Clojure RECIPES

Julian GAMBLE

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Clojure Recipes

This page intentionally left blank

Clojure Recipes

Julian Gamble

Addison-Wesley

New York • Boston • Indianapolis • San Francisco
Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Gamble, Julian, author.
Clojure recipes / Julian Gamble.
pages cm
Includes index.
ISBN 978-0-321-92773-6 (pbk. : alk. paper)—ISBN 0-321-92773-7
1. Clojure (Computer program language) 2. Software patterns. I. Title.
QA76.73.C565G36 2015
005.1—dc23

2015028332

ISBN-13: 978-0-321-92773-6
ISBN-10: 0-321-92773-7
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, October 2015

❖

To my amazing wife Jo-Ann . . .

. . . You Rock!

❖

This page intentionally left blank

Contents

Preface

Who This Book Is For

Clojure Recipes is for people who have started on their journey into Clojure but haven't quite found their feet. Ideally, you are aware that Clojure has lots of parentheses, but the prospect of integrating some libraries to build a working project is still a bit daunting.

If you're comfortable with Clojure and feel like you could easily build a project in a weekend without assistance, then this book is still useful. This is the book you give to someone in the office who is just curious, or who has seen Clojure and wants to get started but needs a helping hand.

Finally, if you're a pragmatist who just needs to get some working code running, then this book is for you. There are lots of examples for you to copy and paste to get your project working.

What This Book Is Not

Clojure Recipes is not an "introduction to Clojure" book. There are some really brilliant books and online resources that cover this topic area. If you want an in-depth explanation of Clojure, then read one of those. This is a "learn by doing" type of book.

What This Book Is About

Clojure Recipes is about "the weekend project." It's about getting something running in a short amount of time. The book makes no assumptions about background knowledge of Clojure, but provides all you need in packaged bites.

The aim of the book is to provide self-contained projects that would have "just enough" for you to get running and see all the pieces hang together. The idea is that you can tweak and extend these projects for your needs.

Why Clojure?

So why should you use Clojure for your next project? Here are a few reasons:

- Clojure focuses on isolating side effects. Whether it is regular business logic or a concurrency scenario, immutability makes your program easier to reason about

and cancels out a whole class of bugs due to state mutation. You can still modify state, but the language encourages you to use it only where necessary.

- Clojure was one of the first languages to make serious use of the ideas in Chris Okasaki's book *Purely Functional Data Structures* (Cambridge University Press, 1999). The big revelation in that book was that data structures implemented in a functional way could be done with an upper bound of time O(n) on costs for reads and writes.
- The benefit of these purely functional data structures was to enable another way to think about Software Transactional Memory (STM) in your program. Clojure introduces constructs like Multiversion Concurrency Control (MVCC) at the application level that previously developers typically only relied upon at the database level.
- Clojure is great at concurrency. In Clojure it is easy to reason about how your program will behave in a concurrency scenario even when there are multiple processes making changes to the one data structure.
- Clojure is pragmatic. Clojure runs on the JVM (in addition to the .NET CLR and JavaScript execution environments). This brings a wealth of libraries from the Java world that can be reused. What's more, when you deploy it, it can look like a jar file so there is no need to tell anyone you're using Clojure at all!
- Clojure has great tooling. Because Clojure runs on the JVM, much of the tooling associated with the JVM for deployment and monitoring is still available to you in the Clojure domain.
- Clojure is a Lisp—one of the oldest programming languages around. This brings a rich heritage of distinctive problem-solving styles and the wisdom of many graybeards who have been chipping away at computing problems for many years.
- Clojure is great at Domain Specific Languages (DSLs). Clojure makes it easy to define languages specifically targeted to the problem you're dealing with. You can easily represent your problem in a completely new way.
- Clojure is fast for developing an end product. Clojure gives you the ability to incrementally compile your program at the REPL, to experiment with it, and to interact with it. With hundreds of functions in the Core libraries, plus access to all the JDK libraries, developers can get a lot done.
- Clojure enables Lean Software Development because, more than other languages, it allows you to delay making decisions about the structure of your program. This comes from a focus on Composition over Inheritance in its idiomatic style, as well as from being able to start writing your program without tying yourself down to a particular expression of the core data.
- Clojure is dynamically typed. Clojure keeps track of the type at run time, so the programmer doesn't have to. This point is enormously controversial at present for three reasons:

1. Types enable you to reason about the behavior of your program in a large system at Compile time. For this reason Clojure enables gradual typing via the `core.typed` library. This way you can add types to your Clojure program as you need it.
2. Clojure is built to interact with Java, which is typed. The Clojure interop with Java allows type hints and type inference.
3. Much talk is still being made about Philip Wadler's papers from the early 1990s on Types and Monads. In particular, proponents claim that Wadler's ideas enable a compiler-based approach to Edsger Dijkstra's claim that "[m]uch of the essence of building a program is in fact the debugging of the specification." Clojure answers this third reason for controversy in two ways. First, Clojure borrows much from the Haskell language and so in a way pays it great homage. Second, the Clojure community values simplicity, probably to the extent that you should know what you want your program to do and whether it is correct, rather than relying on a Compiler to tell you that. (But of course Clojure has a limited ability to use Monads if that's what you really want.)

- Clojure has the REPL, which is an enormously powerful tool for rapid prototyping, testing, and making changes to your program while it is running.
- Clojure has uniform syntax (S-expressions), where the primary representation of the program is a data structure in the language itself (sometimes called *homoiconic*). This might not seem like a big deal, or you might find all the parentheses enormously obnoxious. (Regarding the latter point, several people have suggested that Clojure has fewer parentheses than Java for the equivalent code!) The real benefit of uniform syntax is macros. You can transform and generate Clojure code at Compile time or at run time. One might wonder at the fuss being made over "code as data" until you contrast it with other languages and see the excitement that comes when a new language syntax feature is introduced. Nine times out of ten, that language syntax feature could be implemented as a macro in Clojure. (A key example being `core.async`.)
- Clojure is highly expressive and extensible. What does the expressiveness of a programming language even refer to? The claim is that you can achieve more with fewer lines of code. Paul Graham made particularly high claims with his Arc challenge. Whether his Arc language was superior was never really answered, but his broader point was that homoiconic languages have a natural advantage in being able to express more ideas in fewer lines of code with the power of homoiconic syntax and macros.
- The Clojure community has been enormously innovative. Particular projects of note have been the following:
  - Cascalog—a DSL to generate Hadoop queries with far fewer lines of code.
  - Storm—a highly available, distributed system for processing real-time data.
  - Datomic—a distributed, no-SQL database with point-in-time reproduction of all data, even after "updates."

  - ClojureScript—a JavaScript generator using Clojure syntax enabling DOM transformations and better ways to solve the "callback hell" problem. This is now self-hosting.
  - `core.typed`—enables "gradual typing" for your application so you can add types as required in order to reason about your program.
  - `core.async`—restructures code to provide an inversion of the calling paradigm, allowing the developer to avoid "callback hell," to better enable the processing of multiple real-time threads of information, and complementing the Clojure implementation with similar syntax and functionality in the browser with ClojureScript.
- Clojure allows lots of different programming styles. Of course, it is idiomatic to write immutable functional code in Clojure. But you could write in a stack-based coding style like FORTH if you wanted. Clojure does have some ability to provide compile-time guarantees via Monads. It is even possible to write procedural blocks of code that mutate state. Clojure is flexible.

So take a look at Clojure! It will be great for your next project.

Coding Conventions Used in This Book

Sometimes we run a command on the command line. It looks like this:

```
lein new myproject
```

Then sometimes we show the result of running a command on the command line. It looks like this:

```
Generating a project called myproject based on the 'default' template.
```

Sometimes we show some code in a Clojure file. It looks like this:

```
(ns myproject)

(defn -main[& args]
 (prn "running"))
```

Sometimes we show a command run on a Clojure REPL. It looks like like this:

```
user=> (prn "Hello World")
```

Then sometimes we show the result of running a command on the Clojure REPL. It looks like this:

```
"Hello World"
```

Errata

There is an errata page on the Clojure Recipes website here:
http://clojurerecipes.net/errata.html

About the Author

Julian Gamble is a software engineer who has worked in the financial services industry for more than a decade. When he's not enabling billions of dollars to orbit the globe, he writes and presents on all things software related at juliangamble.com/blog. He lives in Sydney, Australia.

This page intentionally left blank

3

Creating a REST Server in Compojure

In this chapter we will build the first of two REST servers. This is a simple one to do in Compojure. The REST server we build will receive a REST call over http. We will test it using a command line tool, `curl`.

Assumptions

In this chapter we assume the following:

- You have Leiningen installed and on your path on the command line.
- You know how to use `curl` (whether on a Mac or a PC) and have it on your path.
- You understand the concept of http parameters being passed via a GET request URI and via a POST form parameters request.

Benefits

IT organizations now are filled with demands to build backend services implemented in JavaScript for mobile devices like iPhones and rich web clients. Both mobile and rich JavaScript clients work very well with services that implement REST. A great opportunity for you to get Clojure into your organization is by whipping up a REST service in Clojure.

The Recipe—Code

1. Create the project using Leiningen and the Compojure template:

   ```
   lein new compojure rest-demo
   ```

2. Modify the `project.clj` to have the following contents:

```
(defproject rest-demo "0.1.0-SNAPSHOT"
  :min-lein-version "2.0.0"
  :dependencies [[org.clojure/clojure "1.7.0-beta2"]
                 [compojure "1.3.4"]
                 [ring/ring-defaults "0.1.5"]]
  :plugins [[lein-ring "0.9.5"]]
  :ring {:handler rest-demo.handler/app}
  :profiles
  {:dev {:dependencies [[javax.servlet/servlet-api "2.5"]
                        [ring/ring-mock "0.2.0"]]}})
```

3. Ensure that the file `rest-demo/src/rest_demo/handler.clj` looks like this:

```
(ns rest-demo.handler
  (:require [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults
            :refer [wrap-defaults site-defaults]]))

(defn handle-http []
  (context "/:id" [id]
    (defroutes api-routes
      (GET    "/" [] (str "get called: " id "\n"))
      (POST   "/" {form-params :form-params}
           (str "post called: " id "\n" form-params " \n"))
      (PUT    "/" req (str "put called with params: " req))
      (DELETE "/" [] (str "delete called: " id "\n")))))

(defroutes app-routes
  (handle-http)
  (route/not-found (str
   "This is the default page - try "
   "<a href='http://localhost:4000/33'>this</a>\n")))

(def app
  (wrap-defaults app-routes
    (assoc-in site-defaults [:security :anti-forgery] false)))
```

Testing the Solution

Let's give it a run.

1. In the command prompt, change back to the parent directory of your project and start the server using Leiningen:

```
lein ring server-headless 4000
```

Note that we are using the `server-headless` parameter to the `lein ring` command. This starts the server without opening a web browser. Had we merely run (as a hypothetical) `lein ring server 4000,` a new web browser would have opened. (But because we're about to use `curl` to interact with the website, a browser would have gotten in the way.)

2. Open a new command window and type:

   ```
   curl http://localhost:4000/1
   ```

   You should get a response like:

   ```
   get called: 1
   ```

3. Now enter:

   ```
   curl -X DELETE http://localhost:4000/4
   ```

   You should get a response like:

   ```
   delete called: 4
   ```

4. Now enter:

   ```
   curl -X POST -d "id=2" http://localhost:4000/3
   ```

   You should get a response like:

   ```
   post called: 3
   {"id" "2"}
   ```

5. Now enter:

   ```
   curl -X PUT -d "id=2" http://localhost:4000/3
   ```

   You should now get something similar to:

   ```
   put called with params: {:ssl-client-cert nil, :remote-addr
   "0:0:0:0:0:0:0:1%0", :scheme :http, :query-params {}, :context "/3",
   :form-params {"id" "2"}, :request-method :put, :query-string nil,
   :route-params {:id "3"}, :content-type "application/x-www-form-urlencoded",
   :path-info "/", :uri "/3", :server-name "localhost", :params {:id "3", "id"
   "2"}, :headers {"user-agent" "curl/7.27.0", "content-type" "application/
   x-www-form-urlencoded", "content-length" "4", "accept" "*/*", "host"
   "localhost:4000"}, :content-length 4, :server-port 4000, :character-encoding
   nil, :body #<HttpInput org.eclipse.jetty.server.HttpInput@cfefc0>}
   Julians-MacBook-Pro:~
   ```

   You can see a large amount of information is in the parameter map `req`. This also demonstrates the powerful, dynamic, interactive nature of running Compojure with the Leiningen plug-in. You can modify the file and save it, and your changes are accessible instantly to the web browser. (We could also have done all of this on the REPL—but we'll save that for another day.)

Notes on the Recipe

The Compojure library is designed to make RESTful URIs easy to work with. Note in particular the `project.clj` file:

```
            [ring/ring-defaults "0.1.5"]]
  :plugins [[lein-ring "0.9.5"]]
  :ring {:handler rest-demo.handler/app}
```

Notice the `ring-defaults` library. We'll use this when we examine parameters passed in. Also note the `lein-ring` plug-in. This enables us to start the app from the command line. It will also enable us to modify the app when it is running and to see the results without restarting the server.

Also note the `:ring {:handler...` syntax. This points to the part of the application that will handle the incoming requests.

Now look at the file `handler.clj`, in particular the namespace:

```
(ns rest-demo.handler
  (:require [compojure.core :refer :all]
            [compojure.route :as route]
            [ring.middleware.defaults
            :refer [wrap-defaults site-defaults]]))
```

Note that we load the Compojure libraries for handling routes, and we use the `ring.middleware` for parameter handling.

Now observe the `handle-http` function definition:

```
(defn handle-http []
  (context "/:id" [id]
    (defroutes api-routes
      (GET    "/" [] (str "get called: " id "\n"))
      (POST   "/" {form-params :form-params}
        (str "post called: " id "\n" form-params " \n"))
      (PUT    "/" req (str "put called with params: " req))
      (DELETE "/" [] (str "delete called: " id "\n")))))
```

This function handles the parameters for the particular http request type. Here we just do simple handlers for the different http request types. This handles requests in the format:

```
http://servername/2
```

Now note the following function:

```
(defroutes app-routes
  (handle-http)
  (route/not-found (str
  "This is the default page - try "
  "<a href='http://localhost:4000/33'>this</a>\n")))
```

This is the main route-handler function. It delegates most of its responsibilities to our `handle-http` function above. If that returns `nil`, then it displays a `"Not Found"` response.

Now note the following symbol:

```
(def app
  (wrap-defaults app-routes
    (assoc-in site-defaults [:security :anti-forgery] false)))
```

This is the application hook. We point to this in the `project.clj` file. It takes the route definitions in `app-routes` and feeds that into the function result of `wrap-defaults`. The `wrap-defaults` function adds middleware to the route to enable URI parameter input. We switch off the `anti-forgery` middleware so our simple `curl` tests will work. You shouldn't do this in your production application.

Conclusion

We've seen an example of generating RESTful HTTP requests from the command line using the `curl` command, and we implemented handlers for these requests in Compojure.

This page intentionally left blank

Index

D

E

F

G

H

I

J

K

L

M

T

W

Y

Z

This page intentionally left blank

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account. You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.