Develop your own web application with the effective use of the Clojure programming language

Clojure Web Development Essentials

Functional programming enables developers to create and deploy applications quickly and easily. Using the Clojure programming language and the Leiningen build tool, this book teaches you how to leverage various Clojure libraries into the building blocks of a web-based application.

We'll start by generating a new, fully runnable, baseline web application using the Leiningen Luminus template. From there we'll examine the different pieces that comprise our application, and learn the basics of the underlying Ring framework. Subsequent chapters will focus on each technology and leverage those technologies to build additional functionality into our application. You'll learn how to validate user submitted data, authentication, account creation, how to interact with a database, and more. Finally, we'll create an environment configuration and deploy our app.

By the end of this book you'll have created a full fledged application from start to finish, and you'll have the requisite skills to develop an efficient and robust web application.

Who this book is written for

This book is for anyone who's worked with Clojure and wants to use it to start developing applications for the Web. Experience or familiarity with basic Clojure syntax is a must, and exposure to Leiningen (or other similar build tools such as Maven) would be helpful.

What you will learn from this book

- Generate a fully runnable web application using the Luminus Leiningen application template
- Explore the basics of the underlying Ring framework and the Ring Server
- Configure URL Routing, Logging, and some testing basics
- Create new web pages using the Selmer template rendering library
- Validate user-submitted form data
- Store and retrieve data to and from a database
- Configure, package, and deploy the finished application

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 "Getting Started with Luminus"
- A synopsis of the book’s content
- More information on Clojure Web Development Essentials

About the Author

Ryan Baldwin is a theatre major turned computer science geek. Hailing from the prairies of Western Canada, Ryan has been developing software on a wide array of platforms and technologies since 2001. Once, he wrote a crazy system application that compiled XSD Schema Docs into XAML forms that performed two-way binding with underlying XML documents in .NET WPF. Why? Because it had to be done. Another time, he worked on a project that mashed many social networks into one gigantic thing that essentially allowed users to find out all of their indirect connections. It was eventually shelved.

In 2012, he relocated to Toronto, where he works with the University Health Network, developing systems and tools that facilitate patient information exchange. You can often find him wearing headphones and jittering in coffee shops.

I'd like to thank Packt for giving me this opportunity and Dmitri Sotnikov for pushing me to do it. Without either of you, I probably would have gotten a lot more sleep and a lot less experience out of life. I'd also like to thank Chris Kay Fraser, without whose support and vegan brownies I would have never had the confidence to pursue such a project. I'd finally like to thank my family, friends, colleagues, and anybody else who interacted with me over these past several months; I'll buy you a "thank you" beer for putting up with me. All my reviewers who took the time to read, recheck, and provide essential feedback, I owe all of you at least a pitcher of beer (hit me up next time you're in Toronto). And, of course, I'd like to thank you, dear readers; without you, none of this would have happened. I am both humbled and terrified of you.
Clojure Web Development Essentials

Clojure is a beautiful, concise language, and its adoption for web applications is ready and about to explode. In **Clojure Web Development Essentials**, you will learn how to build a Clojure web application from scratch using the Leiningen build tool and the Luminus application template. We'll start by creating a simple example application in the first few pages of the first chapter, and build on that application with each subsequent chapter. We'll cover URL routing, template rendering, database connectivity, form validation, and everything else we need to build a typical web app. By the end of this book, you'll have the knowledge required to venture into the world of web development, and you'll be able to use your skills for the betterment of the Internet.

**What This Book Covers**

**Chapter 1, Getting Started with Luminus**, guides you through creating a new project using the Luminus application template. We'll then dive into what was generated, what the out-of-the-box project dependencies are, and the general file structure of a Luminus web app.

**Chapter 2, Ring and the Ring Server**, describes the core technologies driving our application, and shows you how to use the development web server.

**Chapter 3, Logging**, demonstrates configuration of some basic logging and the Clojure logging library, Timbre.

**Chapter 4, URL Routing and Template Rendering**, starts to dive into the important part of web applications. It shows you how to handle incoming requests using Compojure, and how to render web pages using the Selmer templating engine. We'll also create a sign-up form for our application.

**Chapter 5, Handling Form Input**, teaches you how to validate form data and report form validation errors back to the user.

**Chapter 6, Testing in Clojure**, is a quick tour of automated testing and its use in Clojure.

**Chapter 7, Getting Started with the Database**, is the first of three chapters covering database management and interactivity. We'll set up our application's database, and you will learn how to manage your database schema using the Migratus Leiningen plug. Then we will store the form input created in the fifth chapter using YeSQL.

**Chapter 8, Reading Data from the Database**, continues exploring database interactivity by teaching you how to retrieve data from the database using YeSQL. We'll then create a couple of new web pages that list the most recently added items in our database.
Chapter 9, Database Transactions, gives us a brief overview of what database transactions are. We'll then create a form that transactionally inserts data into multiple tables.

Chapter 10, Sessions and Cookies, demonstrates how sessions and cookies are managed and maintained in Noir. We'll then create an authentication form for our application, and save a cookie in the user's browser to remember their username the next time they log in.

Chapter 11, Environment Configuration and Deployment, guides us through abstracting our environment configuration (such as database connectivity) and describes a few common ways by which we can deploy our application.

Appendix, Using Korma – a Clojure DSL for SQL, covers the modification of the YeSQL model layers to use Korma, a native Clojure Domain Specific Language that can be used to interact with the database if you're not keen on using raw SQL.
Getting Started with Luminus

Ah, getting started! This chapter introduces you to the foundations of Clojure web development using Luminus, a popular web application template for Leiningen. In this chapter, you will:

- Generate a new web application using the Luminus Leiningen template
- Get an introduction to the popular libraries, which Luminus uses to handle the various aspects of a web application, and what those libraries do
- Get an overview of the directory structure generated by Luminus
- Learn how to fire up the web application on your development machine

In this chapter, we'll create a new web application called hipstr, an application that will help us track our vinyl collection and endow us with obscure credibility. We'll build this application with each subsequent chapter by creating our own route handlers, interacting with a database, authenticating users, validating form input, and reading/writing cookies. By the end of this book, we'll know the Clojure web basics well enough that we'll be wearing plaid shirts and sipping bourbon aged in casks from a place nobody's ever heard of.

Leiningen

Our project will rely heavily on Leiningen, a build and task tool for Clojure. Leiningen allows us to easily maintain our application's dependencies, assists us in common tasks such as database migrations, running tests, producing binaries (jars and wars), and a plethora of other things. Leiningen is akin to Java's build tool Maven (http://maven.apache.org), and Ruby's Rake (http://github.com/jimweirich/rake). As Leiningen's web page (http://leiningen.org) concisely puts it: for automating Clojure projects without setting your hair on fire.
Getting Started with Luminus

If you haven't already installed Leiningen 2.x, head over to http://leiningen.org/#install and follow the four simple instructions. It will take just 60 seconds, and the world of Clojure will become your oyster.

After you've installed Leiningen, you'll have access to a new command in your terminal, lein. Invoking this command will invoke Leiningen.

**Using Leiningen**

The basic makeup of a Leiningen task can be summarized as follows:

```bash
# lein $TASK $TASK_ARGUMENTS
```

In the preceding shell pseudo-command, we invoke Leiningen using its binary. The `lein $TASK` argument is the Leiningen task we want to execute (such as `install`, `jar`, etc.), and `$TASK_ARGUMENTS` is any information required for that task to do its job, including additional subtasks and the arguments for a given subtask. You can see a full list of the available tasks in Leiningen by executing the following command:

```bash
# lein --help
```

You can also view the help content for a specific Leiningen task by executing the following command:

```bash
# lein help $TASK
```

You can use these commands whenever you need to know how to do something in Leiningen.

**Generating the application**

Leiningen can generate an application skeleton (or scaffolding) from a plethora of different templates. There's a template for nearly everything such as `clojurescript` projects, web applications (of course), and much more.

To generate a new application, we use the `new` Leiningen task whose basic syntax is as follows:

```bash
# lein new [$TEMPLATE_NAME] $PROJECT_NAME
```
The new task expects, at a minimum, a name for the project ($PROJECT_NAME). Optionally, we can provide a specific template to use ($TEMPLATE_NAME). If we don't specify a template, then lein will use the default template, which is a general template for developing libraries.

For our project we'll use the Luminus template, an excellent template for web applications. Luminus generates a project and wires in the libraries to support pretty much every aspect of web development including sessions, cookies, route handling, and template rendering.

At the time of this writing, the Luminus template was at version 1.16.7. To ensure the code examples in this book work, you can force Leiningen to use a specific version of Luminus by modifying Leiningen's profiles.clj file (typically found in your home directory, in a folder called .lein) to include the specific version of Luminus. For example:

```
:user {:plugins [[luminus/lein-template "1.16.7"]]}  
```

This modification will ensure that version 1.16.7 of the Luminus template is used when generating a Luminus-based application.

Just try the following command:

```
# lein new luminus hipstr  
>> Generating a lovely new luminus project named hipstr...
```

The preceding command will generate a fully runnable application in a directory called hipstr. You can run the application by using cd hipstr to enter into the hipstr directory and then execute the following command:

```
# lein ring server  
>>(Retrieving im/chit/cronj/1.0.1/cronj-1.0.1.pom from clojars)  
>>...a whole bunch of Retrieving...  
>>...and other output...  
>>Started server on port 3000
```
In the preceding command line, the `lein ring server` command updates our class path with the dependencies required to compile and run the app. It then launches the development server (an embedded Jetty server) and starts serving on port 3000. Lastly, it launches our default web browser and navigates to the root page.

The subsequent output of `lein ring server` is a series of debug statements that lets us know what the heck is going on during the startup process. Any generated exceptions or problems that occur while attempting to launch the application will be emitted as part of this output.

Getting help

If anything doesn't go as planned, or you're stumped and confused, feel free to check the Luminus documentation at http://www.luminusweb.net. You can also get some help from people in the Luminus community (https://groups.google.com/forum/?fromgroups#!forum/luminusweb) or the Ring community (https://groups.google.com/forum/?fromgroups#!forum/ring-clojure). Of course, there's always the Clojure group on Google Groups (https://groups.google.com/forum/#!forum/clojure).

Dependencies of the app

The Luminus template provides good starting defaults for a typical web application by using popular libraries. It also configures common tasks (such as logging) and provides a few default route handlers (URL handlers).

Taking a peek at the generated project.clj file, we see all the dependencies included by the luminus template. At the time of writing, the project.clj file produced the following dependencies:

```clojure
:dependencies [[org.clojure/clojure "1.6.0"]
               [lib-noir "0.9.4"]
               [ring-server "0.3.1"]
               [selmer "0.7.2"]
               [com.taoensso/timbre "3.3.1"]
               [com.taoensso/tower "3.0.2"]
               [markdown-clj "0.9.55"]
```

Luminus is a popular and active project, and is constantly getting better. Between now and the time this book goes to press and you purchasing one for each of your friends and yourself, it’s possible that the template will have changed. At the time of writing, version 1.16.7 of the luminus template was used. If you used a more recent version your results may vary.

The first dependency should look familiar (if not, then this book isn't for you... yet). The rest, however, might appear to be a mystery. I'll spare you the effort of searching it online and break it down for you.

- **lib-noir:** This contains a slough of useful utilities to create web applications using the Ring framework, such as routing, redirections, static resources, password hashing, file uploads, sessions and cookies, and so on. It's the workhorse for much of the plumbing common to all web applications. Visit the following website: https://github.com/noir-clojure/lib-noir.

- **ring-server:** This is a bit of an omnibus library, encompassing several other Ring-related libraries. Ring is a web application library, which acts as an abstraction between our web application (hipstr) and the underlying web server or servlet container. You can think of it as something akin to Java's Servlet API (which Ring fulfills), Python's WSGI, or Ruby's Rack. Ring Server, by contrast, is a library that starts a web server capable of serving a Ring handler. We'll get into more detail in Chapter 2, Ring and the Ring Server. To get more information about Ring Server, visit: https://github.com/weavejester/ring-server

- **selmer:** This is an HTML template rendering a library modeled after the ubiquitous Django framework. Selmer allows us to generate dynamic pages, script loops and conditional rendering, extend other Selmer templates, and so on. We'll talk more about Selmer in Chapter 4, URL Routing and Template Rendering. To get more information on selmer, visit: https://github.com/yogthos/Selmer

- **timbre:** Timbre is a pure Clojure logging library. It's pretty much like every other logging library on the planet, complete with somewhat confusing configuration. We'll cover Logging in Chapter 3, Logging. You can also visit https://github.com/ptaoussanis/timbre, to get more information on Timbre.
• tower: This is similar to its sibling timbre, and is a pure Clojure library that provides support for internationalization and localization. You can refer to https://github.com/ptaoussanis/tower.

• markdown-clj: This is a simple library that allows us to compile markdown to html. For more information, you can visit https://github.com/yogthos/markdown-clj.

• environ: This allows us to create different application configurations for different environments (think development versus production). We’ll work with environ in Chapter 11, Environment Configuration and Deployment.

• cronj: This is a simple, straightforward library for creating cron-like scheduled tasks. To know more about cronj, visit https://github.com/zcaudate/cronj.

• noir-exception: This provides prettified, rendered, exception stacks in the browser as well as to log files. The noir-exception library highlights your application’s namespaces in their own color, easily separating your called code from the rest of the first and third party Clojure libs.

• prone: This produces the most amazing exception reporting output you might have ever seen. (https://github.com/magnars/prone).

Luminus file structure
The luminus template generates web applications using a fairly typical directory structure. However, it also produces a number of Clojure namespaces that can cause a bit of confusion if you’re brand new to Clojure web development. You can either open the project using your favorite Clojure editor, or do the following from the terminal:

```bash
# find . -print | sed -e 's;[^/]*;|____);g;s;____|; |;g'
```

The preceding command line is a nasty thing to eyeball and type. You can copy and paste the preceding command from http://bit.ly/1F3TmdJ.
In either case, you should see output similar to the following:

```
bigfoot:hipster ryanbaldwin$ find . -print | sed -e 's;[\r\n]/; |\;

devignore
__Procfile
__project.clj
__README.md
__resources
  __public
    __css
      __screen.css
    __fonts
      __glyphicons-halflings-regular.eot
      __glyphicons-halflings-regular.svg
      __glyphicons-halflings-regular.ttf
      __glyphicons-halflings-regular.woff
    __img
      __js
      __md
  __docs.md
templates
  __about.html
  __home.html
__src
  __hipster
    __handler.clj
    __layout.clj
    __middleware.clj
    __repl.clj
    __routes
    __home.clj
    __session_manager.clj
    __util.clj
__test
  __hipster
    __test
      __handler.clj
bigfoot:hipster ryanbaldwin$ cd ..
```

Luminus generates three directories at the root of the application directory: `resources`, `src`, and `test`.

The `resources` directory contains the files that will compose the front end of our applications. The `public` folder contains resources publicly available to the client, such as our JavaScript, CSS, and images. By contrast, the `templates` directory contains our Selmer templates used for the heavy rendering of HTML parts. All of these files will be made available on our class path; however, only those in the `public` folder will be actually available to the client.
The `src` directory contains all of the necessary namespaces for running our application, and the `test` directory contains all the necessary namespaces for testing our `src`.

In addition to the directories, however, Luminus also generated some files in the `src` directory. These files are the bare minimum requirement to successfully run our application, and each one handles specific functionality. Let’s take a brief look at the base functionality contained in each file.

### util.clj

The `hipstr.util` namespace is a simple namespace where you can put various helper functions you find yourself frequently using during the development of your application. Out of the box, Luminus generates a `hipstr.util` namespace with a single function, `md->html`, which converts markdown into HTML. Typically, I try to avoid namespaces such as `util.clj` because they eventually turn into the junk drawer in your kitchen, but they can be useful on smaller projects if things don’t get too crowded. The following block of code shows the `hipstr.util` namespace:

```clojure
(ns hipstr.util
 (:require [noir.io :as io]
 [markdown.core :as md]))
(defn md->html
 "reads a markdown file from public/md and returns an HTML string"
 [filename]
 (md/md-to-html-string (io/slurp-resource filename)))
```

### session_manager.clj

One of lib-noir’s exposed functionalities is session management (which we’ll discuss in detail in Chapter 10, *Sessions and Cookies*). The default session pool in Luminus is an in-memory session pool, a shortcoming of which is that expired sessions are only removed from memory when the server handles a request associated with an expired session. As a result, old stale sessions can linger in memory indefinitely, straining memory resources on the server. Luminus boilerplates a `cronj` job in the `hipstr.sessions-manager` namespace, which occasionally removes stale, unused sessions. By default, the job runs every 30 minutes. Take a look at the following lines of code:

```clojure
(ns hipstr.session-manager
 (:require [noir.session :refer [clear-expired-sessions]]
 [cronj.core :refer [cronj]]))
```
(def cleanup-job
  (cronj
   :entries
   [[:id "session-cleanup"
     :handler (fn [_ _] (clear-expired-sessions))
     :schedule "* /30 * * * *
     :opts {}]]

layout.clj

The hipstr.layout namespace houses the functions that are used to render the HTTP response body. By default, Luminus creates a single function, render, which will render any Selmer template onto the HTTP response. The following lines of code is for the hipstr.layout namespace:

(ns hipstr.layout
  (:require [selmer.parser :as parser]
            [clojure.string :as s]
            [ring.util.response :refer [content-type response]]
            [compojure.response :refer [Renderable]]
            [environ.core :refer [env]])

(def template-path "templates/")

(deftype RenderableTemplate [template params] Renderable
  (render [this request]
    (content-type
      (->> (assoc params
        (keyword
          (s/replace template #".html" "-selected")"active"
        :dev (env :dev)
        :servlet-context
        (if-let [context (:servlet-context request)]
          ;; If we're not inside a servlet environment
          ;; (for example when using mock requests), then
          ;; .getContextPath might not exist
          (try (.getContextPath context)
           (catch IllegalArgumentException _
             context))))
      (parser/render-file (str template-path template))
      response))

[15]
Getting Started with Luminus

"text/html; charset=utf-8")

(defn render [template & [params]]
  (RenderableTemplate. template params))

The key to the hipstr.layout namespace is that it remains high level and generic. You should avoid writing functions with domain knowledge in this namespace, and instead focus on generating response bodies. If you put an explicit URL or filename in this namespace, you're probably doing it wrong.

middleware.clj

Middleware, for the unfamiliar, is a function that can work with an incoming request prior to the request being handled by the main application (that is our proverbial business logic). Its function is similar to how a car moves through an assembly line; each employee working the line is responsible for interacting with the car in some specific way. Much like how at the end of the assembly line the car is in its final state and ready for consumption, so is the request in its final state and ready for processing by the main application. The following code is for the hipstr.middleware namespace:

(ns hipstr.middleware
  (:require [taoensso.timbre :as timbre]
           [selmer.parser :as parser]
           [environ.core :refer [env]]
           [selmer.middleware :refer [wrap-error-page]]
           [prone.middleware :refer [wrap-exceptions]]
           [noir-exception.core :refer [wrap-internal-error]]))

(defn log-request [handler]
  (fn [req]
    (timbre/debug req)
    (handler req)))

(def development-middleware [wrap-error-page wrap-exceptions])

(def production-middleware [#(wrap-internal-error % :log (fn [e] (timbre/error e)))]

(defn load-middleware []
  (concat (when (env :dev) development-middleware)
          production-middleware))
The hipstr.middleware namespace has two primary responsibilities. The first is that it ties together all the different middleware we want across any of our runtime environments. The second is that it gives us a place to add additional middleware, if desired. Of course, there's nothing prohibiting us from writing our middleware in a new namespace, but for the sake of simplicity and for this book, we'll simply create additional middleware in the hipstr.middleware namespace.

routes/home.clj

One of the directories that Luminus generated was a route folder. Routes are what tie a request to a specific handler (or, in layman's terms, a chunk of code to be executed based on the URL the request is sent to). Luminus generates 2 routes for us:

- A / route, which renders the result of calling the home-page function, which ultimately renders the home page you see at startup
- A /about route, which renders the result of the about-page function, responsible for rendering the about.html page

Take a look at the following lines of code:

```clj
(ns hipstr.routes.home
 (:require [compojure.core :refer :all]
          [hipstr.layout :as layout]
          [hipstr.util :as util]))

(defn home-page []
  (layout/render
   "home.html" {:content (util/md->html "/md/docs.md")}))

(defn about-page []
  (layout/render "about.html"))

(defroutes home-routes
  (GET "/" [] (home-page))
  (GET "/about" [] (about-page)))
```

We will create a couple of our own routing namespaces over the course of this book. The routes we'll create in those namespaces will follow the same pattern demonstrated in the preceding hipster.routes.home namespace. We'll talk a bit more about routes in Chapter 4, URL Routing and Template Rendering.
handler.clj

Everything we’ve seen in this chapter is brought together into a single, harmonious, running application in the hipstr.handler namespace, explained in the following lines of code. Opening the file for a cursory scan reveals our cron job to clean up expired sessions, the home-routes from the hipstr.routes.home namespace, the configuration of our Timbre logging, and so on.

(ns hipstr.handler
 (:require [compojure.core :refer [defroutes]]
  ; ... snipped for brevity ...
  [cronj.core :as cronj]]))

(defroutes base-routes
  (route/resources "/")
  (route/not-found "Not Found"))

(defn init
  "init will be called once when
  app is deployed as a servlet on
  an app server such as Tomcat
  put any initialization code here"
  []
  ;... snipped for brevity ...)

(defn destroy
  "destroy will be called when your application
  shuts down, put any clean up code here"
  []
  ;... snipped for brevity ...)

;; timeout sessions after 30 minutes
(def session-defaults
  {:timeout (* 60 30)
   :timeout-response (redirect "/")})

(defn- mk-defaults
  "set to true to enable XSS protection"
  [xss-protection?]
  ;... snipped for brevity ...
)

(def app (app-handler
  ;; add your application routes here
  )
Chapter 1

[home-routes base-routes]
;; add custom middleware here
:middleware (load-middleware)
:ring-defaults (mk-defaults false)
;; add access rules here
:access-rules []
;; serialize/deserialize the following data formats
;; available formats:
:formats [:json-kw :edn :transit-json])

We'll get into detail about what all is happening, and when, in Chapter 2, *Ring and the Ring Server*.

**repl.clj**

The last Luminus generated namespace, hipstr.repl, is one that often confuses beginners because it's strikingly similar to hipster.handler. The hipstr.repl namespace has a start-server and stop-server function, much like hipster.handler. However, hipstr.repl allows us to start and stop our development server from the Clojure REPL. This might seem like a weird thing to do, but by running our server from the REPL we can modify our running system and the changes will be "automagically" reloaded in our server. No need for the time consuming and frustrating "compile-deploy-restart-grab-a-coffee-and-twiddle-your-thumbs cycle!"

(ns hipstr.repl
 (:use hipstr.handler
  ring.server.standalone
  [ring.middleware file-info file]))

(defonce server (atom nil))

defn get-handler []
   ;; #'app expands to (var app) so that when we reload our code,
   ;; the server is forced to re-resolve the symbol in the var
   ;; rather than having its own copy. When the root binding
   ;; changes, the server picks it up without having to restart.
   ; ... snipped for brevity ...
)

defn start-server
   "used for starting the server in development mode from REPL"
   [& [port]]
Incorporating the REPL into your development workflow is a wonderful thing to do. You can load your namespace into the REPL while you work on it and test the code while you're developing right then and there. In fact, some IDEs such as LightTable take this a step further, and will "live-evaluate" your code as you type. The ability of running the dev server from the REPL completes the circle.

If you're not currently using a decent IDE for Clojure development, I strongly encourage you to give LightTable a try. It's free, open source, lightweight, and very different than anything you're used to. It's quite good. Check it out at http://www.lighttable.com.

**Summary**

In this chapter, you learned how to generate a new Clojure-based web application using Leiningen and the Luminus template. We also got a high-level understanding of each dependency, and how Luminus structures its projects. In the next chapter we'll take a detailed look at the Ring and Ring Server libraries, and what they're responsible for. It sounds a little dry, I know, but I recommend that you read it. There will be cake and punch at the end, but without all the calories of cake and punch.
Where to buy this book

You can buy Clojure Web Development Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.