Clojure For React Developers

Toni Väisänen

June 25, 2023

Contents

1 What is Clojure(Script)

Clojure is a general dynamic functional programming language that can be used to build fullstack web applications.

> Clojure is a dynamic, general-purpose programming language, combining the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure is a compiled language, yet remains completely dynamic – every feature supported by Clojure is supported at runtime. Clojure provides easy access to the Java frameworks, with optional type hints and type inference, to ensure that calls to Java can avoid reflection. [1]

And Clojurescript is the version that compiles to JS.

> ClojureScript is a compiler for Clojure that targets JavaScript. It emits JavaScript code which is compatible with the advanced compilation mode of the Google Closure optimizing compiler. [1]

2 Who is This Book For?

This book is meant for developers who have background in React web development and are interested in learning Clojure. React experience is not a must but the book doesn't go into details on how React works. Only how to use it with Clojure. Some of the concepts are explained with JS examples translated to Clojure and vice versa.

[1] //clojure.org

After reading this book and you should be able to get started in Clojurescript web development using the common JS libraries. There's also a wealth of information and useful tools in the Clojure ecosystem that are not covered in this book and my hope is that getting started on the route with more traditional JS-like approach you get curious about all the other aspects as well.

3 Prerequisites

To be able to follow along you should have a working **node** installation configured on your machine and an editor of your choice. A brief explanation is provided how to setup VSCode and Calva to interact with the REPL.

So make sure that you have:

- Installed Node
- Installed VSCode
- Installed the VSCode Calva extension

Of course you can use editor of your choice. Intellij has Cursive a popular pluging for Clojure. Emacs users use CIDER and for VIM there's fireplace. Here we'll be setting up Calva because VSCode's popularity among React devs.

4 Clojure: a Quick Tour

Let's fire up a REPL in the terminal so you can follow along with the examples.

```
$ npx create-cljs-project app
$ cd app
$ npx shadow-cljs node-repl
```

4.1 Namespaces

Clojure code is organized in `namespaces` which can be thought of as modules in other languages. Namespace is declared with a `ns` macro.

```
(ns namespace-name)
```

If you are following along in the REPL you might have noticed that the `cljs.user` in the prompt changed to `namespace-name`.

```
cljs.user=> (ns namespace-name)
nil
namespace-name=>
```

Namespace can have have a docstring as an optional second argument.

```
(ns namespace-name
  "Docstring for the namespace")
```

As in Javascript and Typescript you can import code from other modules.

```
(ns namespace-name
  "Docstring for the namespace"
  :require [other-namespace :as other])

;; use function from another namespace
(other/function arg1)
```

You can use available JS modules by importing them as you would in JS by using the quoted library name when requiring the module.

```
(ns namespace-name
  "Docstring for the namespace"
  :require [other-namespace :as other]
           ["axios" :as axios])

;; use JS modules similarly
(axios/get "url...")
```

Read more about namespaces from the official reference and differences between Clojurescript and Clojure namespaces.

4.2 Variables

In Clojure variables are defined with `def` macro.

```
(def one 1)
(def hello "hello world")
```

After the variables are defined they can be used as expected.

```
(inc one)
;; => 2
```

4.3 Data Types

A quick look at the basic data structures in Clojure. This is in no means an exhaustive list of what's available. Refer to the official documentation for complete picture.

4.3.1 Maps

```
(def number-variable 1)
;; number-variable
;; => 1

(def string-variable "variable")
;; string-variable
;; => "variable"
```

Maps are hash maps comparable to objects in JS.

```
(def map-variable {:a 1 :b 2})
;; map-variable
;; => {:a 1, :b 2}
```

To get values from a map you can use `get` with or without a default in the case that the value is not found. This is similar to pythons `object.get`

```
(get map-variable :a)
;; => 1

(get map-variable :c :not-found)
;; => :not-found
```

The value can also be obtained with they keyword as a function or the map itself as a function applied with the keyword.

```
;; use the key as a function to get the value
(:a map-variable)
;; => 1
;; or use the map as a function to get the value
(map-variable :a)
;; => 1
```

To remove value from a map use `dissoc` but remember that this do not alter the original map, it returns a new value of existing parameters minus the removed parameter.

```
(dissoc map-variable :b)
;; => {:a 1}
```

4.3.2 Vectors

Vectors in Clojure are like arrays in Javascript. Clojure also has lists but we'll concentrate just on the arrays in this context. Vectors are defined with brackets `[]` and lists with parenthesis ().

```
(def vector-variable [1 2 3])
```

We can retrieve a value in and index with `get`.

```
(get vector-variable 0)
;; => 1
```

You can map a function over a vector similarly as you do in JS.
For example the following JS would translate into

```
[1,2,3].map(value => value + 1)
```

the following.

```
(map inc vector-variable)
;; => (2 3 4)
```

In Clojure we do not use the dot notation to access the prototype's methods, but we use a dedicated function `map` and declare all the arguments. Technically you could do this by using JS interop from Clojurescript, but in this case you would not be using the Clojure data structures.

I'll add an example here as a sneak peak and we'll talk more abou the JS interop a bit later.

```
;; array creates a JS array in Clojurescript
;; and by using .map we are using the method of this array
(.map (array 1 2 3) (fn [value] (+ value 1)))
;; => [2 3 4]
```

```
// Compiled JS
[(1),(2),(3)].map((function (value){return (value + (1));}));
```

But let's get back on the topic. Similarly as with `map` we can `filter` and `reduce` vectors.

```
(filter odd? [1 2 3])
;; => (1 3)

(reduce + [1 2 3])
;; => 6

(reduce + 10 [1 2 3])
;; => 16
```

Clojure has threading macros that helps chaining this type of operations together.

```
(->> [1 2 3]
     (map inc)
     (filter even?))
;; => (2 4)
```

Which in practice translates to following.

```
(filter even? (map inc [1 2 3]))
```

We can evaluate the above expression with `macroexpand` function To confirm that it is equivalent.

```
(macroexpand
 '(->> [1 2 3]
       (map inc)
       (filter even?)))
;; => (filter even? (map inc [1 2 3]))
```

Read more about threading macros in the threading macro guide.

4.4 Functions

Functions are defined with `defn` macro.

```
(defn hello-world []
  (println "Hello, World!"))
```

We can inspect the produced Javascript by setting the dynamic variable `*print-fn-bodies*` to true.

```
cljs.user=> (set! *print-fn-bodies* true)

true

cljs.user=> (defn hello-world []
              (println "Hello, World!"))

[#object[cljs$user$hello_world
         "function cljs$user$hello_world(){
            return cljs.core.println.call(null,"Hello, World!");
         }"]]
```

As you can see from the output the result is plain old javascript that uses CLJS core library println function to do the printing.

```
function hello_world(){
    return cljs.core.println.call(null,"Hello, World!");
}
```

This is a good way to get familiar on what is happening behind the scenes. Now, let's do explore more about functions. Function arguments are defined in the vector.

```
(defn hello [name]
  (println (str "Hello " name))
```

Functions can be anonymous and functions can return functions

```
(defn hello-to [name]
  (fn [] (str "Hello " name))

(def hello-to-you (hello-to "you"))
```

```
(with-out-str (hello-to-you))
;; => "Hello you"
```

Anonymous functions can be declared with a reader macro #

```
(defn hello-to [name]
  #(str "Hello " %))

(def hello-to-you (hello-to "you"))

(with-out-str (hello-to-you))
;; => "Hello you"
```

`with-out-str` is a macro that captures the standard output input from a function and returns the captured values as an input so we can inspect the printed charactecs as values.

If we evalue the anynomous function created with # we can see that the arity is generated based on the number of arguments in the function body

```
cljs.user=> (def add #(+ %1 %2))

function cljs$user$add(p1__25209_SHARP_,p2__25210_SHARP_){
    return (p1__25209_SHARP_ + p2__25210_SHARP_);
}
```

By adding an extra arg it's reflected on the argument list.

```
cljs.user=> (def add #(+ %1 %2 %3))

function cljs$user$add(p1__25214_SHARP_,
                       p2__25215_SHARP_,
                       p3__25216_SHARP_){
    return ((p1__25214_SHARP_ + p2__25215_SHARP_) + p3__25216_SHARP_);
}
```

There's still a lot to cover in Clojure but this should be enough for us to get you started on the React side of things.

5 Create a New Project

We'll be creating a simple project where we use the open Star Wars API to fetch characters from the movies and show these characters in the browser. We will also setup a development environment with `devcards` to have a dedicated space to work on the components without having to deal with the application as a whole.

On top of that we will setup unit testing with three types of test runners. One that we can see in the context of the component. Second where we have all the tests of the project in a browser view. Third node test runner for the that can be used withing the CI. And as a cherry on top we'll configure Github Actions to run tests and deploy the application on Github Pages on new commits.

Create a new project with `create-cljs-project` npm package.

```
$ npx create-cljs-project app
```

This sets up the basic project folder structure.

```
app
 node_modules
 package.json
 package-lock.json
 shadow-cljs.edn
 src
```

And adds `shadow-cljs` as a dev dependency as the Clojurescript tooling.

```
{
  "name": "app",
  "version": "0.0.1",
  "private": true,
  "devDependencies": {
    "shadow-cljs": "2.21.0"
  },
  "dependencies": {}
}
```

Clojure related dependecies and configuration is in `shadow-cljs.edn`

```
{:source-paths
 ["src/dev"
  "src/main"
  "src/test"]

 :dependencies
 []

 :builds
 {}}
```

As the configuration shows, it there's no dependencies by default nor build configurations. To be able to compile our Clojurescript source code into Javascript, we need to setup a build target for that.

But let's start by creating some source code to compile in the first place.

```
mkdir src/main/app
touch src/main/app/core.cljs
```

And write the following code into that file.

```
(ns app.core)
;; Here we define the name for the namespace that is like a "module" in Javascript
;; The name `core` is used often in clojure similarly as `index.js` in Javascript

(defn start
  "We'll configure this to run after loading"
  []
  (prn "app start"))

(defn stop
  "We'll configure this to run before loading"
  []
  (prn "app stop"))

(defn init
  "We'll configure this to be run when index.html is loaded the first time."
  []
  (js/console.log "Browser loaded the code"))
```

and after this lets configure the html file to load the code

```
mkdir public
touch public/index.html
```

```
<!DOCTYPE html>
<html>
    <body>
        <div id="app"/>
        <script src="/js/compiled/main.js" type="text/javascript"></script>
        <script>app.core.init();</script>
    </body>
</html>
```

5.1 Set browser build target

And finally lets configure the build to emit the `main.js` file from Clojure sources.

```
 ;; shadow-cljs configuration
{:source-paths ["src/dev"
                "src/main"
                "src/test"]

 :dependencies [
                ;; Use Chrom(e/ium), do not work on Firefox
                [binaryage/devtools "0.9.7"]

                ;; This is used for interacting with the application
                ;; from the browser. A bit more of that later.
                [cider/cider-nrepl "0.28.1"]]

 :builds
 {:app {
        ;; the javascript bundle is targeted to browser env
        :target     :browser

        ;; the module `:main` is written here as `main.js`
        :output-dir "public/js/compiled"
        :asset-path "/js/compiled"

        ;; modules created from Clojurescript sources
```

```
 :modules {:main {:entries [core.app]}}

 ;; set up development related configuration
 :devtools
 {
  ;; before live-reloading any code call this function
  :before-load core.app/stop

  ;; after live-reloading finishes call this function
  :after-load  core.app/start

  ;; serve the public directory over http at port 3000
  :http-port   3000
  :http-root   "public"

  ;; initialize devtools
  :preloads    [devtools.preload]}}}}
```

Now we are ready to start the development server.

```
$ npx shadow-cljs watch app

shadow-cljs - HTTP server available at http://localhost:3000
shadow-cljs - server version: 2.21.0 running at http://localhost:9630
shadow-cljs - nREPL server started on port 35837
shadow-cljs - watching build :app
[:app] Configuring build.
[:app] Compiling ...
[:app] Build completed. (144 files, 0 compiled, 0 warnings, 1.83s)
```

Navigate to `localhost:3000` to load the `index.html` file to your browser via our development server.

5.2 Setup VSCode and Calva

Now we are ready to set up our editor to interact with oure application. Lastly lets setup a connection between our application and our text editor.

Open the VSCode command prompt with and search for:

```
Connect to a running REPL server in your project
```

Select `app`, `shadow-cljs`, `:app` when prompetd and you should be ready to go. You can confirm by evaluating a Clojure form in your editor.

```
(+ 1 1)
```

Move your cursor over or inside the parenthesis and press `ALT+Enter`. If you see the number `2` floating around the cursor after this you've connected to the Clojure REPL successfully.

5.3 Javascript Interop

Now that we have the editor connected to the browser let's take a look how to talk with the browser in Clojurescript. In practice it is as simple as prefixing every browser's JS API command with `js/` and call the method as you would do in JS. For example evaluating the following code in should prompt the alert window in the browser.

```
(js/alert 1)
```

To access values like the `document.location`

```
js/document.location
;; => #object[Location http://localhost:3000/]
```

Or simply using browsers JS console to log some values.

```
(js/console.log 123)
```

5.3.1 Interactin With the DOM

Let's create an input element dynamically from our editor and update the DOM on the fly in the running browser.

```
;; lets create an input element
(def el (js/document.createElement "input"))
;; => [#object[HTMLInputElement [object HTMLInputElement]]]

(set! (.-id el)) "input"
;; => "input"

(.appendChild (js/document.getElementById "app") el)
;; => #object[HTMLInputElement [object HTMLInputElement]]
```

By this point you should have a new input field in the browser window. Next change the input's value from the editor by evaluating the following

```
;; lets update the value of that input
(set! (.-value el) "some value from the browser")
```

You should see the new value in the browser.. Magic! One last experiment, edit the input's value in the browser and see if you can retrieve the updated value dynamically in your editor.

```
(.. (js/document.getElementById "input") -value)
;; => "some edited value from the browser"
```

Voila! This is the magic behind Clojure REPL. Interacting with the application in real time from your editor without needing to refresh the whole application for every change. Let's continue on to the main event, setting up React rendering with Clojurescript.

6 Configure React

We'll be using the Helix library as our React wrapper of choice and there's a reason for this. This is only a thin layer of Clojurescript to interact with the React API so there's no extra complexity introduced. It is common to use Reagent that is another Clojurescript React wrapper that introduces it's own philosophy and quirks. For now, it's enough that you know it exists but no need to delve any deeper at this point.

6.1 Installing dependencies

The only Clojurescript dependency we need to get started with is Helix, add it to your dependencies in `shadow-cljs.edn` file. You can find the latest version from `https://clojars.org/lilactown/helix`.

```
{...
 :dependencies [[binaryage/devtools "0.9.7"]
                [cider/cider-nrepl "0.28.1"]
                ;; React wrapper
                [lilactown/helix "0.1.10"] ...}
```

We also need the matching JS counterpart `react` and `react-dom`.

```
$ npm install react react-dom
```

Clojurescript dependencies can be loaded dynamically but for us to have access to the updated JS dependecies we need to restart our development server.

6.2 Setup React Rendering

Let's start by creating a new file for utility functions where we define our render function we can use to mount our application into the DOM.

```
1  (ns app.utils
2    (:require [helix.core :refer [$]]
3              ["react" :as react]
4              ["react-dom/client" :as rdom]))
5
6  (defn app-container []
7    (js/document.getElementById "app"))
8
9  (defonce root (atom nil))
10
11 (defn react-root []
12   (when-not @root
13     (reset! root (rdom/createRoot (app-container))))
14   @root)
15
16 (defn render
17   [App]
18   (.render (react-root)
19            ($ react/StrictMode
20               ($ App))))
```

I'll explain the steps we took starting from the bottom. Line 16 defines a render function that takes a React component as the only argument that will be mounted as the application root wrapped in `React.StrictMode` and then we are calling the render method of react-root created with `react-dom/client` modules `createRoot` function.

We get the root instance with `react-root` function defined on line `11`. It initializes the value in the root atom we defined if the value is not already initialized and then returning it. Clojure reference describes atoms as:

> Atoms provide a way to manage shared, synchronous, independent state. They are a reference type like refs and vars. ... Changes to atoms are always free of race conditions.
>
> `https://clojure.org/reference/atoms`

On the line 9 we define a variable `root` with defonce that defines a variable if and only if the value is not already defined.

```
cljs.user=> (defonce once 1)
[1]
cljs.user=> (defonce once 2)
nil
```

In practice we are creating a singleton "instance" for react-root that is accessed by calling the `react-root` function. `helix.core/$` is a macro that renders make a React element out of the given component but let's explore that in moment!

```
(defn ^:dev/after-load init
  "This function is used in the `index.html`
  to load the application."
  []
  (.render (react-root)
           ($ react/StrictMode ($ App))))
```

Now that we have the rendering covered we can create the first component, `App`.

```
1  (ns app.core
2    (:require [helix.core :refer [defnc $]]
3              [helix.dom :as d]
4              ;; import the namespace (module)
5              ;; with the render function
6              [app.utils :as utils]))
7
8  (defnc App []
9    (d/div "This is a React component."))
10
11 (defn ^:dev/after-load init []
12   ;;
13   (utils/render ($ App)))
```

Here the `defnc` macro on the creates a `React.FunctionalComponent` that we can then render with the `$` macro. Our init function has the meta `^:dev/after-load` keyword to tell `shadow-cljs` that whenever the source files are loaded this should be evaluated. The `App` components produces the same results as the following React component.

```
const App = () => {
  return <div>This is a React component.</div>
}
```

Evaluating our `App` creates a valid React type.

```
=> App

function app$core$App_render(props__12584__auto__,maybe_ref__12585__auto__){
    var vec__25299 = new cljs.core.PersistentVector(
        null,
        2,
        5,
        cljs.core.PersistentVector.EMPTY_NODE,
        [helix.core.extract_cljs_props.call(
            null,
            props__12584__auto__),
         maybe_ref__12585__auto__],
        null);

    return helix
        .core
        .get_react
        .call(null)
        .createElement("div",null,"This is a React component.");
}
```

Create a new React element from a valid React type.

```
=> ($ App)
{"$$typeof" "Symbol(react.element)",
 "type" #object[app$core$App_render],
 "key" nil,
 "ref" nil,
 "props" #js {},
```

```
 "_owner" nil,
 "_store" #js {}}
```

And if we compare this to the JS

```
const react = require("react");

const App = () => {
    return react.createElement("div", null, "This is a React component");
};

console.log(App);
console.log(App());
```

```
[Function: App]
{
  '$$typeof': Symbol(react.element),
  type: 'div',
  key: null,
  ref: null,
  props: { children: 'This is a React component' },
  _owner: null,
  _store: {}
}
```

We can see how these relate.

6.3 Simple List App

First create a new file for these ‘app/starwars.cljs‘ and import required libraries.

```
(ns app.starwars
  (:require [helix.core :refer [defnc $]]
            [helix.dom :as d]))

  ;; Let's start with listing the characters in a variable and
  (def people [{:name "Luke" :details "Luke's story"}
               {:name "Chewbacca" :details "Chwebacca's story"}
               {:name "C-3PO" :details "C-3PO's story"}])

```

```
10
11    ;; Create React.FunctionalComponent
12    (defnc People
13      []
14      ;; define state for selecting the detail
15      ;; hooks/use-state is a wrapper for React.useState
16      ;; we could also use React.useState here and it would work
17      (let [[selected set-selected] (hooks/use-state nil)]
18        (d/div
19         (d/h1 "Starwars People")
20         (d/ul
21          ;; iterate over people and create <li>{name}</li> for each
22          (for [{:keys [name] :as person} people]
23            (d/li {:key name
24                   ;; on click detail, set the clicked person as selected
25                   :on-click #(set-selected person)} name)))
26         ;; if person selected show the details
27         (when selected
28           (d/div
29            (:details selected))))))
```

Import the starwars namespace into the core

```
(ns app.core
  (:require [helix.core :refer [defnc $]]
            [helix.dom :as d]
            ;; import the starwars
            [app.starwars :as sw]
            ["react" :as react]
            ["react-dom/client" :as rdom]))
```

Replace the `App` with `sw/People`.

```
(defn ^:dev/after-load init
  "This function is used in the `index.html`
  to load the application."
  []
  (.render (react-root)
           ;; `$` is a macro to make a React
           ;; element out of the given component
           ($ react/StrictMode ($ sw/People))))
```

6.3.1 Fetch people from SWAPI

```
(ns app.starwars
  (:require [helix.core :refer [defnc $]]
            [helix.dom :as d]
            ;; add hooks and pprint
            [helix.hooks :as hooks]
            [clojure.pprint :refer [pprint]]))

  (defn fetch-people
    "
    Just a wrapper for

    fetch(URL)
       .then((response) => response.json())
       .then((data) => console.log(data))"
    []
    (-> (js/fetch "https://swapi.dev/api/people")
        (.then (fn [response] (.json response)))
        (.then (fn [data]
                 (js->clj data.results
                          :keywordize-keys true)))))

  (defnc PeopleFromAPI
    []
    (let [[people set-people] (hooks/use-state [])
          [selected set-selected] (hooks/use-state nil)]

      ;; React.useEffect
      (hooks/use-effect
       ;; run this hook only once
       :once
       (fn [] ;; fetch the people
         (-> (fetch-people)
             ;; and set-people with the result data
             (.then set-people))))
      (d/div
       (d/h1 "Starwars People")
       (d/ul
        (for [{:keys [name] :as person} people]
```

```
39            (d/li {:key      name
40                   :on-click #(set-selected person)} name)))
41        (when selected
42          (d/pre
43           ;; think this as JSON.stringify(selected)
44           (with-out-str (pprint selected)))))))
```

Replace the `sw/People` with `sw/PeopleFromAPI`.

```
(defn ^:dev/after-load init
  "This function is used in the `index.html`
  to load the application."
  []
  (.render (react-root)
           ;; `$` is a macro to make a React
           ;; element out of the given component
           ($ react/StrictMode ($ sw/PeopleFromAPI))))
```

7 Common Libraries and Advanced Interop

Let's take a look how to use a couple of the common React libraries in CLJS and learn by example how the inter-op looks like and how to translate between JS and CLJS.

7.1 React Router

```
1  (ns app.react-router-app
2    (:require [helix.core :refer [fnc defnc $ <>]]
3              [helix.dom :as d]
4              [app.utils :as utils]
5              ["react" :as react]
6              ["react-dom/client" :as rdom]
7              ["react-router-dom" :refer [Navigate
8                                          createBrowserRouter
9                                          RouterProvider]]))
10
11 (defnc Root []
12   (<>
13    (d/nav
14     (d/a {:href "/"} "landing")
```

```
     (d/a {:href "/people"} "people"))
    (d/div {:id "container"})))

(def router
  (createBrowserRouter
   (clj->js [{:path "/react-router-app.html"
              :element ($ Navigate {:to "/"})}
             {:path "/"
              :element ($ Root)
              :children [{:path "/people"
                          :element (fnc []
                                        (d/div "people"))}]}])))

(defn ^:dev/after-load init
  []
  (utils/render
   ($ RouterProvider {:router router})))
```

7.2 React Query

Install the library

```
npm install react-query
```

https://shadow-cljs.github.io/docs/UsersGuide.html#_using_npm_packages

Create new file `query.cljs` where we'll setup a lightweight wrapper for React Query so we do not need to worry about the interop at the view or component level.

```
(ns app.query
  "Lightweight CLJS wrapper for React Query"
  (:require ["@tanstack/react-query" :as react-query]))

;; define the query client
(def query-client-provider
  (react-query/QueryClientProvider.))

(defonce query-client (react-query/QueryClient.))

(defn use-query
```

```
  "create clojure wrapper for useQuery"
  [query-key query-fn]
  (let [result (react-query/useQuery
                ;; useQuery is expecting a JS object
                ;; instead of CLJS map
                #js {:queryFn  query-fn
                     :queryKey (into-array query-key)})]
    {:data result.data
     :loading? result.isLoading}))
```

Now that we have our own interface for the library let's put it to use.

```
(ns app.react-query-app
  (:require [helix.core :refer [defnc $]]
            [helix.dom :as d]
            [app.utils :as utils]
            [app.query :refer [query-client-provider
                               query-client
                               use-query]]
            ["react" :as react]
            ["react-dom/client" :as rdom]
            [helix.hooks :as hooks]
            [clojure.pprint :refer [pprint]]
            ["@tanstack/react-query" :as react-query]))

;; TODO make this working

(defnc PeopleWithReactQuery
  []
  (let [[selected set-selected] (hooks/use-state nil)
        ;; use react-query to handle the query state
        {people :data
         loading? :loading?} (use-query ["people"] fetch-people)]
    (if loading?
      (d/div "Loading...")
      (d/div
       (d/h1 "Starwars People")
       (d/ul
        (for [{:keys [name] :as person} people]
          (d/li {:key
```

```
29                    name
30                    :on-click #(set-selected person)} name)))
31          (when selected
32            (d/pre
33             (with-out-str
34               (pprint selected))))))))
35
36  (defnc WrapQueryClient [{:keys [children]}]
37    ($ query-client-provider {:client query-client}
38       children))
39
40  ;; and it's a wrap
41  (defnc QueryApp []
42    ($ WrapQueryClient
43       ($ PeopleWithReactQuery)))
44
45  (defn ^:dev/after-load init
46    []
47    (utils/render ($ QueryApp)))
```

Replace the `sw/PeopleFromAPI` with `sw/StarWarsApp`.

```
(defn ^:dev/after-load init
  "This function is used in the `index.html`
  to load the application."
  []
  (.render (react-root)
           ;; `$` is a macro to make a React
           ;; element out of the given component
           ($ react/StrictMode ($ sw/StarWarsApp))))
```

7.3 React Intl

8 State Management

Since you've coming from React you have probably asked the question but "what about redux?" and by now you have all the know-how how to use Redux in the case you really want to. What I'm going to do, is to introduce you to the Clojure way of doing state management.

9 Setup Testing

First ew need to setup a test runner. Add new build target called for the tests. This configuration sets up the tests so that we get a test runner in the browser at localhost:3001

```
{...
 :builds {
          ....
          :test {:target           :browser-test
                 :test-dir         "out/test"
                 :devtools {:http-root "out/test"
                            :http-port 3001}
                 :compiler-options {:output-feature-set :es8}}
          }}
```

Start by creating the folder for `app` tests and the first test file.

```
mkdir -p src/test/app
touch src/test/app/starwars_test.cljs
```

Test files are just Clojure files that use the testing libraries. Here we import a few test functions/macros that are meant for defining tests and for assertion.

```
(ns app.starwars-test
  (:require [cljs.test :refer [deftest testing is]]))

(deftest test-example
  (testing "assertion works"
    (is (= 0 1))))
```

Start the test runner in watch mode to make the tests run on file changes.

```
$ npx shadow-cljs watch :test
```

Now if you navigate to your `localhost:3001` you should see a view with a report of one failing test.

Now lets assume that we need to create a filter function for the starwars people to be used in our component. Replace the `test-example` with the following.

```
(deftest filter-by-name
  (testing "filter by name works as expected"
    (let [people [{:name "Luke"} {:name "Chewbacca"} {:name "C-3PO"}]
          expected [{:name "Luke"}]]
    (is (= expected
           (sw/filter-by-name "uke" people))))))
```

And then write the implementation

```
(ns app.starwars
  (:require ...
            [clojure.string :as str]))

(defn filter-by-name [search people]
  (filter (fn [person]
            (str/includes? (:name person)
                           search))
          people))
```

And now you should see one passing test in the report view.
Now we want to add a input field to let the user use the filter
Let's set some visual tools to help with this.

10 Devcards

Devcards is a Clojurescript library that allows us to render our components outside the application context and it provides a catalog of these components that can be used as development environment. Let's set up the rendering for devcards under `src/dev`.

```
mkdir src/dev/app
touch src/dev/app/devcards.cljs
```

Configure the devcards to re-render on `^:dev/after-load` as we did earlier for the application.

```
(ns app.devcards
  (:require [devcards.core :as dc :include-macros true]
            ;; load the namespaces with devcards
```

```
            [app.starwars-test]))

(defn ^:dev/after-load start! []
  (dc/start-devcard-ui!))

(defn init! [] (start!))

(init!)
```

Add devcards to shadow-cljs dependencies for "storybook" like features.

```
{...
 :dependencies [...
                [devcards "0.2.5"]]
 :builds {...
          {:target           :browser
           :output-dir       "public/js/compiled"
           :asset-path       "/js/compiled"
           :compiler-options {:devcards           true
                              :output-feature-set :es8}
           :devtools {:after-load app.devcards/init!}
           :modules {:dev {:entries [app.devcards]}}}}
```

Install devcards javascript dependencies.

```
npm install showdown
```

And lastly create a new html file for loading the devcards separate from the app itself.

```
<!DOCTYPE html>
<html>
    <body>
        <div id="app"/>
        <script src="/js/compiled/dev.js" type="text/javascript"></script>
    </body>
</html>
```

Create the new component with filtering.

```
(defnc PeopleFiltering
  []
  ;; define state for selecting the detail
  (let [[selected set-selected] (hooks/use-state nil)
        [filter set-filter] (hooks/use-state "")]
    (d/div
     (d/h1 "Starwars People")
     (d/input {:on-change (fn [e] (set-filter e.target.value))})
     (d/p "Searching for: " filter)
     (d/ul
      ;; iterate over people and create <li>{name}</li> for each
      (for [{:keys [name] :as person} (filter-by-name filter people)]
        (d/li {:key      name
               ;; on click detail, set the clicked person as selected
               :on-click #(set-selected person)} name)))
     ;; if person selected show the details
     (when selected
       (d/div
        (:details selected))))))
```

Now we can write our first devcard.

```
(ns app.starwars-test
  (:require [cljs.test :refer [testing is]]
            [helix.core :refer [$]]
            [devcards.core :as dc :include-macros true]
            [app.starwars :as sw]))

(def people [{:name "Luke"} {:name "Chewbacca"} {:name "C-3PO"}])

;; describe the data
(dc/defcard people
  people)

;; use dc/deftest instead of cljs.test to
;; render the test results with the devcards
(dc/deftest filter-by-name
  (let [people [{:name "Luke"} {:name "Chewbacca"} {:name "C-3PO"}]]
    (testing "filter by name works as expected"
      (is (= [{:name "Luke"}]
```

```
19              (sw/filter-by-name "uke" people))))
20     (testing "filter by name works case insensitive"
21       (is (= [{:name "Luke"}]
22              (sw/filter-by-name "luke" people))))))
23
24 ;; Render the component under development
25 (dc/defcard PeopleWithFiltering
26   ($ sw/PeopleFiltering))
```

11 Building the Production Version

11.1 Build Targets

11.2 Running Tests in CI

11.3 Build Production Bundle

12 Outro

Now that we've gone through all this it's time to wrap up and give you some pointers where to go next with your Clojure adventure.