Crane Manual

Fernando Borretti

June 4, 2014

Overview

Crane is an ORM for Common Lisp. It's intended to provide a simple, object-oriented interface to relational databases, inspired by the simplicity of the Django ORM and the flexible, non-opinionated philosophy of SQLAlchemy.

Structure

Crane mainly uses two libraries:

sxql A DSL for generating SQL through function composition.

cl-dbi A backend-independent interface to relational DBMSs. At present, only PostgreSQL, SQLite version 3, and MySQL are supported, but support for Oracle and MS SQL Server can be added.

Connecting

Crane autoconnects to all the databases specified in the `:databases` key of the configuration when the `crane:connect` function is called (No parameters).

Configuration for the databases might look like this:

```
(setup
 :migrations-directory
 (merge-pathnames
  #p"migrations/"
  (asdf:system-source-directory :myapp))
 :databases
 '(:main
   (:type :postgres
```

```
    :name "myapp_db"
    :user "user"
    :pass "user")))

(connect)
```

The value of `:databases` is a plist that maps a database's name (Not the actual name, but rather an identifier, like `:main` or `:users-db`) to a list of connection parameters, called the *connection spec*.

Crane maintains a list of connection specs for every supported database backend, and ensures that all required parameters and no parameters other than the required and optional ones are passed. Connection specs for all supported backends are listed in Appendix A: Connecting.

Tables

Crane uses the metaobject protocol to bind SQL tables and CLOS objects through a `TABLE-CLASS` metaclass. Table classes can be defined simply through the `deftable` macro, the syntax being:

```
(deftable <name> (<superclass>*)
  <field-or-option>*)
```

For example:

```
(deftable enemy ()
  (name :type string :pk t)
  (age :type integer :check '(:> 'age 12))
  (address :type 'string :nullp t :foreign (important-addresses :cascade :cascade))
  (fatal-weakness :type text :default "None")
  (identifying-color 'type (string 20) :unique t :foreign (colors name)))
```

Creating, Saving, and Deleting Objects

`create`

Syntax: `(create <class> <params>*)`

Create an instance of a class on the database.

Examples:

```
(create 'user :name "Eudoxia")

(create 'company :name "Initech" :founded 1994)
```

`save`

Syntax: `(save <instance>)`

Save an instance's fields to the database.

Examples:

```
(let ((point (create 'point :x 556.3 :y 26.7)))
  ;; Make some changes
  (setf (point-distance-from-origin point)
        (euclidean-distance point '(0 0)))
  ;; Save
  (save point))
```

`del`

Syntax: `(del <instance>)`

Delete an instance from the database.

Examples:

```
(defun delete-user (username)
  (del (single 'user :name username)))
```

Making Queries

High Level API

`filter`

Syntax: `(filter <class> <params>*)`

Return a list of objects that satisfy the `params`.

Examples:

```
  (filter 'company :country "US"
                   (:< nemployees 40))
```

`single`

Syntax: `(filter <class> <params>*)`

Return a single object that satisfies the parameters.

A variant, `single!`, will signal a condition when no object satisfies the parameters.

`get-or-create`

Functional SQL

Migrations

Your schema will change, and this is a fact. Most ORMs hope the users will be happy running manual `ALTER TABLEs` or provide migration functionality through an external plugin (Alembic for SQLAlchemy, South for the Django ORM).

Migrations are completely built into Crane, and are designed to be intrusive: You redefine the schema, reload, and Crane takes care of everything. If your migration plan is too complicated for Crane, then you write a simple function that does some transformations and Crane puts that in its migration history, all that without ever having to leave your Lisp environment or accessing the shell.

Example

```
(deftable employees
  (name :type string :null nil)
  (age  :type integer)
  (address :type string :null nil))
```

Now, if you decide that addresses can be nullable, you just redefine the class (Make the change, and either `C-c C-c` on Emacs or Quickload your project):

```
(deftable employees
  (name :type string :null nil)
  (age  :type integer)
  (address :type string))
```

And Crane will spot the difference and perform the migration automatically.

Transactions

Crane supports a thin wrapper over CL-DBI's transaction capabilities.

`with-transaction`

Syntax: `(with-transaction ([db-name *default-db*]) <body>*)`

Execute `body` inside a transaction. If the code executes, the transaction is comitted. If a condition is signalled, the transaction is rolled back.

Examples:

```
(with-transaction (:my-db)
  (let ((restaurants (filter '<restaurant> ...)))
      (loop for restaurant in restaurants do
          ...
          (save restaurant))))
```

`begin-transaction`

Syntax: `(begin-transaction [db-name *default-db*])`

Start a transaction on the database `db-name`.

`commit`

Syntax: `(commit [db-name *default-db*])`

Commit the current transaction on the database `db-name`.

`rollback`

Syntax: `(rollback [db-name *default-db*])`

Abort the current transaction on the database `db-name`.

Fixtures

```
;;;; initial-data.lisp
(app:user
  (:name "eudoxia"
   :groups (:admin :staff))
  (:name "joe"
   :groups (:admin)))
(app:company
  (:name "Initech"
   :city "Denver"))

;;;; myapp.asd
(asdf:defsystem myapp
  :defsystem-depends-on (:clos-fixtures)
  :components ((:module "src"
                :components
                ((:fixture "initial-data")))))
```

Inflate/Deflate

```
(definflate (stamp 'timestamp)
  ;; Inflate a timestamp value
  ;; into a timestamp object
  (local-time:universal-to-timestamp stamp))

(defdeflate (stamp local-time:timestamp)
  ;; Deflate a timestamp object
  ;; into a string
  (local-time:format-timestring nil stamp))
```

Appendix A: Connecting

PostgreSQL

Required:

`:name` Database name.
`:user` User name.
`:pass` User password.

Optional:

`:host` Host that runs the database server. Default: "`localhost`".
`:port` Port the database server listens on. Default: `5432`.
`:ssl :yes` enables secure SSL connections to the server. This might be useful if

the server is running on a host other than the default. Note that OpenSSL must be installed on both machines. For mpre information, see the relevant PostgreSQL documentation. Default: `:no`.

SQLite

Required:

`:name` The name of the database. As usual, a value of `:memory:` will create an in-memory database.

MySQL

Required and Optional: Same as PostgreSQL, except for `:ssl`. The default port number `3306`.