Extracted from:

Web Development with Clojure

Build Bulletproof Web Apps with Less Code

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

edited by Michael Swaine

Web Development with Clojure

Build Bulletproof Web Apps with Less Code

Dmitri Sotnikov

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The team that produced this book includes:

Michael Swaine (editor)
Potomac Indexing, LLC (indexer)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-937785-64-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2014

CHAPTER 4

Database Access

In the previous chapters we've primarily focused on handling the interaction between the client and the server, and only skimmed over the topic of persisting our data. In this chapter, we'll cover how to work with relational databases using the clojure.java.jdbc library. We'll then discuss how to write a simple application to generate a PDF report from database records.

Working with Relational Databases

By virtue of running on the Java Virtual Machine, Clojure has access to any database that can be accessed via Java Database Connectivity (JDBC).[1] With it, we can easily access a large number of RDBMS databases, such as MySQL, SQL Server, PostgreSQL, and Oracle. Several libraries are available for working with these databases. Let's take a quick look at our options.

The simplest library for dealing with relational databases is clojure.data.jdbc. When using this library you will have to write custom SQL for each type of database you intend to use. If you know you're going to be using a particular database, such as MySQL or PostgreSQL, this will likely not be a problem for you. However, if you ever decide to migrate, be prepared to modify your queries to match the syntax of your new database.

Another approach for dealing with RDBMS is to use a higher-level library such as SQL Korma (http://sqlkorma.com/). This library will let you write your queries using a Clojure domain-specific language (DSL) and generate the SQL statements targeting the specified back end. The obvious advantage here is that you no longer have to write SQL by hand. However, you will have to learn the DSL and will be limited to accessing only the databases it supports. Later in the book we'll see an example of using it.

1. http://en.wikipedia.org/wiki/Java_Database_Connectivity

For now, we'll focus on using the clojure.data.jdbc library, as it provides all the functionality we need without any additional complexity. We'll use PostgreSQL as our database engine throughout this book.

If you choose to use a different database engine, be aware that there might be slight syntactic differences in your SQL queries.

Accessing the Database

To access the database, we first need to include the necessary libraries in our project.clj file. We need to reference the java.jdbc library as well as the driver for the database we'll be accessing. In case of PostgreSQL we'll require the following dependencies:

```
[org.clojure/java.jdbc "0.2.3"]
[postgresql/postgresql "9.1-901.jdbc4"]
```

With that in place, we can create a new namespace to serve as the model for the application. This namespace is conventionally called models.db. We first have to reference the clojure.data.jdbc library the same way we did in the examples in Chapter 1, *Getting Your Feet Wet*, on page ?.

```
(:require [clojure.java.jdbc :as sql])
```

Next we need to define our database connection. We can do this in several ways. Let's look at these options and their pros and cons.

Defining a Parameter Map

The simplest way to define a connection is by providing a map of connection parameters.

```
(def db {:subprotocol "postgresql"
         :subname "//localhost/my_website"
         :user "admin"
         :password "admin"})
```

This is a common approach; its downside is that the connection information is stored directly in the source. The parameters you're able to supply are also more limited than they would be if you were using the driver directly.

Specifying the Driver Directly

Another option is to provide a JDBC data source and configure it manually. This option is useful if you wish to specify any driver-specific parameters not accessible through the idiomatic parameter map configuration.

```
(def db
  {:datasource
    (doto (PGPoolingDataSource.)
     (.setServerName   "localhost")
     (.setDatabaseName "my_website")
     (.setUser         "admin")
     (.setPassword     "admin")
     (.setMaxConnections 10))})
```

Defining a JNDI String

Finally, we can define the connection by specifying the Java Naming and Directory Interface (JNDI) name for a connection managed by the application server.

```
(def db {:name "jdbc/myDatasource"})
```

Here we've provided the JNDI name as a string. The actual connection will be configured on the application server you're using, and must be given the same name as the one defined in the application. When the application runs, it will query the server for the actual connection details using the name supplied.

This option separates the code in the application from the environment, which is nice. For example, you might have separate development, staging, and production servers. You can point the JNDI connection in each one to its respective database, and when you deploy your application it will pick up the connection details from the environment. The application code does not need to change, and you don't need to remember to keep track of separate profiles or environment configurations when building it.

Now that we have a database connection, let's look at how to accomplish some common tasks with it. Each database operation must be wrapped using the with-connection macro. This macro ensures that the connection is cleaned up before the function exits.

Creating Tables

We create tables by calling the create-table function and providing it the table name, followed by the columns and their types. Let's write a function to create a table to store user records, where each record has an ID and a password.

```
(defn create-users-table []
  (sql/with-connection db
    (sql/create-table
      :users
      [:id "varchar(32) PRIMARY KEY"]
      [:pass "varchar(100)"])))
```

Here, create-table is called to create a new users table. The macro takes a keyword specifying the table name, followed by vectors representing the columns. Each column has the format of [:name type], where name is the name of the column and the type can either be a SQL string or a keyword such as :int, :boolean, or :timestamp. Note: the name of the column cannot have dashes because those are not valid SQL syntax.

Selecting Records

To select records from our database we use the with-query-results macro. It accepts a vector containing the SQL string followed by its arguments and returns a result as a lazy sequence. This allows us to work with the returned data without having to load the entire result into memory.

Because the result is lazy, we must make sure we evaluate it if we intend to return it from the function. If we don't, with-connection will close our connection when we leave the function and the result will be nil. We can use doall to force the evaluation of the entire result. However, if we simply select an element as seen in the following code, that will cause the result to be evaluated implicitly.

```
(defn get-user [id]
  (sql/with-connection db
    (sql/with-query-results
      res ["select * from users where id = ?" id] (first res))))
```

In that code, we've created a function that accepts the user ID parameters and returns the first item from the result set.

Note that we're using a parameterized query by specifying a vector containing the prepared statement string followed by its parameters. This approach is commonly used to prevent SQL injection attacks.

Inserting Records

There are a number of options for inserting records into the database. If you have a map whose keys match the names of the columns in the table, then you can simply use the insert-record function.

```
(defn add-user [user]
  (sql/with-connection db
    (sql/insert-record :users user)))

(add-user {:id "foo" :pass "bar"})
```

If you want to insert multiple records simultaneously, you can use the insert-records function instead.

```
(sql/with-connection db
  (sql/insert-records
    :users
    {:id "foo" :pass "x"}
    {:id "bar" :pass "y"}))
```

We can also use the insert-rows function to specify the records given the values.

```
(defn add-user [id pass]
  (sql/with-connection db
    (sql/insert-rows :users
      [id pass])))
```

The function expects a vector containing the values for each of the columns defined in the table. In case we only want to insert a partial row, we can use insert-values instead.

```
(sql/insert-values :users [:id] ["foo"])
```

The first parameter is the table name. It is followed by a vector specifying the names of the columns to be updated. Lastly, we have another vector containing the values for the columns.

Updating Existing Records

To update an existing record, you can use the update-values and update-or-insert-values functions. The first will require the record to exist in the database, and the second will attempt to update the record and insert a new one if necessary.

```
(sql/update-values
   :users
   ["id=?" "foo"]
   {:pass "bar"})

(sql/update-or-insert-values
  :users
  ["id=?" "foo"]
  {:pass "bar"})
```

Deleting Records

To delete records from the database, we can use the delete-rows function:

```
(sql/delete-rows :users ["id=?" "foo"])
```

Transactions

We use transactions when we want to run multiple statements and ensure that the statements will be executed only if all of them can be run successfully.

If any of the statements throw an exception, then the transaction will be rolled back to the state prior to running any of the statements.

```
(sql/with-connection db
    (sql/transaction
      (sql/update-values
        :users
        ["id=?" "foo"]
        {:pass "bar"})

      (sql/update-values
        :users
        ["id=?" "bar"]
        {:pass "baz"})))
```

Report Generation

In this section we'll cover how we can easily generate reports from the data we collect in our database using the clj-pdf library.[2] Then we'll discuss how to serve the generated PDF to the browser using the appropriate response type.

Our application will have an employee table that will be populated with some sample data. We'll use this data to create a couple of different PDF reports and allow the users to select the type of report they wish to view.

The first thing we'll need to do is configure our database. For this example we'll be using the PostgreSQL database.

Setting Up the PostgreSQL Database

Installing PostgreSQL is very easy. If you're using OS X, then you can simply run Postgres.app.[3] On Linux, you can install PostgreSQL from your package manager. For example, if you're using Ubuntu you can run sudo apt-get install postgresql.

Once installed, we set the password for the user postgres using the psql shell. The shell can be invoked by running the psql command from the console.

```
sudo -u postgres psql postgres
\password postgres
```

With the default user set up we'll create an admin user with the password set to admin.

```
CREATE USER admin WITH PASSWORD 'admin';
```

2. https://github.com/yogthos/clj-pdf
3. http://postgresapp.com/

Then we can create a schema called REPORTING to store our reports by running the following command:

```
CREATE DATABASE REPORTING OWNER admin;
```

Note that we're using the admin user here to save time. You should always create a dedicated user and grant only the necessary privileges for any database you wish to run in production.

With the database configuration out of the way, let's create a new application called *reporting-example* using the compojure-app template.

We'll now open the project.clj file and add the necessary dependencies to it:

```
:dependencies [...
               [postgresql/postgresql "9.1-901.jdbc4"]
               [org.clojure/java.jdbc "0.2.3"]
               [clj-pdf "1.11.6"]
```

Let's start the read-evaluate-print loop (REPL) by running (start-server) in the reporting-example.repl namespace.

With the REPL running, let's create a new namespace called reporting-example.models.db and add our database configuration there.

We'll navigate to the db namespace and create our database connection using clojure.java.jdbc.

reporting-example/src/reporting_example/models/db.clj

```
(ns reporting-example.models.db
  (:require [clojure.java.jdbc :as sql]))

(def db {:subprotocol "postgresql"
         :subname "//localhost/reporting"
         :user "admin"
         :password "admin"})
```

Then we'll make an employee table and populate it with the sample data:

reporting-example/src/reporting_example/models/db.clj

```
(defn create-employee-table []
  (sql/create-table
    :employee
    [:name "varchar(50)"]
    [:occupation "varchar(50)"]
    [:place "varchar(50)"]
    [:country "varchar(50)"]))

(sql/with-connection
  db
  (create-employee-table)
  (sql/insert-rows
    :employee
    ["Albert Einstein", "Engineer", "Ulm", "Germany"]
    ["Alfred Hitchcock", "Movie Director", "London", "UK"]
    ["Wernher Von Braun", "Rocket Scientist", "Wyrzysk", "Poland"]
    ["Sigmund Freud", "Neurologist", "Pribor", "Czech Republic"]
    ["Mahatma Gandhi", "Lawyer", "Gujarat", "India"]
    ["Sachin Tendulkar", "Cricket Player", "Mumbai", "India"]
    ["Michael Schumacher", "F1 Racer", "Cologne", "Germany"]))
```

Finally, we'll write a function to read the records from the table:

reporting-example/src/reporting_example/models/db.clj

```
(defn read-employees []
  (sql/with-connection db
    (sql/with-query-results rs ["select * from employee"] (doall rs))))
```

Let's run read-employees to make sure everything is working as expected. We should see something like the following:

```
(read-employees)

({:country "Germany",
  :place "Ulm",
  :occupation "Engineer",
  :name "Albert Einstein"}
  {:country "UK",
  :place "London",
  :occupation "Movie Director",
  :name "Alfred Hitchcock"}
  ...)
```

You'll notice that the result of calling read-employees is simply a list of maps where the keys are the names of the columns in the table.

Let's see how we can use this to create a table listing the employees in our database.

Report Generation

The clj-pdf library uses syntax similar to Hiccup's to define the elements in the document. The document itself is represented by a vector. The document vector must contain a map representing the metadata as its first element. The metadata is followed by one or more elements representing the document's content.

Let's create a namespace called reporting-example.reports and look at a few examples of creating PDF documents. We'll use the pdf function to create the reports, and the template function to format the input data.

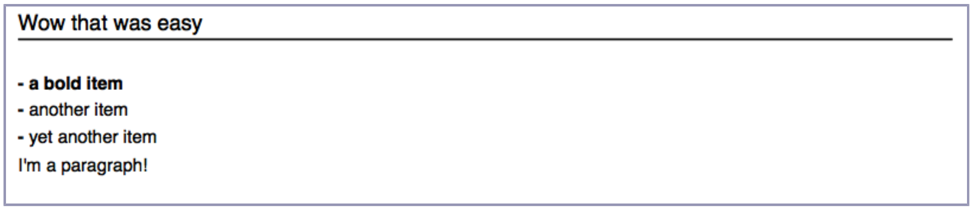
```
(ns reporting-example.reports
  (:require [clj-pdf.core :refer [pdf template]]))
```

The pdf function accepts two arguments. The first can be either a vector representing the document or an input stream from which the elements will be read. The second can be a string representing the output file name or an output stream.

Let’s generate our first PDF by running the following in our reports namespace:

```
(pdf
  [{:header "Wow that was easy"}
   [:list
    [:chunk {:style :bold} "a bold item"]
    "another item"
    "yet another item"]
   [:paragraph "I'm a paragraph!"]]
  "doc.pdf")
```

As you can see, the report consists of vectors, each starting with a keyword identifying the type of element, followed by optional metadata and the content. In the preceding report we have a list that contains three rows, followed by a paragraph. The PDF will be written to a file called doc.pdf in our project’s root. The contents of the file should look like the following figure.


Wow that was easy

- **a bold item**
- another item
- yet another item

I'm a paragraph!

Figure 13—Our first PDF

Next, let’s see how we can use the template macro to format the employee data into a nice table. This macro uses $ to create anchors to be populated from the data using the keys of the same name.

The template returns a function that accepts a sequence of maps and applies the supplied template to each element in the sequence. In our case, since we’re building a table, the template is simply a vector with the names of the keys for each cell in the row. We’ll add the following template to the reporting-example.reports namespace.

```
(def employee-template
  (template [$name $occupation $place $country]))
```

Let's add the reference to our db namespace and try running our template against the database:

reporting-example/src/reporting_example/reports.clj

```
(ns reporting-example.reports
  (:require [clj-pdf.core :refer [pdf template]]
            [reporting-example.models.db :as db]))
```

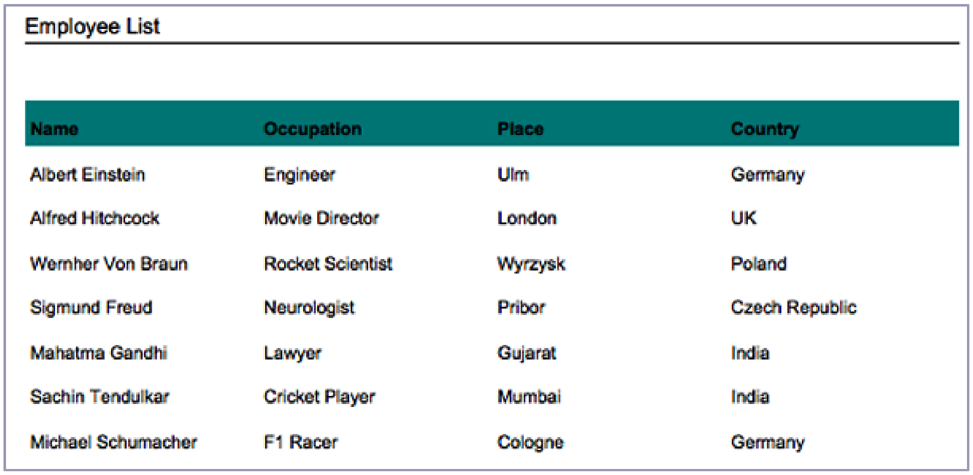
We should see the following output after running (employee-template (take 2 (db/read-employees))) in the REPL:

```
(["Albert Einstein" "Engineer" "Ulm" "Germany"]
 ["Alfred Hitchcock", "Movie Director", "London", "UK"])
```

Looks like our template works as expected. Let's use it to generate a report containing the full list of our employees:

```
(pdf
 [{:header "Employee List"}
  (into [:table
         {:border false
          :cell-border false
          :header [{:color [0 150 150]} "Name" "Occupation" "Place" "Country"]}]
        (employee-template (db/read-employees)))]
 "report.pdf")
```

The resulting report should look like the following figure.



Employee List

Name	Occupation	Place	Country
Albert Einstein	Engineer	Ulm	Germany
Alfred Hitchcock	Movie Director	London	UK
Wernher Von Braun	Rocket Scientist	Wyrzysk	Poland
Sigmund Freud	Neurologist	Pribor	Czech Republic
Mahatma Gandhi	Lawyer	Gujarat	India
Sachin Tendulkar	Cricket Player	Mumbai	India
Michael Schumacher	F1 Racer	Cologne	Germany

Figure 14—Employee table report

Of course, the template we used for this report is boring. Let's look at another example. Here we'll output the data in a list and style each element:

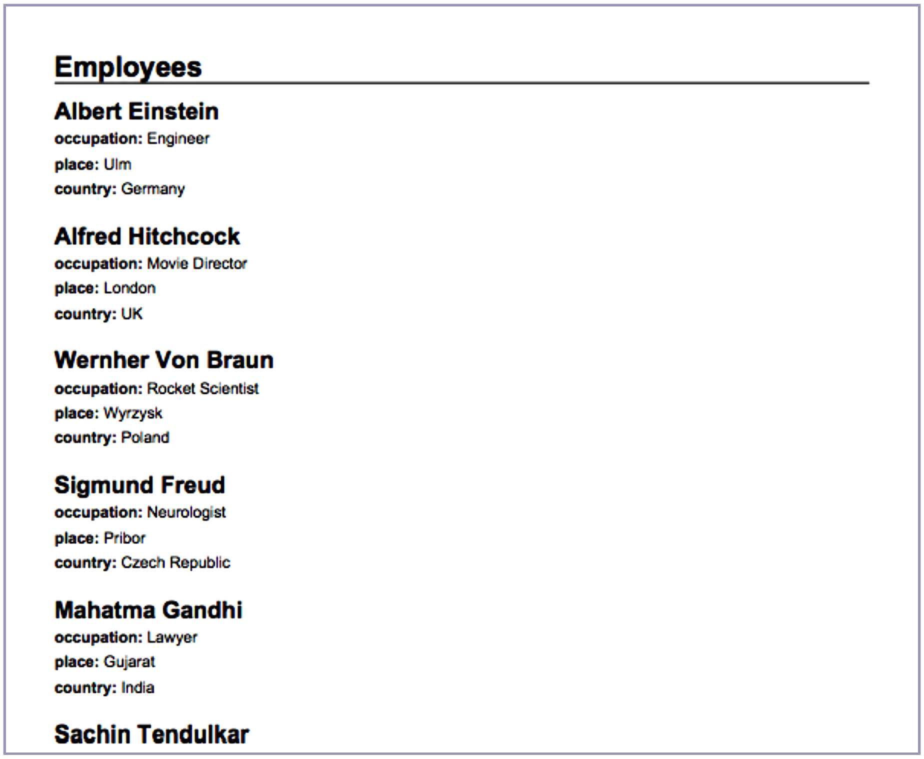
reporting-example/src/reporting_example/reports.clj

```
(def employee-template-paragraph
  (template
    [:paragraph
     [:heading {:style {:size 15}} $name]
     [:chunk {:style :bold} "occupation: "] $occupation "\n"
     [:chunk {:style :bold} "place: "] $place "\n"
     [:chunk {:style :bold} "country: "] $country
     [:spacer]]))
```

Now let's create a report using the employee-template-paragraph by running the following:

```
(pdf
  [{}
   [:heading {:size 10} "Employees"]
   [:line]
   [:spacer]
   (employee-template-paragraph (db/read-employees))]
  "report.pdf")
```

Our new report will look like the following figure.



Employees

Albert Einstein
occupation: Engineer
place: Ulm
country: Germany

Alfred Hitchcock
occupation: Movie Director
place: London
country: UK

Wernher Von Braun
occupation: Rocket Scientist
place: Wyrzysk
country: Poland

Sigmund Freud
occupation: Neurologist
place: Pribor
country: Czech Republic

Mahatma Gandhi
occupation: Lawyer
place: Gujarat
country: India

Sachin Tendulkar

Figure 15—Employee list report

Displaying the Reports

Now that we've created a couple of reports on our data, let's see how we can serve them from our application. We'll write the functions to create a list and table reports using the preceding examples:

reporting-example/src/reporting_example/reports.clj

```
(defn table-report [out]
  (pdf
    [{:header "Employee List"}
     (into [:table
            {:border false
             :cell-border false
             :header [{:color [0 150 150]} "Name" "Occupation" "Place" "Country"]}]
           (employee-template (db/read-employees)))]
    out))

(defn list-report [out]
  (pdf
    [{}
     [:heading {:size 10} "Employees"]
     [:line]
     [:spacer]
   (employee-template-paragraph (db/read-employees))]
    out))
```

Next, we'll navigate to reporting-example.routes.home and add some references needed to generate the report route.

reporting-example/src/reporting_example/routes/home.clj

```
(ns reporting-example.routes.home
  (:require [hiccup.element :refer [link-to]]
            [ring.util.response :as response]
            [compojure.core :refer [defroutes GET]]
            [reporting-example.reports :as reports]
            [reporting-example.views.layout :as layout]))
```

We'll update the home function to provide links to each of the reports:

reporting-example/src/reporting_example/routes/home.clj

```
(defn home []
  (layout/common
    [:h1 "Select a report:"]

    [:ul
     [:li (link-to "/list" "List report")]
     [:li (link-to "/table" "Table report")]]))
```

Now we'll write a function to generate the response. We'll create an input stream using a supplied byte array and set it as the response. We'll also set the appropriate headers for the content type, the content disposition, and the length of the content.

reporting-example/src/reporting_example/routes/home.clj
```
(defn write-response [report-bytes]
  (with-open [in (java.io.ByteArrayInputStream. report-bytes)]
    (-> (response/response in)

        (response/header "Content-Disposition" "filename=document.pdf")
        (response/header "Content-Length" (count report-bytes))
        (response/content-type "application/pdf")) ))
```

We'll write another function to generate the report. This function will create a ByteArrayOutputStream that will be used to store the report. Then it will call one of our report-generation functions with it. Once the report is generated we'll call write-response with the contents of the output stream.

reporting-example/src/reporting_example/routes/home.clj
```
(defn generate-report [report-type]
  (try
    (let [out (new java.io.ByteArrayOutputStream)]
      (condp = (keyword report-type)
        :table (reports/table-report out)
        :list  (reports/list-report out))
      (write-response (.toByteArray out)))

    (catch Exception ex
      {:status 500
       :headers {"Content-Type" "text/html"}
       :body (layout/common
               [:h2 "An error has occured while generating the report"]
               [:p (.getMessage ex)])})))
```

Last but not least, we're going to create a new route to serve our reports.

reporting-example/src/reporting_example/routes/home.clj
```
(defroutes home-routes
  (GET "/" [] (home))
  (GET "/:report-type" [report-type] (generate-report report-type)))
```

You should now be able to navigate to http://localhost:3000 and select a link to one of the reports. When you click on the link the corresponding report will be served.

What You've Learned

This covers the basics of working with relational databases. You've now learned how to do the basic database operations and seen a simple reporting application in action. As we've covered in this chapter, database records are easily mapped to Clojure data structures. Therefore, the Clojure community sees object-relational mapping libraries as unnecessary.

In the next chapter we'll put together all the skills you've learned so far to write a picture-gallery application.