Database Access

Contents

Configuring the Database

Luminus defaults to using Migratus for database migrations and HugSQL for database interaction. The migrations and a default connection will be setup when using a database profile such as `+postgres`.

Configuring Migrations

We first have to set the connection strings for our database in `dev-config.edn` and `test-config.edn` files. These files come with a generated configuration for development and testing respectively:

`dev-config.edn`:

```
{:database-url "jdbc:postgresql://localhost/my_app_dev?user=db_user&pa
```

`test-config.edn`:

```
{:database-url "jdbc:postgresql://localhost/myapp_test?user=db_user&pa
```

Then we can create SQL scripts to migrate the database schema, and to roll it back. These are applied using the numeric order of the ids. Conventionally the current date is used to prefix the filename. The files

are expected to be present under the `resources/migrations` folder. The template will generate sample migration files for the users table.


```
resources/migrations/20150720004935-add-users-table.down.sql
resources/migrations/20150720004935-add-users-table.up.sql
```

With the above setup we can run the migrations as follows:

```
lein run migrate
```

Applied migration can then be rolled back with:

```
lein run rollback
```

Migrations can also be run via the REPL, the `user` namespace provides the following helper functions:

`(reset-db)` - resets the state of the database

`(migrate)` - runs the pending migrations

`(rollback)` - rolls back the last set of migrations

`(create-migration "add-guestbook-table")` - creates the up/down migration files with the given name

important: the database connection must be initialized before migrations can be run in the REPL

Please refer to the Database Migrations section for more details.

Setting up the database connection

The connection settings are found in the `<app>.db.core` namespace of the application. By default the database connection is expected to be provided as the `DATABASE_URL` environment variable.

The conman library is used to create a pooled connection.

The connection is initialized by calling the `conman/connect!` function with the connection atom and a database specification map. The `connect!` function will create a pooled JDBC connection using the HikariCP library. You can see the complete list of the connection options here. The connection can be terminated by calling the `disconnect!` function.

```
(ns myapp.db.core
  (:require
    ...
    [<app>.config :refer [env]]
    [conman.core :as conman]
    [mount.core :refer [defstate]]))

(defstate ^:dynamic *db*
```

```
  :start (conman/connect!
           {:jdbc-url (env :database-url)})
  :stop (conman/disconnect! *db*))
```

The connection is specified using the `defstate` macro. The `connect!` function is called when the `*db*` component enters the `:start` state and the `disconnect!` function is called when it enters the `:stop` state.

The connection needs to be dynamic in order to be automatically rebound to a transactional connection when calling query functions within the scope of `conman.core/with-transaction`.

The lifecycle of the `*db*` component is managed by the mount library as discussed in the Managing Component Lifecycle section.

The `<app>.core/start-app` and `<app>.core/stop-app` functions will initialize and tear down any components defined using `defstate` by calling `(mount/start)` and `(mount/stop)` respectively. This ensures that the connection is available when the server starts up and that it's cleaned up on server shutdown.

When working with multiple databases, a separate atom is required to track each database connection.

Translating SQL types

Certain types require translation when persisting and reading the data. The specifics of how different types are translated will vary between different database engines. It is possible to do automatic coercison of types read from the database by extending the `clojure.java.jdbc/IResultSetReadColumn` protocol.

For example, if we wanted to convert columns containing `java.sql.Date` objects to `java.util.Date` objects, then we could write the following:

```
(defn to-date [sql-date]
  (-> sql-date (.getTime) (java.util.Date.)))

(extend-protocol jdbc/IResultSetReadColumn
  java.sql.Date
  (result-set-read-column [value metadata index]
    (to-date value)))
```

The `result-set-read-column` function must accept `value`, `metadata`, and `index` parameters. The return value will be set as the data in the result map of the query.

Conversely, if we wanted to translate the data to the SQL type, then we'd extend the `java.util.Date` type:

```
(extend-type java.util.Date
  jdbc/ISQLParameter
  (set-parameter [value ^PreparedStatement stmt idx]
    (.setTimestamp stmt idx (java.sql.Timestamp. (.getTime value)))))
```

The type extensions would typically be placed in the `<app>.db.core` namespace, and will get loaded automatically when the project starts. The templates using Postgres and MySQL databases come with some extensions enabled by default.

Working with HugSQL

HugSQL takes the approach similar to HTML templating for writing SQL queries. The queries are written using plain SQL, and the dynamic parameters are specified using Clojure keyword syntax. HugSQL will use the SQL templates to automatically generate the functions for interacting with the database.

Conventionally the queries are placed in the `resources/sql/queries.sql` file. However, once your application grows you may consider splitting the queries into multiple files.

The format for the file can be seen below:

```
-- :name create-user! :! :n
-- :doc creates a new user record
INSERT INTO users
(id, first_name, last_name, email, pass)
VALUES (:id, :first_name, :last_name, :email, :pass)
```

The name of the generated function is specified using `-- :name` comment. The name is followed by the command and the result flags.

The following command flags are available:

`:?` - query with a result-set (default)

`:!` - any statement

`:<!` - support for `INSERT ... RETURNING`

`:i!` - support for insert and jdbc `.getGeneratedKeys`

The result flags are:

`:1` - one row as a hash-map

`:*` - many rows as a vector of hash-maps

`:n` - number of rows affected (inserted/updated/deleted)

`:raw` - passthrough an untouched result (default)

The query itself is written using plain SQL and the dynamic parameters are denoted by prefixing the parameter name with a colon.

The query functions are generated by calling the the `conman/bind-connection` macro. The macro accepts the connection var and one or more query files such as the one described above.

```
(conman/bind-connection conn "sql/queries.sql")
```

Note that it's also possible to bind multiple query files by providing additional file names to the `bind-connection` function:

```
(conman/bind-connection conn "sql/user-queries.sql" "sql/admin-queries
```

Once `bind-connection` is run the query we defined above will be mapped to `myapp.db.core/create-user!` function. The functions generated by `bind-connection` use the connection found in the `conn` atom by default unless one is explicitly passed in. The parameters are passed in using a map with the keys that match the parameter names specified:

```
(create-user!
  {:id "user1"
   :first_name "Bob"
   :last_name "Bobberton"
   :email "bob.bobberton@mail.com"
   :pass "verysecret"})
```

The generated function can be run without parameters, e.g:

```
(get-users)
```

It can also be passed in an explicit connection, as would be the case for running in a transaction:

```
(def some-other-conn
  (conman/connect! {:jdbc-url "jdbc:postgresql://localhost/myapp_test?

(create-user!
  some-other-conn
  {:id "user1"
   :first_name "Bob"
   :last_name "Bobberton"
   :email "bob.bobberton@mail.com"
   :pass "verysecret"})
```

The `conman` library also provides a `with-transaction` macro for running statements within a transaction. The macro rebinds the connection to the transaction connection within its body. Any SQL query functions generated by running `bind-connection` will default to using the transaction connection withing the `with-transaction` macro:

```
(with-transaction [conn]
  (jdbc/db-set-rollback-only! conn)
  (create-user!
    {:id         "foo"
     :first_name "Sam"
     :last_name  "Smith"
     :email      "sam.smith@example.com"})
  (get-user {:id "foo"}))
```

Massaging key names from SQL to Clojure style

HugSQL can be told to automatically transform underscores in the result keys into dashes by using camel-snake-kebab library:

```
(ns yuggoth.db.core
  (:require ...
            [camel-snake-kebab.extras :refer [transform-keys]]
            [camel-snake-kebab.core :refer [->kebab-case-keyword]]))

(defn result-one-snake->kebab
  [this result options]
  (->> (hugsql.adapter/result-one this result options)
       (transform-keys ->kebab-case-keyword)))

(defn result-many-snake->kebab
  [this result options]
  (->> (hugsql.adapter/result-many this result options)
       (map #(transform-keys ->kebab-case-keyword %))))

(defmethod hugsql.core/hugsql-result-fn :1 [sym]
  'yuggoth.db.core/result-one-snake->kebab)

(defmethod hugsql.core/hugsql-result-fn :one [sym]
  'yuggoth.db.core/result-one-snake->kebab)

(defmethod hugsql.core/hugsql-result-fn :* [sym]
  'yuggoth.db.core/result-many-snake->kebab)

(defmethod hugsql.core/hugsql-result-fn :many [sym]
  'yuggoth.db.core/result-many-snake->kebab)
```

See the official documentation for more details.
