
user=> (plan/select! ds
                     [:id :product :unit_price :unit_count :customer_id]
                     ["select * from invoice where customer_id = ?" 100])
;; selects specific keys (as qualified keywords):
```

```
              (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; or:
user=> (plan/select! ds
                     [:invoice/id :invoice/product
                      :invoice/unit_price :invoice/unit_count
                      :invoice/customer_id]
                     ["select * from invoice where customer_id = ?" 100])
;; selects specific keys (as qualified keywords -- ignoring the table name):
user=> (into []
              (map #(select-keys % [:foo/id :bar/product
                                    :quux/unit_price :wibble/unit_count
                                    :blah/customer_id]))
              (jdbc/plan ds ["select * from invoice where customer_id = ?" 100]))
;; or:
user=> (plan/select! ds
                     [:foo/id :bar/product
                      :quux/unit_price :wibble/unit_count
                      :blah/customer_id]
                     ["select * from invoice where customer_id = ?" 100])
```

> Note: you need to be careful when using stateful transducers, such as `partition-by`, when reducing over the result of `plan`. Since `plan` returns an `IReduceInit`, the resource management (around the `ResultSet`) only applies to the `reduce` operation: many stateful transducers have a completing function that will access elements of the result sequence -- and this will usually fail after the reduction has cleaned up the resources. This is an inherent problem with stateful transducers over resource-managing reductions with no good solution.

Datasources, Connections & Transactions

parameters passed in), and then closes it. If you're not using a connection pooling datasource (see below), that can be quite an overhead: setting up database connections to remote servers is not cheap!

If you want to run multiple SQL operations without that overhead each time, you can create the connection yourself and reuse it across several operations using `with-open` and `next.jdbc/get-connection`:

```
(with-open [con (jdbc/get-connection ds)]
  (jdbc/execute! con ...)
  (jdbc/execute! con ...)
  (into [] (map :column) (jdbc/plan con ...)))
```

If any of these operations throws an exception, the connection will still be closed but operations prior to the exception will have already been committed to the database. If you want to reuse a connection across multiple operations but have them all rollback if an exception occurs, you can use `next.jdbc/with-transaction`:

```
(jdbc/with-transaction [tx ds]
  (jdbc/execute! tx ...)
  (jdbc/execute! tx ...)
  (into [] (map :column) (jdbc/plan tx ...)))
```

`with-transaction` behaves somewhat like Clojure's `with-open` macro: it will (generally) create a new `Connection` for you (from `ds`) and set up a transaction on it and bind it to `tx`; if the code in the body executes successfully, it will commit the transaction and close the `Connection`; if the code in the body throws an exception, it will rollback the transaction, but still close the `Connection`.

If `ds` is a datasource, `with-transaction` will call `get-connection` on it, bind `tx` to that `Connection` and set up a transaction; run the code in the body and either commit or rollback the transaction; close the `Connection`.

If `ds` is something else, `with-transaction` will call `get-datasource` on it first and then proceed as above.

Here's what will happen in the case where `with-transaction` is given a `Connection`:

```
(with-open [con (jdbc/get-connection ds)]
  (jdbc/execute! con ...) ; auto-committed

  (jdbc/with-transaction [tx con] ; will commit or rollback this group:
    ;; note: tx is bound to the same Connection object as con
    (jdbc/execute! tx ...)
    (jdbc/execute! tx ...)
    (into [] (map :column) (jdbc/plan tx ...)))

  (jdbc/execute! con ...)) ; auto-committed
```

You can read more about working with transactions further on in the documentation.

> Note: Because `get-datasource` and `get-connection` return plain JDBC objects (`javax.sql.DataSource` and `java.sql.Connection` respectively), `next.jdbc/with-options` and `next.jdbc/with-logging` (see **Logging** below) cannot flow options across those calls, so if you are explicitly managing connections or transactions as above, you would need to have local bindings for the wrapped versions:

```
    (jdbc/with-transaction [tx con-opts] ; will commit or rollback this group:
      (let [tx-opts (jdbc/with-options tx (:options con-opts))]
        (jdbc/execute! tx-opts ...)
        (jdbc/execute! tx-opts ...)
        (into [] (map :column) (jdbc/plan tx-opts ...))))

    (jdbc/execute! con-opts ...))) ; auto-committed
```

As of 1.3.894, you can use `next.jdbc/with-transaction+options` instead, which will automatically rewrap the `Connection` with the options from the initial transactable. Be aware that means you cannot use Java interop on the new connectable because it is no longer a plain Java `java.sql.Connection` object.

Prepared Statement Caveat

Not all databases support using a `PreparedStatement` for every type of SQL operation. You might have to create a `java.sql.Statement` instead, directly from a `java.sql.Connection` and use that, without parameters, in `plan`, `execute!`, or `execute-one!`. See the following example:

```
(require '[next.jdbc.prepare :as prep])

(with-open [con (jdbc/get-connection ds)]
  (jdbc/execute! (prep/statement con) ["...just a SQL string..."])
  (jdbc/execute! con ["...some SQL..." "and" "parameters"]) ; uses PreparedStatement
  (into [] (map :column) (jdbc/plan (prep/statement con) ["..."])))
```

First, you need to add the connection pooling library as a dependency, e.g.,

```
com.zaxxer/HikariCP {:mvn/version "5.0.1"}
;; or:
com.mchange/c3p0 {:mvn/version "0.9.5.5"}
```

Check those libraries' documentation for the latest version to use!

Then import the appropriate classes into your code:

```
(ns my.main
  (:require [next.jdbc :as jdbc]
            [next.jdbc.connection :as connection])
  (:import (com.zaxxer.hikari HikariDataSource)
           ;; or:
           (com.mchange.v2.c3p0 ComboPooledDataSource PooledDataSource)))
```

Finally, create the connection pooled datasource. `db-spec` here contains the regular `next.jdbc` options (`:dbtype`, `:dbname`, and maybe `:host`, `:port`, `:classname` etc -- or the `:jdbcUrl` format mentioned above). Those are used to construct the JDBC URL that is passed into the datasource object (by calling `.setJdbcUrl` on it). You can also specify any of the connection pooling library's options, as mixed case keywords corresponding to any simple setter methods on the class being passed in, e.g., `:connectionTestQuery`, `:maximumPoolSize` (HikariCP), `:maxPoolSize`, `:preferredTestQuery` (c3p0).

In addition, for HikariCP, you can specify properties to be applied to the underlying `DataSource` itself by passing `:dataSourceProperties` with a hash map containing those properties, such as `:socketTimeout`:

```
                    :dataSourceProperties {:socketTimeout 30}})
```

(under the hood, `java.data` converts that hash map to a `java.util.Properties` object with `String` keys and `String` values)

If you need to pass in extra connection URL parameters, it can be easier to use `next.jdbc.connection/jdbc-url` to construct URL, e.g.,

```
(connection/->pool com.zaxxer.hikari.HikariDataSource
                   {:jdbcUrl
                    (connection/jdbc-url {:dbtype "mysql" :dbname "thedb" :useSSL false}
                    :username "dbuser" :password "secret"})
```

Here we pass `:useSSL false` to `jdbc-url` so that it ends up in the connection string, but pass `:username` and `:password` for the pool itself.

> Note: both HikariCP and c3p0 defer validation of the settings until a connection is requested. If you want to ensure that your datasource is set up correctly, and the database is reachable, when you first create the connection pool, you will need to call `jdbc/get-connection` on it (and then close that connection and return it to the pool). This will also ensure that the pool is fully initialized. See the examples below.

Some important notes regarding HikariCP:

- Authentication credentials must use `:username` (if you are using c3p0 or regular, non-pooled, connections, then the db-spec hash map must contain `:user`).
- When using `:dbtype "jtds"`, you must specify `:connectionTestQuery "SELECT 1"` (or some other

- When using PostgreSQL, and trying to set a default `:schema` via HikariCP, you will need to specify `:connectionInitSql "COMMIT;"` until this HikariCP issue is addressed.

You will generally want to create the connection pooled datasource at the start of your program (and close it before you exit, although that's not really important since it'll be cleaned up when the JVM shuts down):

```
(defn -main [& args]
  ;; db-spec must include :username
  (with-open [^HikariDataSource ds (connection/->pool HikariDataSource db-spec)]
    ;; this code initializes the pool and performs a validation check:
    (.close (jdbc/get-connection ds))
    ;; otherwise that validation check is deferred until the first connection
    ;; is requested in a regular operation:
    (jdbc/execute! ds ...)
    (jdbc/execute! ds ...)
    (do-other-stuff ds args)
    (into [] (map :column) (jdbc/plan ds ...))))
;; or:
(defn -main [& args]
  (with-open [^PooledDataSource ds (connection/->pool ComboPooledDataSource db-spec)]
    ;; this code initializes the pool and performs a validation check:
    (.close (jdbc/get-connection ds))
    ;; otherwise that validation check is deferred until the first connection
    ;; is requested in a regular operation:
    (jdbc/execute! ds ...)
    (jdbc/execute! ds ...)
    (do-other-stuff ds args)
    (into [] (map :column) (jdbc/plan ds ...))))
```

`stop` lifecycle. `next.jdbc` has support for Component built-in, via the `next.jdbc.connection/component` function which creates a Component-compatible entity which you can `start` and then invoke as a function with no arguments to obtain the `DataSource` within.

```
(ns my.data.program
  (:require [com.stuartsierra.component :as component]
            [next.jdbc :as jdbc]
            [next.jdbc.connection :as connection])
  (:import (com.zaxxer.hikari HikariDataSource)))

;; HikariCP requires :username instead of :user in the db-spec:
(def ^:private db-spec {:dbtype "..." :dbname "..." :username "..." :password "..."})

(defn -main [& args]
  ;; connection/component takes the same arguments as connection/->pool:
  (let [ds (component/start (connection/component HikariDataSource db-spec))]
    (try
      ;; "invoke" the data source component to get the javax.sql.DataSource:
      (jdbc/execute! (ds) ...)
      (jdbc/execute! (ds) ...)
      ;; can pass the data source component around other code:
      (do-other-stuff ds args)
      (into [] (map :column) (jdbc/plan (ds) ...))
      (finally
        ;; stopping the component will close the connection pool:
        (component/stop ds)))))
```

If you have want to either modify the connection pooled datasource after it is created, or want to perform

Working with Additional Data Types

By default, `next.jdbc` relies on the JDBC driver to handle all data type conversions when reading from a result set (to produce Clojure values from SQL values) or setting parameters (to produce SQL values from Clojure values). Sometimes that means that you will get back a database-specific Java object that would need to be manually converted to a Clojure data structure, or that certain database column types require you to manually construct the appropriate database-specific Java object to pass into a SQL operation. You can usually automate those conversions using either the `ReadableColumn` protocol (for converting database-specific types to Clojure values) or the `SettableParameter` protocol (for converting Clojure values to database-specific types).

In particular, PostgreSQL does not seem to perform a conversion from `java.util.Date` to a SQL data type automatically. You can `require` the `next.jdbc.date-time` namespace to enable that conversion.

If you are working with Java Time, some JDBC drivers will automatically convert `java.time.Instant` (and `java.time.LocalDate` and `java.time.LocalDateTime`) to a SQL data type automatically, but others will not. Requiring `next.jdbc.date-time` will enable those automatic conversions for all databases.

> Note: `next.jdbc.date-time` also provides functions you can call to enable automatic conversion of SQL date/timestamp types to Clojure data types when reading result sets. If you need specific conversions beyond that to happen automatically, consider extending the `ReadableColumn` protocol, mentioned above.

The `next.jdbc.types` namespace provides over three dozen convenience functions for "type hinting" values so that the JDBC driver might automatically handle some conversions that the default parameter setting function does not. Each function is named for the corresponding SQL type, prefixed by `as-`: `as-`

character varying. You can wrap such strings with `(as-other "...")` which tells PostgreSQL to treat this as `java.sql.Types/OTHER` when setting the parameter.

Processing Database Metadata

JDBC provides several features that let you introspect the database to obtain lists of tables, views, and so on. `next.jdbc` does not provide any specific functions for this but you can easily get this metadata from a `java.sql.Connection` and turn it into Clojure data as follows:

```
(with-open [con (p/get-connection ds opts)]
  (-> (.getMetaData con) ; produces java.sql.DatabaseMetaData
      ;; return a java.sql.ResultSet describing all tables and views:
      (.getTables nil nil nil (into-array ["TABLE" "VIEW"]))
      (rs/datafiable-result-set ds opts)))
```

Several methods on `DatabaseMetaData` return a `ResultSet` object, e.g., `.getCatalogs()`, `.getClientInfoProperties()`, `.getSchemas()`. All of those can be handled in a similar manner to the above. See the Oracle documentation for `java.sql.DatabaseMetaData` (Java 11) for more details.

If you are working with a generalized datasource that may be a `Connection`, a `DataSource`, or a wrapped connectable (via something like `with-options` or `with-transaction`), you can write generic, `Connection`-based code using `on-connection` which will reuse a `Connection` if one is passed or create a new one if needed (and automatically close it afterward):

```
(on-connection [con ds]
  (-> (.getMetaData con) ; produces java.sql.DatabaseMetaData
```

> Note: to avoid confusion and/or incorrect usage, you cannot pass options to `on-connection` because they would be ignored in some cases (existing `Connection` or a wrapped `Connection`).

As of 1.3.894, if you want the options from a wrapped connectable to flow through to the new connectable inside `on-connection`, you can use the `on-connection+options` variant of the macro. This will automatically rewrap the connectable produced with the options from the initial connectable. Be aware that means you cannot use plain Java interop inside the body of the macro because the connectable is no longer a plain Java `java.sql.Connection` object.

Logging

Sometimes it is convenient to have database operations logged automatically. `next.jdbc/with-logging` provides a way to wrap a datasource (or connection) so that operations on it will be logged via functions you provide.

There are two logging points:

- Logging the SQL and parameters prior to a database operation,
- Logging the result of a database operation.

`next.jdbc/with-logging` accepts two or three arguments and returns a connectable that can be used with `plan`, `execute!`, `execute-one!`, `prepare`, or any of the "friendly SQL functions". Since it uses a similar wrapping mechanism to `next.jdbc/with-options`, the same caveats apply -- see **Datasources, Connections & Transactions** above for details.

Logging SQL and Parameters

```
(jdbc/execute! log-ds ["more SQL" "other" "params"]))
```

The `my-sql-logger` function will be invoked for each database operation, with two arguments:

- The fully-qualified symbol identifying the operation,
- The vector containing the SQL string followed by the parameters.

The symbol will be one of: `next.jdbc/plan`, `next.jdbc/execute!`, `next.jdbc/execute-one!`, or `next.jdbc/prepare`. The friendly SQL functions invoke `execute!` or `execute-one!` under the hood, so that is how they will be logged.

The logging function can do anything it wants with the SQL and parameters. If you are logging parameter values, consider sensitive data that you might be passing in.

Logging Results

```
(let [log-ds (jdbc/with-logging ds my-sql-logger my-result-logger)]
  (jdbc/execute! log-ds ["some SQL" "and" "params"])
  ...
  (jdbc/execute! log-ds ["more SQL" "other" "params"]))
```

In addition to calling `my-sql-logger` as described above, this will also call `my-result-logger` for `execute!` and `execute-one!` operations (`plan` and `prepare` do not execute database operations directly so they do not produce results). `my-result-logger` will be invoked with three arguments:

- The fully-qualified symbol identify the operation,

The return value of the result logger function is ignored.

The symbol will be one of: `next.jdbc/execute!` or `next.jdbc/execute-one!`. The friendly SQL functions invoke `execute!` or `execute-one!` under the hood, so that is how they will be logged.

The "state" argument allows you to return data from the first logging function, such as the current time, that can be consumed by the second logging function, so that you can calculate how long an `execute!` or `execute-one!` operation took. If the first logging function returns `nil`, that will be passed as the second argument to your second logging function.

The result set data structure could be arbitrarily large. It will generally be a vector for calls to `execute!` or a hash map for calls to `execute-one!`, but its shape is determined by any `:builder-fn` options in effect. You should check if `(instance? Throwable result)` to see if the call failed and the logger has been called with the thrown exception.

For `plan` and `prepare` calls, only the first logging function is invoked (and the return value is ignored). You can use the symbol passed in to determine this.

Naive Logging with Timing

This example prints all SQL and parameters to `*out*` along with millisecond timing and results, if a result set is available:

```
dev=> (def lds (jdbc/with-logging ds
 #_=>            (fn [sym sql-params]
 #_=>              (prn sym sql-params)
 #_=>              (System/currentTimeMillis))
 #_=>            (fn [sym state result]
```

```
dev=> (sql/find-by-keys lds :foo {:name "Person"})
next.jdbc/execute! ["SELECT * FROM foo WHERE name = ?" "Person"]
next.jdbc/execute! 813 1
[#:FOO{:NAME "Person"}]
dev=>
```

A more sophisticated example could use `sym` to decide whether to just log the SQL and some parameter values or return the current time and the SQL and parameters, so that the result logging could log the SQL, parameters, *and* result set information with timing.

Support from Specs

As you are developing with `next.jdbc`, it can be useful to have assistance from `clojure.spec` in checking calls to `next.jdbc`'s functions, to provide explicit argument checking and/or better error messages for some common mistakes, e.g., trying to pass a plain SQL string where a vector (containing a SQL string, and no parameters) is expected.

You can enable argument checking for functions in `next.jdbc`, `next.jdbc.connection`, `next.jdbc.prepare`, and `next.jdbc.sql` by requiring the `next.jdbc.specs` namespace and instrumenting the functions. A convenience function is provided:

```
(require '[next.jdbc.specs :as specs])
(specs/instrument) ; instruments all next.jdbc API functions

(jdbc/execute! ds "SELECT * FROM fruit")
Call to #'next.jdbc/execute! did not conform to spec.
```

A convenience function also exists to revert that instrumentation:

```
(specs/unstrument) ; undoes the instrumentation of all next.jdbc API functions
```

❮ Changes

Friendly SQL Functions ❯