St. Louis Clojure

ClojureScript Introduction

Christopher Mark Gore

cgore.com

Tuesday, May 26, AD 2015

Why ClojureScript?

Have you ever seen JavaScript?

Why ClojureScript?

Have you ever seen Clojure?

Why I Care

I have a side project that I'm writing in Ruby on Rails.

The backend is mostly done, but I need to make a nicer frontend for it.

This means JavaScript needs to happen somehow.

How I'm going to do stuff.

- I'm using Leiningen for the projects:
  `http://leiningen.org/`
- I'm using `cljs-kickoff` to template the project:
  `https://github.com/konrad-garus/cljs-kickoff`
- That uses `lein-cljsbuild`:
  `https://github.com/emezeske/lein-cljsbuild`
- And also `lein-ring`:
  `https://github.com/weavejester/lein-ring`

ClojureScript History

1958 LISP

1969 ARPANET

1984 Common Lisp

1990 WWW

1994 Netscape Navigator

1995 JavaScript

2007 Clojure

2012? ClojureScript

cljs-kickoff

There is some real work to standing up the file structure for any Clojure project, and ClojureScript is no exception. That's where `cljs-kickoff`[a] comes in: it sets up a minimal Leiningen template for ClojureScript with `lein-cljsbuild`.

```
$ lein new cljs-kickoff hello-world
```

[a] https://github.com/konrad-garus/cljs-kickoff

cljs-kickoff

```
hello-world
├── project.clj
├── resources
│   └── public
│       ├── css
│       │   └── page.css
│       └── help.html
└── src
    ├── clj
    │   └── hello_world
    │       └── server.clj
    └── cljs
        └── hello_world
            └── client.cljs
```

`lein-cljsbuild`

Your beautiful ClojureScript code needs to be *"compiled"* (air quotes) into ugly JavaScript to actually work, and you can use `lein-cljsbuild`[a] does that for you automatically whenever you change a relevant file.

On a dedicated shell session:

```
$ lein cljsbuild auto
```

[a] https://github.com/emezeske/lein-cljsbuild

lein-ring

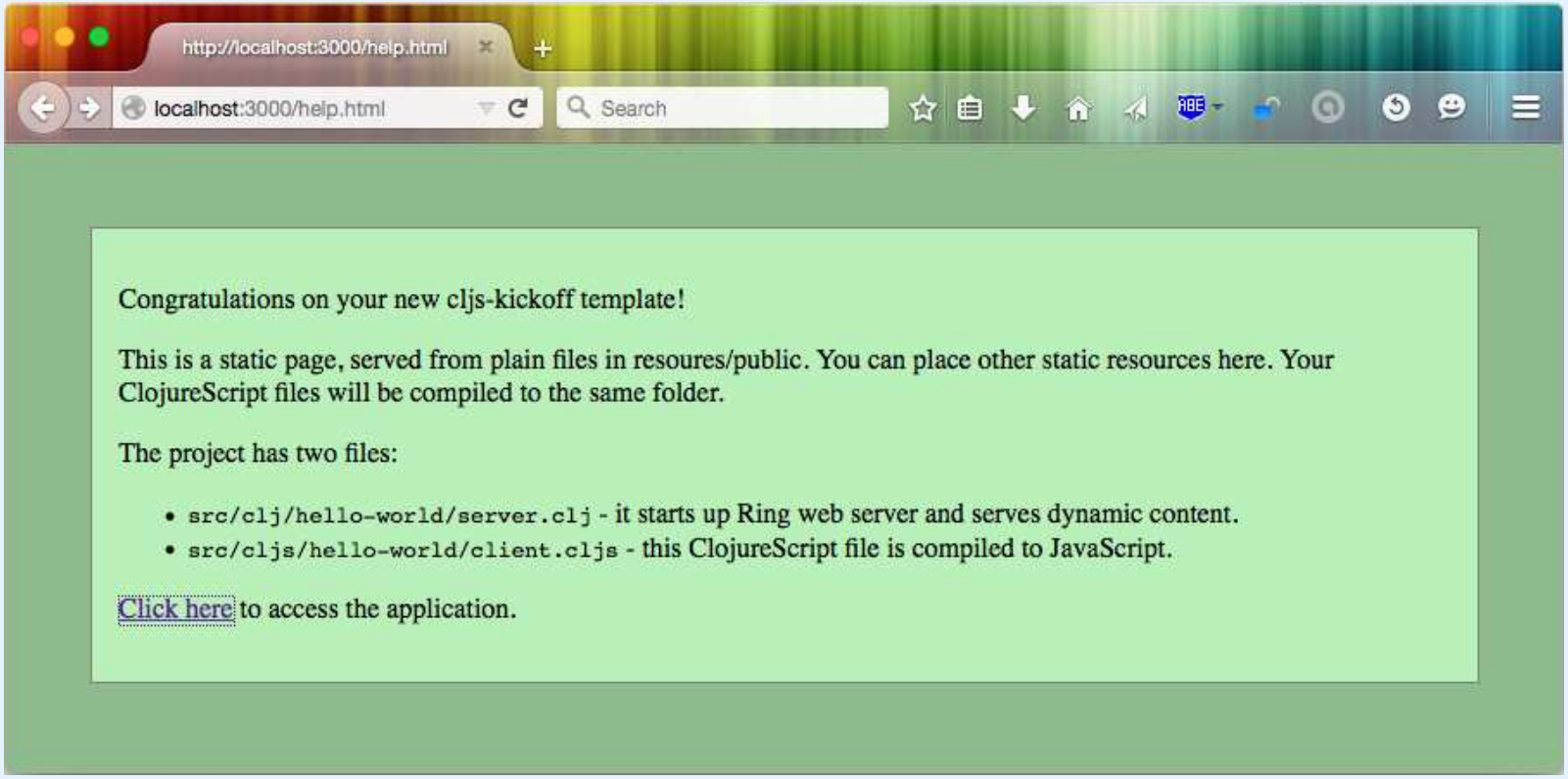
You will want a simple web server to put up your ClojureScript, and you probably want it in Clojure. Ring[a] is a popular one, loosely similar in approach to Ruby's Rack, and `lein-ring`[b] provides a lot of nice Leiningen shortcuts for Ring.

On a dedicated shell session:

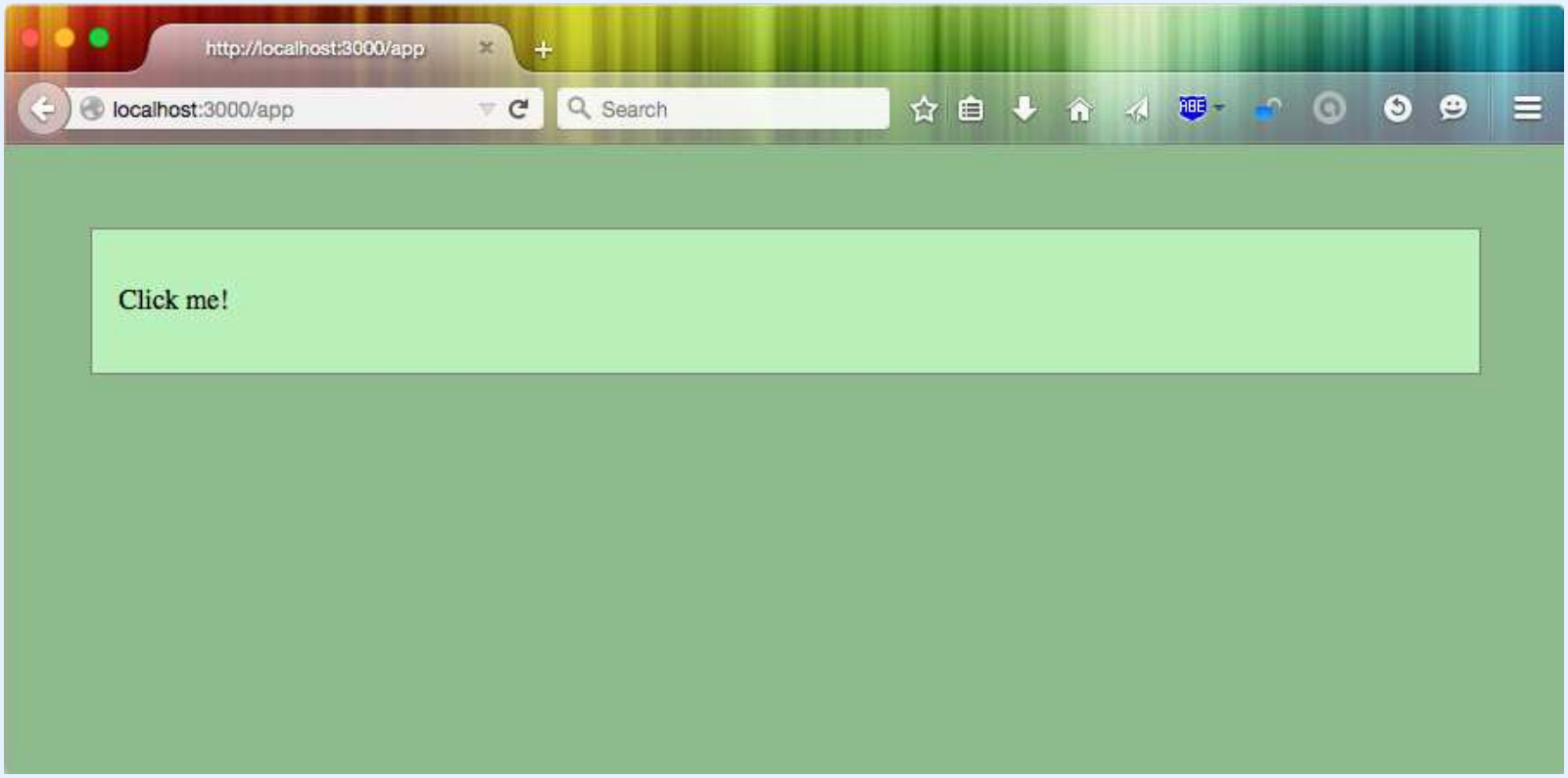
```
$ lein ring server
```

[a] https://github.com/ring-clojure/ring
[b] https://github.com/weavejester/lein-ring

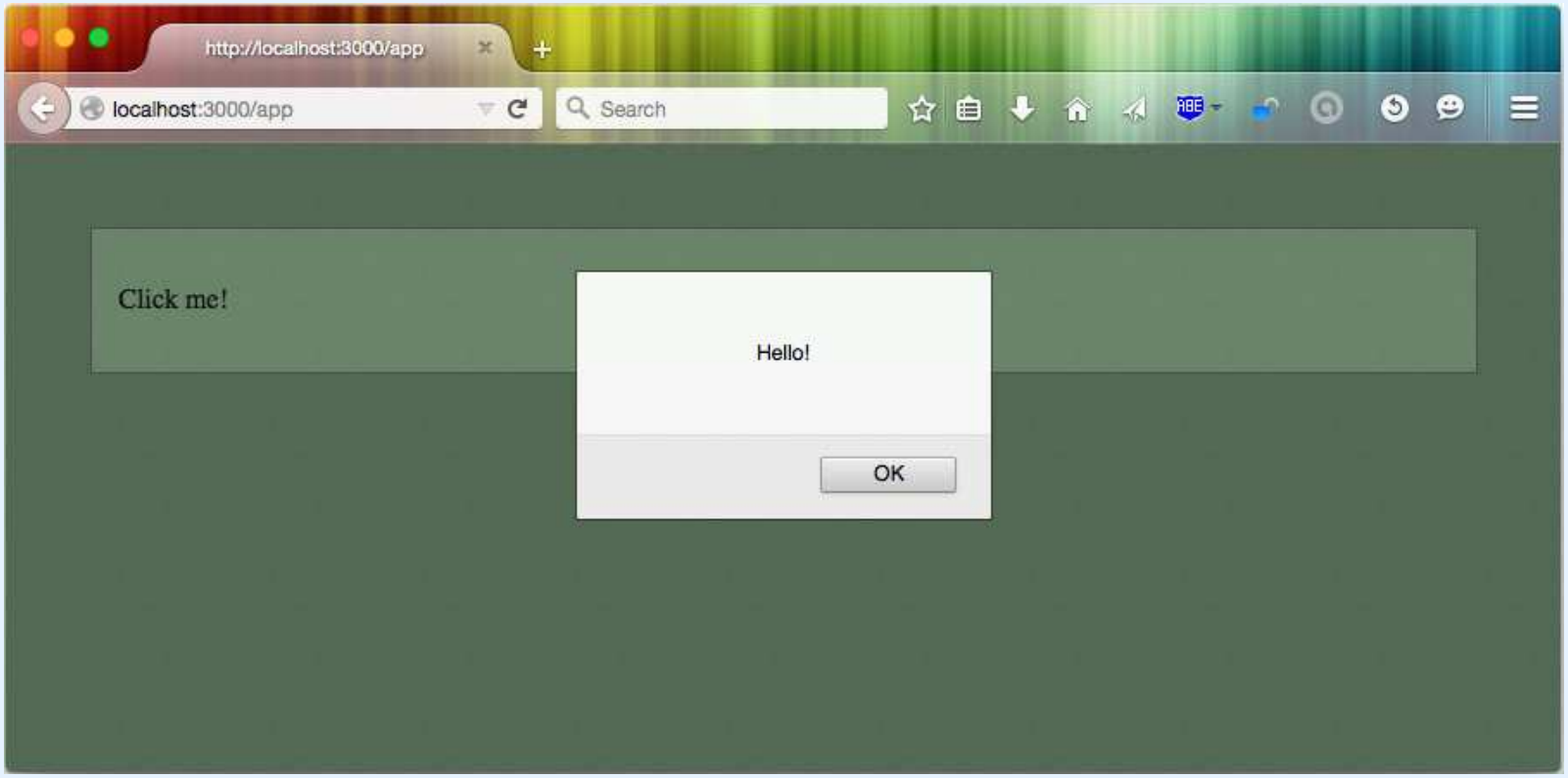
Hello, World!


http://localhost:3000/help.html
localhost:3000/help.html
Search
Congratulations on your new cljs-kickoff template!
This is a static page, served from plain files in resoures/public. You can place other static resources here. Your ClojureScript files will be compiled to the same folder.
The project has two files:
src/clj/hello-world/server.clj - it starts up Ring web server and serves dynamic content.
src/cljs/hello-world/client.cljs - this ClojureScript file is compiled to JavaScript.
Click here to access the application.

Hello, World!



Hello, World!


http://localhost:3000/app
localhost:3000/app
Search
Click me!
Hello!
OK

project.clj: Dependencies

```
1 (defproject hello-world "0.1.0-SNAPSHOT"
2   ;; ...
3   :dependencies [[org.clojure/clojure "1.5.1"]
4                  [org.clojure/clojurescript "0.0-2156"]
5                  [ring "1.2.1"]]
6   ;; ...
7   )
```

project.clj: Plugins

```
1 (defproject hello-world "0.1.0-SNAPSHOT"
2   ;; ...
3   :plugins [[lein-cljsbuild "1.0.2"]
4             [lein-ring "0.8.10"]]
5   ;; ...
6   )
```

project.clj: ClojureScript Build

```
1  (defproject hello-world "0.1.0-SNAPSHOT"
2    ;; ...
3    :hooks [leiningen.cljsbuild]
4    :source-paths ["src/clj"]
5    :cljsbuild {
6      :builds {
7        :main {
8          :source-paths ["src/cljs"]
9          :compiler {:output-to "resources/public/js/cljs.js"
10                     :optimizations :simple
11                     :pretty-print true}
12         :jar true}}}
13   ;; ...
14   )
```

project.clj: Ring Server

```
1 (defproject hello-world "0.1.0-SNAPSHOT"
2   ;; ...
3   :main hello-world.server
4   :ring {:handler hello-world.server/app})
```

src/clj/hello_world/server.clj

This is pretty boring, a really simple server with a single static page. The most important part is the `script` directive.

```
(defn render-app []
  {:status 200
   :headers {"Content-Type" "text/html"}
   :body
   (str "<!DOCTYPE html>"
        ;; ...
        "<p id=\"clickable\">Click me!</p>"
        ;; ...
        "<script src=\"js/cljs.js\"></script>"
        ;; ...
        )})
```

src/cljs/hello_world/client.cljs

```
1 (ns hello-clojurescript)
2
3 (defn handle-click []
4   (js/alert "Hello!"))
5
6 (def clickable (.getElementById js/document "clickable"))
7 (.addEventListener clickable "click" handle-click)
```

ClojureScript is Clojure!

ClojureScript is great! Everything you can do in Clojure, you can do in ClojureScript![a]

```
(+ 1 2 3)

(defn add2 [x]
  (+ x 2))

(def foo 42)

(defn average [x y]
  (/ (+ x y)
     2))

(map + [1 2 3]
       [4 5 6])
```

[a] Except for when you can't.

You Can Get to JavaScript if You Need It

Just like Java is hiding just under the covers of Clojure, you have JavaScript hiding just under the covers of ClojureScript. You can easily get to anything in JavaScript from ClojureScript.

```
;; console.log("hi!")
(.log js/console "hi!")
;; document.getElementById("clickable")
(.getElementById js/document "clickable")
```

ClojureScript Web REPL

There is an online ClojureScript REPL at `clojurescript.net`.

Calling JavaScript Methods

You can easily call methods on JavaScript objects from ClojureScript.

```
;; Basic form
(.the-method target-object args ...)
;; document.getElementById("clickable")
(def clickable (.getElementById js/document "clickable"))
;; clickable.addEventListener("click", handle-click)
(.addEventListener clickable "click" handle-click)
```

Accessing JavaScript Properties

You can easily access the properties of JavaScript objects from ClojureScript.

```
;; Basic form
(.-property target-object -property)
;;  document.title
(.-title js/document) ; => "Some String"
```

Setting JavaScript Properties

You can easily set the properties of JavaScript objects from ClojureScript.

```
;; Basic form
(set! (.-property target-object) new-value)
;; document.title = "Hi There"
(set! (.-title js/document) "Hi There")
```

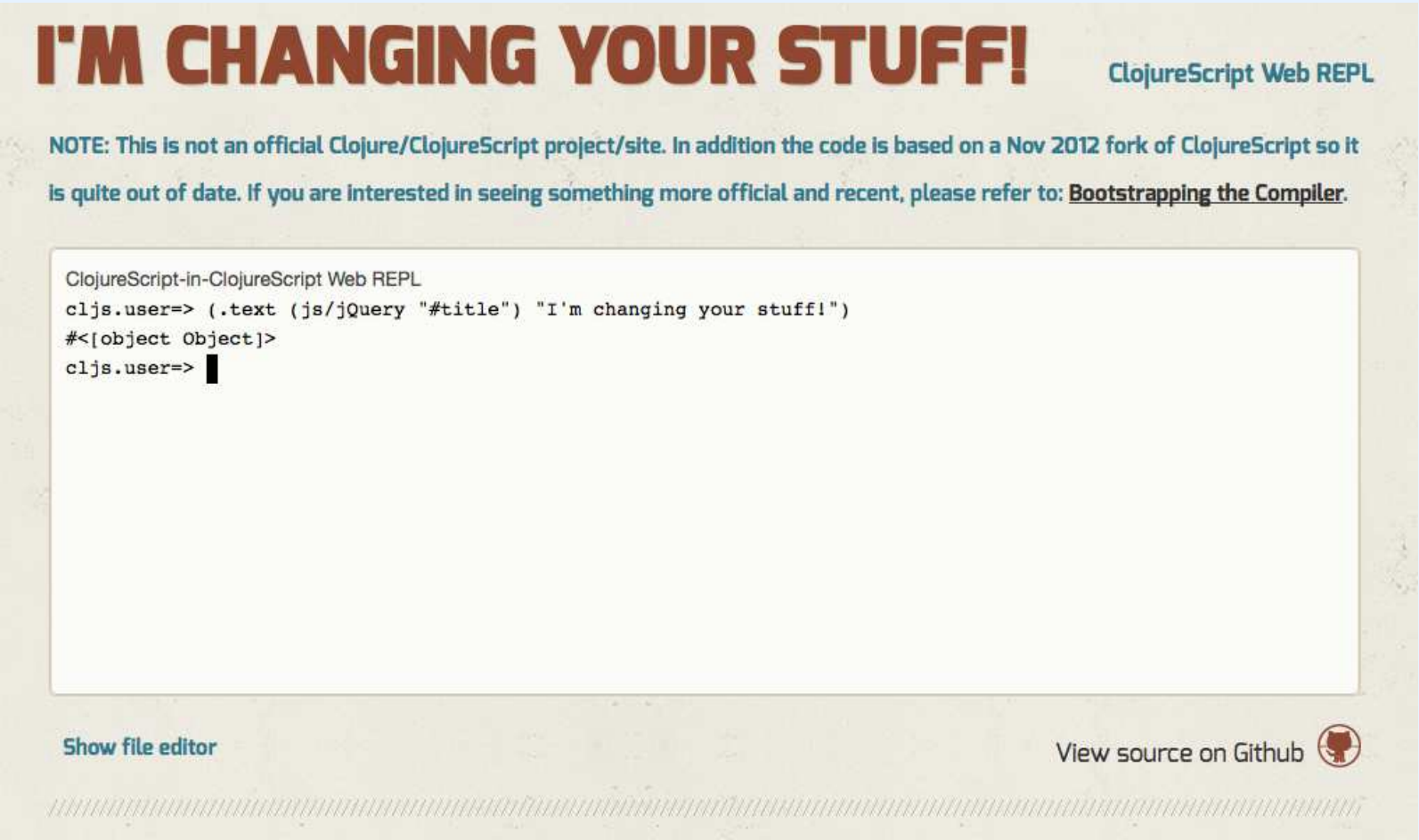
Direct JavaScript

JavaScript has an `eval` function, and we can get to it from ClojureScript.

```
(js/eval "2+2") ; => 4
(js/eval "document.title␣=␣\"Hi␣there\"")
(js/eval "x␣=␣123")
js/x ; => 123
(js/eval "Math.random()") ; => 0.9831978017934505
(rand) ; You would probably do this instead.
```

Using External JavaScript Libraries

You can easily use existing JavaScript libraries from ClojureScript.



ClojureScript REPL in Emacs

A web REPL is nice, but to do real work I need a real REPL in a real text editor[a]. We can use the Austin[b] plugin for this. Add the following to your `project.clj`[c] file:

```
:profiles {:dev {:plugins [[com.cemerick/austin "0.1.6"]]}}
```

Then run `lein repl` (or the nREPL inside of Emacs) in the project and launch:

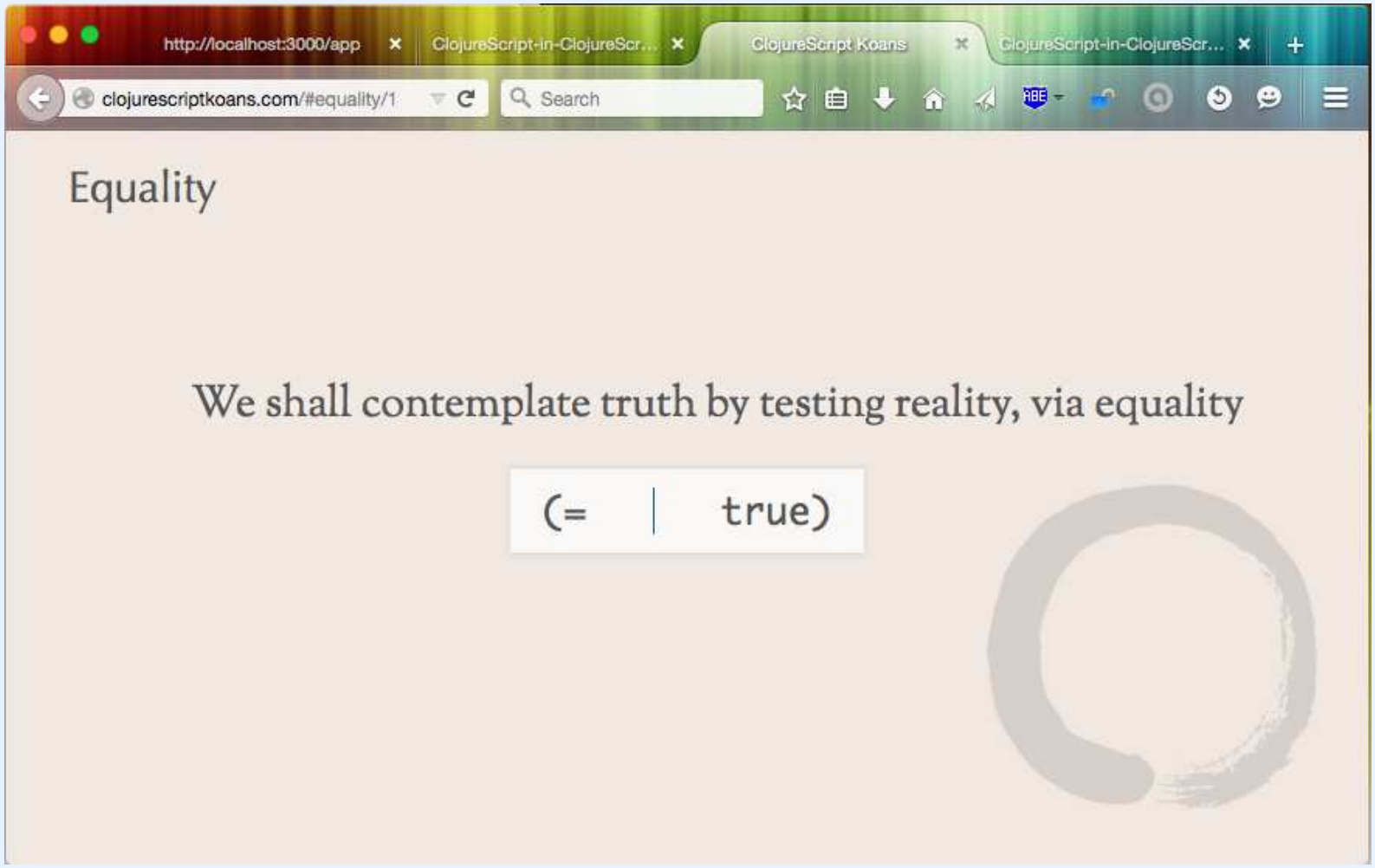
```
(cemerick.austin.repls/exec)
```

[a] That means Emacs or Vim, and since this is a Lisp, Emacs thanks to nepotism.

[b] `https://github.com/cemerick/austin`

[c] An example working project is in the Austin repo. The versions from `cljs-kickoff` don't seem to play well with Austin.

ClojureScript Koans: `clojurescriptkoans.com`



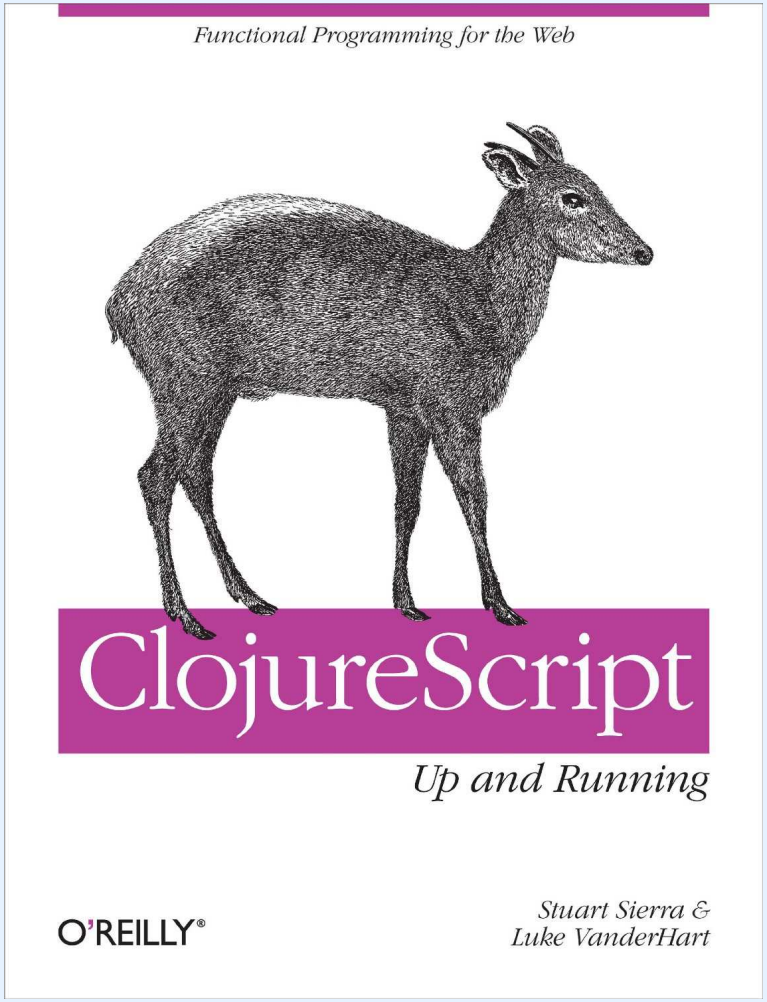
Modern ClojureScript

A really nice set of tutorials is *Modern ClojureScript*, available at:

`https://github.com/magomimmo/modern-cljs`

ClojureScript: Up and Running

This seems to be the only book out there, from 2012.



Variables

```
;; Top-level variables.
(def foo "bar") ; var foo = "bar"
;; Local variables.
(defn foo []          ; function foo() {
  (let [bar 1         ;     var bar = 1;
        baz 2]        ;     var baz = 2;
    (+ bar baz 3))    ;     return bar + baz + 3;
                      ; }
```

Destructuring Bind

```
(def m {:first "Robert"        ; var m = {first: "Robert",
        :last "Rogers"         ;           last: "Rogers",
        :born "11/07/1731"     ;           born: "11/07/1731",
        :died "05/18/1795"})   ;           died: "05/18/1795"};
(let [{:keys [first last born] m}] ; var first = m.first
                                   ; var last = m.last
                                   ; var born = m.born
  ;; first + " " + last + " was born on " + born + ".";
  (str first "␣" last "␣was␣born␣on␣" born "."))
```

Arrays

In JavaScript:

```
var a = new Array();
var b = [];
var c = [1 2 3];
c[1] // => 2
```

In ClojureScript:

```
(def a (array))
(def b (array 1 2 3))
(nth b 1) ; => 2
```

Other Collections

In ClojureScript there are also immutable vectors, immutable lists, and immutable sets, not just arrays:

```
;; Lists
(def l (list))
(def m '())
(def n (list 1 2 3))
(def o '(1 2 3)) ; n = o
;; Vectors
(def v (vector))
(def u [])
(def w [1 2 3])
;; Sets
(def s #{1 2 3})
(def t #{1 1 1 2 2 2 3 3 3}) ; s = t
```

Hash Maps

In JavaScript:

```
1 var m = { // Keys must all be strings
2   "foo": 1,
3   "bar": 2
4 };
5 m["foo"];
6 m.foo;
```

In ClojureScript:

```
1 (def m {:foo     1
2         :bar     2
3         12       3    ; Non-string keys
4         [1 2 3]  4}) ; Anything can be a key
5 (get m :foo)
```

Functions with Variadic Arguments

In JavaScript, you need to manipulate the arguments object yourself. In ClojureScript, you can do this:

```
(defn foo
  ([]
   "no arguments")
  ([x]
   (str "one argument - x = " x))
  ([x y]
   (str "two arguments - x+y = " (+ x y))))
```

Named Parameters and Default Values

You can't do this in JavaScript.

```
(defn foo [x y ; Positional arguments
           & {:keys [bar baz]}] ; Keyword arguments
  (+ x y bar baz))
(foo 1 2 :bar 3 :baz 4) ; => 10
(defn foo2 [x y
            & {:keys [bar baz]
               :or {bar 100      ; Default for bar
                    baz 1000}}] ; Default for baz
  (+ x y bar baz))
(foo2 1 2) ; => 1103
```

My Project's Layout?

1. The backend is in Ruby on Rails.
2. Single/multiple ClojureScript project(s)?
3. Single/multiple git repos?
4. JavaScript too, or just ClojureScript?
5. Other pitfalls?

Questions?