Extracted from:

Web Development with Clojure, Third Edition

Build Large, Maintainable Web Applications Interactively

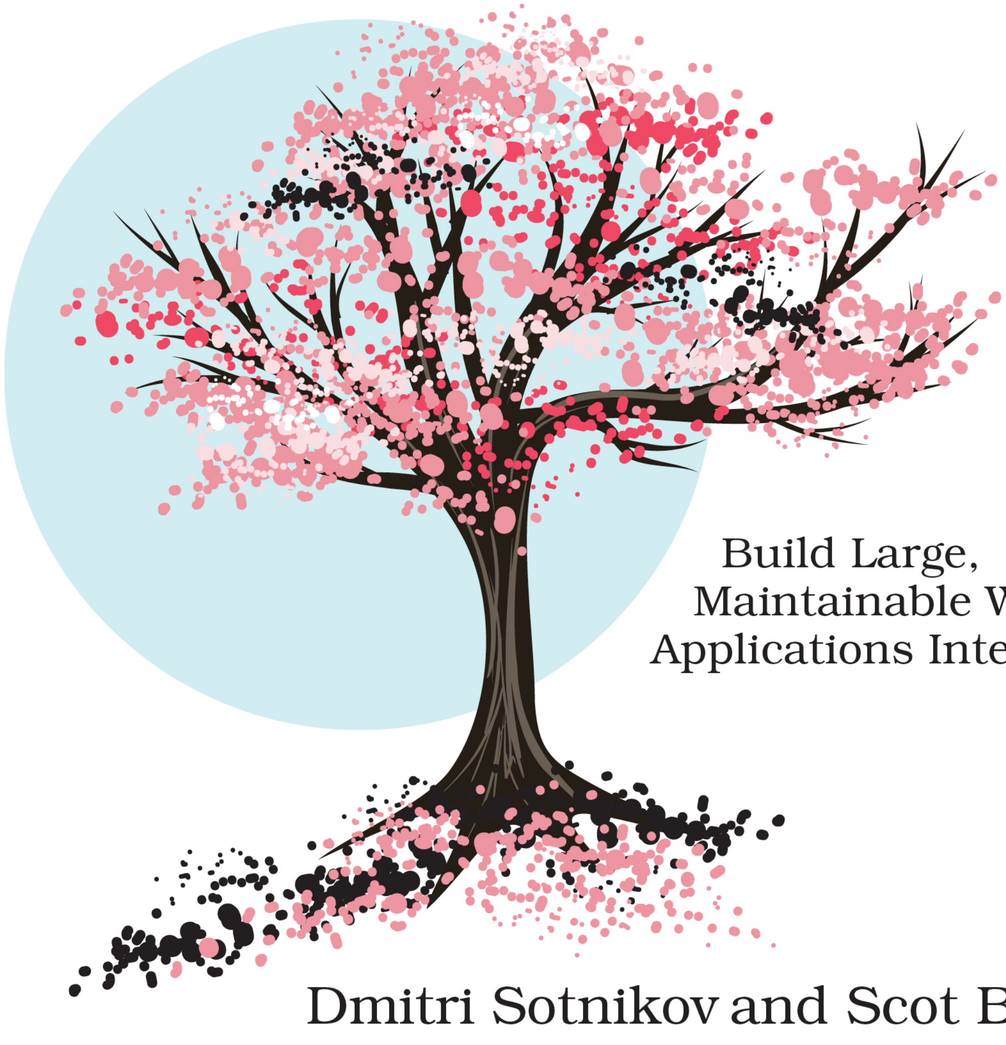
The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Web Development with Clojure

3rd Edition



Build Large, Maintainable Web Applications Interactively

Dmitri Sotnikov and Scot Brown

edited by Michael Swaine

Web Development with Clojure, Third Edition

Build Large, Maintainable Web Applications Interactively

Dmitri Sotnikov
Scot Brown

The Pragmatic Bookshelf
Raleigh, North Carolina

Pragmatic Bookshelf

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Michael Swaine
Copy Editor: L. Sakhi MacMillan
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-682-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—July 2021

Defining HTTP Routes

We've now confirmed that we're able to store and retrieve the messages from the database, and we've written tests to make sure we don't have any regressions. Next, we'll need to write a user interface that calls these functions. We'll create HTTP endpoints and have these call the function that corresponds to the user's intended action. The endpoints are commonly referred to as *routes*, and the route that renders the home page for our application is found in the guestbook.routes.home namespace.

guestbook-base/src/clj/guestbook/routes/home.clj

```
(ns guestbook.routes.home
  (:require
   [guestbook.layout :as layout]
   [guestbook.db.core :as db]
   [clojure.java.io :as io]
   [guestbook.middleware :as middleware]
   [ring.util.response]
   [ring.util.http-response :as response]))

(defn home-page [request]
  (layout/render request "home.html" {:docs (-> "docs/docs.md"
                                                io/resource
                                                slurp)}))

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

(defn home-routes []
  [""
   {:middleware [middleware/wrap-csrf
                 middleware/wrap-formats]}
   ["/" {:get home-page}]
   ["/about" {:get about-page}]])
```

You can see that the / route calls the home-page function that in turn renders the home.html template. You can also see that we're passing a map of parameters to the render function; currently the only parameter being passed is the :docs key. These parameters indicate dynamic content that is injected into our template before it's sent to the client. Let's take a quick look at the contents of the resources/html/home.html file:

guestbook-base/resources/html/home.html

```
{% extends "base.html" %}
{% block content %}
  <div class="content">
  {{docs|markdown}}
  </div>
{% endblock %}
```

You can see that this file extends a template called base.html and renders a block called *content*. The parent template provides a common layout for the pages in our application, and each individual page can render the portion of the page relevant to it. If you're familiar with Rails or Django templates, then this syntax should look very familiar. You'll also note that the templates are set up to use Bulma CSS[13] as the default scaffolding for the page layout.

The templates use a context map to populate the dynamic content. The keys in the map are used as template variables. For example, the {{docs|markdown}} statement corresponds to the :docs key in the map that was passed to the layout/render function by the home-page function.

Let's update our resources/html/home.html template to display our messages.

First, let's replace {{docs|markdown}} with a wrapper that will center and format our page nicely:

guestbook/resources/html/home.html
```
{% block content %}
<div class="content">
    <div class="columns is-centered">
        <div class="column is-two-thirds">
            <!-- Content -->
        </div>
    </div>
</div>
{% endblock %}
```

Then let's add some HTML inside our wrapper that displays a list of existing messages. Just like with docs, our messages are supplied using a variable called messages. Each item in messages is a map containing keys called timestamp, message, and name. We iterate over the messages and create an li tag for each message inside a ul tag, like so:

guestbook/resources/html/home.html
```
<div class="columns">
    <div class="column">
        <h3>Messages</h3>
        <ul class="messages">
            {% for item in messages %}
            <li>
                <time>
                    {{item.timestamp|date:"yyyy-MM-dd HH:mm"}}
                </time>
                <p>{{item.message}}</p>
                <p> - {{item.name}}</p>
```

13. https://bulma.io/documentation/

```
            </li>
            {% endfor %}
        </ul>
    </div>
</div>
```

Okay, let's go back to the guestbook.routes.home namespace and add the code to render the existing messages. Note we already have a reference to the guestbook.db.core namespace in the ns declaration at the top of the home namespace.

We can now update the home-page function to associate the messages with the :messages key when rendering the template.

guestbook/src/clj/guestbook/routes/home.clj

```
(defn home-page [request]
  (layout/render
   request "home.html" {:messages (db/get-messages)}))
```

Since we've already populated a message in our database during earlier testing, we should see it when we reload the page. We can now take a look at adding a form to create new messages from the page.

Now we need to create another div that contains a form for submitting new messages. Note that we need to provide a {% csrf-field %} in our form.[14] Luminus enables anti-forgery protection by default, and any POST requests that don't contain the anti-forgery token are rejected by the server.

guestbook/resources/html/home.html

```
<div class="columns">
    <div class="column">
        <form method="POST" action="/message">

            {% csrf-field %}
            <div class="field">
                <label class="label" for="name">
                    Name
                </label>
                <input class="input"
                       type="text"
                       name="name"
                       value="{{name}}" />
            </div>

            <div class="field">
                <label class="label" for="message">
                    Message
                </label>
                <textarea
```

14. http://en.wikipedia.org/wiki/Cross-site_request_forgery

```
        class="textarea"
        name="message">{{message}}</textarea>
</div>
```

```
            <input type="submit"
                   class="button is-primary"
                   value="comment" />

        </form>
    </div>
</div>
```

Our final template should look as follows:

guestbook/resources/html/home.html

```
{% extends "base.html" %}
{% block content %}
<div class="content">
    <div class="columns is-centered">
        <div class="column is-two-thirds">
            <!-- Content -->
            <div class="columns">
                <div class="column">
                    <h3>Messages</h3>
                    <ul class="messages">
                        {% for item in messages %}
                        <li>
                            <time>
                                {{item.timestamp|date:"yyyy-MM-dd HH:mm"}}
                            </time>
                            <p>{{item.message}}</p>
                            <p> - {{item.name}}</p>
                        </li>
                        {% endfor %}
                    </ul>
                </div>
            </div>
            <div class="columns">
                <div class="column">
                    <form method="POST" action="/message">

                        {% csrf-field %}
                        <div class="field">
                            <label class="label" for="name">
                                Name
                            </label>
                            <input class="input"
                                   type="text"
                                   name="name"
                                   value="{{name}}" />
                        </div>

                        <div class="field">
                            <label class="label" for="message">
                                Message
                            </label>
```

```
                                        <textarea
                                            class="textarea"
                                            name="message">{{message}}</textarea>
                                    </div>
                                    <input type="submit"
                                           class="button is-primary"
                                           value="comment" />

                                </form>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
{% endblock %}
```

We now need to create a new route on the server, called */message*, to respond to the HTTP POST method. The route should call the function save-message! with the request to create a new message.

guestbook/src/clj/guestbook/routes/home.clj

```
["/message" {:post save-message!}]
```

The route handler calls the save-message! function that follows. This function grabs the params key from the request. This key contains a map of parameters that were sent by the client when the form was submitted to the server.

guestbook/src/clj/guestbook/routes/home.clj

```
(defn save-message! [{:keys [params]}]
 (db/save-message! params)
 (response/found "/"))
```

Since we named our fields *name* and *message*, they match the fields we defined in our table: to create a new record all we have to do is call the save-message! function from the db namespace with the params map. Once the message is saved, we redirect back to the home page. The final code in the namespace should look as follows:

guestbook/src/clj/guestbook/routes/home.clj

```
(ns guestbook.routes.home
  (:require
   [guestbook.layout :as layout]
   [guestbook.db.core :as db]
   [clojure.java.io :as io]
   [guestbook.middleware :as middleware]
   [ring.util.response]
   [ring.util.http-response :as response]))

(defn home-page [request]
  (layout/render
   request "home.html" {:messages (db/get-messages)}))
```

```
(defn save-message! [{:keys [params]}]
 (db/save-message! params)
 (response/found "/"))

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

(defn home-routes []
  [""
   {:middleware [middleware/wrap-csrf
                 middleware/wrap-formats]}
   ["/" {:get home-page}]
   ["/message" {:post save-message!}]
   ["/about" {:get about-page}]])
```

At this point our guestbook should display existing messages as well as allow the users to post new messages. As a last touch, we'll add some CSS to style our app. Static assets such as CSS, images, and JavaScript are found in the resources/public folder and are served without the need to define routes for them. Let's add the following CSS in the resources/public/css/screen.css file:

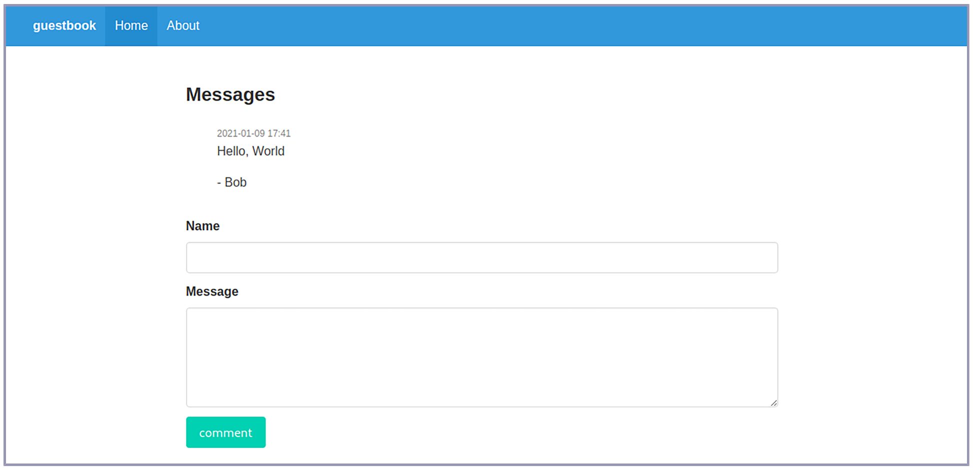
guestbook/resources/public/css/screen.css

```
ul.messages {
          list-style: none;
}

ul.messages li {
          padding: 0.5em;
          border-bottom: 1px dotted #ccc;
}

ul.messages li:last-child {
          border-bottom: none;
}

li time {
          font-size: 0.75em;
    color: rgba(0, 0, 0, 0.5);
}
```

The guestbook page should now look like the figure on page 8.

Validating Input

What else should we do? Currently, our guestbook doesn't do any validation of user input. That's weak. Let's see how we can ensure that user messages contain the necessary information before trying to store them in the database.



Messages

2021-01-09 17:41
Hello, World

- Bob

Name

Message

Luminus defaults to using the Struct library[15] to handle input validation. The library provides a straightforward way to check that our parameter map contains the required values.

Struct uses struct.core/validate function for handling validation. This function accepts a map containing the parameters followed by the validation schema. The schema is used to validate the input and return error messages for any invalid fields.

Many common validators such as required, email, matches, and so on are provided by Struct out of the box. These validators can be used individually or chained together to validate different aspects of the input value. Also, we can easily create custom validators for situations where the default ones won't do.

Before we see how validation works, we want to include struct.core in our guestbook.routes.home namespace.

```
(ns guestbook.routes.home
  (:require
  ...
  [struct.core :as st]))
```

We can now use the st/validate function to check that the input values are valid and to produce an error message when they're not. In our case, we need to ensure both that the username is not empty and that the message has at least ten characters before we persist them to the database. Our validation schema looks as follows:

15. http://funcool.github.io/struct/latest

guestbook-validation/src/clj/guestbook/routes/home.clj

```clojure
(def message-schema
  [[:name
    st/required
    st/string]
   [:message
    st/required
    st/string
    {:message "message must contain at least 10 characters"
     :validate (fn [msg] (>= (count msg) 10))}]])
```

The validation schema specifies that both the :name and the :message keys are required, that they're strings, and that the message must be more than 9 characters in length. The validation function then calls the st/validate function, passing it the params along with the schema to validate the message.

guestbook-validation/src/clj/guestbook/routes/home.clj

```clojure
(defn validate-message [params]
  (first (st/validate params message-schema)))
```

The result of the validate function is a vector where the first element is either nil when the validation passes or a map of errors. The keys in the map are the parameters that failed validation and the values are the error messages.

The next step is to hook up the validation function into our workflow. Currently, the save-message! function attempts to store the message and then redirects back to the home page. We need to add the ability to pass back the error message along with the original parameters when validation fails.

A common approach for this is to use a flash session to track the errors. Flash sessions have a lifespan of a single request, making them ideal storage for this purpose. The save-message! function validates the input and checks for errors. If it finds errors, it associates a :flash key with the response that contains the parameters along with the errors. If no errors are generated, it saves the message to the database and redirects as it did before.

guestbook-validation/src/clj/guestbook/routes/home.clj

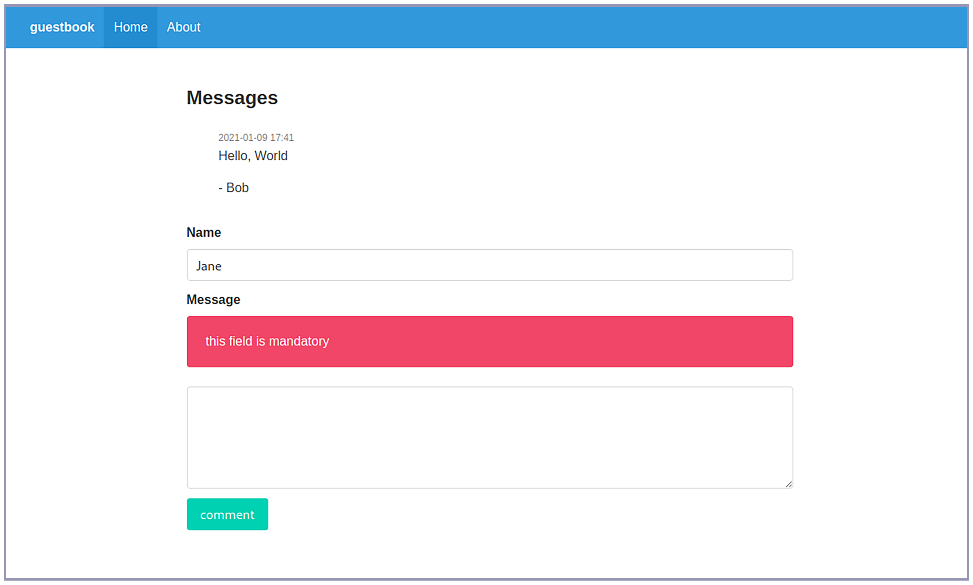
```clojure
(defn save-message! [{:keys [params]}]
  (if-let [errors (validate-message params)]
    (-> (response/found "/")
        (assoc :flash (assoc params :errors errors)))
    (do
      (db/save-message! params)
      (response/found "/"))))
```

We can now update the home-page function to check for the :flash key. Let's select the name, message, and errors keys from the flash session and merge them with our parameter map.

guestbook-validation/src/clj/guestbook/routes/home.clj
```
(defn home-page [{:keys [flash] :as request}]
  (layout/render
   request
   "home.html"
   (merge {:messages (db/get-messages)}
          (select-keys flash [:name :message :errors]))))
```

Finally, let's update our page to render the errors when they're present (also see figure on page 11).

guestbook-validation/resources/html/home.html
```
<div class="columns">
    <div class="column">
        <form method="POST" action="/message">

            {% csrf-field %}
            <div class="field">
                <label class="label" for="name">
                    Name
                </label>
                {% if errors.name %}
                <div class="notification is-danger">
                    {{errors.name|join}}
                </div>
                {% endif %}
                <input class="input"
                       type="text"
                       name="name"
                       value="{{name}}" />
            </div>

            <div class="field">
                <label class="label" for="message">
                    Message
                </label>
                {% if errors.message %}
                <div class="notification is-danger">
                    {{errors.message|join}}
                </div>
                {% endif %}
                <textarea
                    class="textarea"
                    name="message">{{message}}</textarea>
            </div>
            <input type="submit"
                  class="button is-primary"
                  value="comment" />

        </form>
    </div>
</div>
```



Messages

2021-01-09 17:41
Hello, World

- Bob

Name

Jane

Message

this field is mandatory

Running Standalone

Up to now, we've been running our app using the lein run command. This starts an embedded server in development mode so that it watches files for changes and reloads them as needed. To package our application for deployment, we can package it into a runnable JAR as follows:

```
$ lein uberjar
```

The archive will be created in the target folder of our application, and we can run it using the java command.

Since we're using a database, we also have to make sure that the connection is specified as an environment variable. When we ran our application in development mode, the connection variable was provided in the dev-config.edn file. However, now that the application has been packaged for production, this variable is no longer available. Let's create a connection variable and then run our application as follows:

```
$ export DATABASE_URL="jdbc:h2:./guestbook_dev.db"
$ java -jar target/uberjar/guestbook.jar
```

What You've Learned

Okay, that's the whirlwind tour. By this point you should be getting a feel for developing web applications with Clojure, and you should be comfortable with some of the Clojure basics. You saw how to use Leiningen to create and manage applications. You learned about HTTP routing and some basic HTML templating. While we didn't explore many aspects of the skeleton application that was generated for us, you saw how the basic request life cycle is handled.

We'll be diving deeper and writing more code in upcoming chapters. If you aren't already, you should start using one of the popular Clojure-aware editor plugins, such as Cursive (IntelliJ), Calva (VSCode), Cider (Emacs), Fireplace (Vim), or Counterclockwise (Eclipse). We can't overstate the value of these tools, and we strongly recommend taking the time to set one up. See Appendix 2, Editor Configuration, on page ?, for guides on setting up a few popular Clojure editors.

In the next chapter, we'll delve into the details of the Clojure web stack to understand some of the details of how our application works.