Clojure For React Developers

Toni Väisänen

September 3, 2023

Contents

1 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
```

1.1 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 to describe what is the namespace used for, similar to function docstrings.

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

As in Javascript and Typescript you can import code from other modules. Required namespace can be aliased with the `:as` keyword and then used with `alias/var-name` syntax.

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

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

Node modules can be imported imported and used as in JS by using the quoted library name when requiring the module.

```
cljs.user=> (require '["fs" :as fs])
cljs.user=> (fs/readdirSync ".")
[".calva" ".clj-kondo" ".gitignore" ".lsp" ".shadow-cljs"
 "node_modules" "out" "package-lock.json" "package.json"
 "shadow-cljs.edn" "src"]
```

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

1.2 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
```

1.3 Data Types

I recommend using these as reference as we go further, since this intro is just a quick overview to introduce basics:

- Clojurescript Cheatsheet: `https://cljs.info/cheatsheet/`
- Clojurescript API: `http://cljs.github.io/api/`

1.3.1 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 useful in cases where in JS you'd reach out for the or operator.

```
const var = obj.maybeKey || "default-value"
```

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

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

In the case there's more than one possible `undefined` value in this chain.

```
const var = obj.maybeKey || obj.secondMaybe || "default-value"
```

In Clojure just reach out to `or`.

```
cljs.user=> (def obj {})
cljs.user=> (or (:maybe-key obj) (:second-maybe obj) "default-value")
"default-value"
```

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}
```

1.3.2 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 ().

```
cljs.user=> (type [])
cljs.core/PersistentVector

cljs.user=> (type ())
cljs.core/EmptyList

cljs.user=> (type (1 2))
[true ""]:repl/exception! ;; here we got an error!

cljs.user=> (type '(1 2))
cljs.core/List
```

Empty list can be initialized with just the parenthesis `()` but if there's any elements in the list a quote `'` or `quote` needs to be used.

```
cljs.user=> (quote (1 2 3))
(1 2 3)
cljs.user=> '(1 2 3)
(1 2 3)

(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 are not using the prototype methods as in JS. Instead 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 about 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
(def data (array 1 2 3))

(.map (array 1 2 3) (fn [value] (+ value 1)))
(array.map (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.

1.4 Branching

Here's a few of the branching options, look up the rest from the cheatsheet!

```
cljs.user=> (if (true? true) "true" "false")
"true"

cljs.user=> (if-not (true? true) "true" "false")
"false"

cljs.user=> (when true "continue")
"continue"

cljs.user=> (when false "do nothing")
nil

cljs.user=> (when-not true "reverse 'when'")
nil

cljs.user=> (when-let [value 1] (inc value))
2

cljs.user=> (when-let [value nil] (inc value))
nil

cljs.user=> (when-first [value []] :found-value)
nil

cljs.user=> (when-first [value [:first :second]] value)
:first

cljs.user=> (if-let [value 2] (inc value) 0)
3

cljs.user=> (if-let [value nil] (inc value) 0)
0
```

1.5 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.

2 Project

We already created the project for the REPL exploration so let's just use that one to continue. The project structure should look something like this.

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

And `package.json` should have `shadow-cljs` as a dev dependency.

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

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

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

 :dependencies
 []

 :builds
 {}}
```

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

But let's start by writing some source code to compile and creating an HTML file to load the soon compiled javascript bundle into the browser.

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

And here's some Clojurescript code for the 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"))
```

In the content you might have noticed the mystical `js/console.log` Just a moment...

Create the public folder and index.html file.

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

The index file could look like this.

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

2.1 Set browser build target

And finally lets configure the build target in `shadow-cljs.edn` 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
                ;; (unless you have the latest version of Firefox
                ;;  with custom formatters enabled)
                [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 [app.core]}}

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

         ;; after live-reloading finishes call this function
         :after-load  app.core/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, and open up the devtools to see if everything up and running as expexted.

2.2 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` and if this doesn't work on your operating system try searching the right command from VS code's command prompt with "Calva evaluate".

If you see the number `2` floating around the cursor after this you've connected to the Clojure REPL successfully.

2.3 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 print values to browser console using JS `console.log` function from the REPL.

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

2.3.1 Interacting With the DOM

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

You can do this in the REPL or write the code in the `core.cljs` file and evaluate the code with Calva.

```
;; 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.

3 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.

3.1 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.

3.2 Setup React Rendering

Let's start by creating a new file `src/main/app/utils.cljs` 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 creates React elements from valid types but let's explore that a bit more in moment! 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              [app.utils :as utils]))
5
6  (defnc App []
7    (d/div "This is a React component."))
8
9  (defn ^:dev/after-load init []
10   (utils/render App))
```

If VSCode is warning on the App we need to import Helix's `clj-kondo` configurations.

For this we need to create new file called `deps.edn` in the project root directory.

```
{:deps {lilactown/helix {:mvn/version "0.1.10"}}}
```

And next we run `clj-kondo` as instructed by the documentation. `https://cljdoc.org/d/clj-kondo/clj-kondo/2023.03.17/doc/configuration#importing`

```
$ clj-kondo --lint "$(clojure -Spath)" --copy-configs --skip-lint
$ clj-kondo --lint $(clojure -Spath) --dependencies --parallel
```

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.

3.3 Build an App

TODO: layout of the app

```
|  nav | nav    |
| list | detail |
```

We need a navbar, listview and a detail view.

Let's start by creating the template with navigation and the content container.

TODO: Should I setup the routing from the get go. Probably!

Let's start on the navigation part. First let's see how we can use `react-router` to define the frontend routes.

Here we'll be doing some JS/CLJS interop.

But first things first, install `react-router`

```
npm install react-routerv6
```

```clojure
(ns app.router
  (:require [helix.core :refer [defnc $]]
            ["react-router-dom" :refer [createBrowserRouter
                                        RouterProvider
                                        NavLink
                                        Outlet]]))

(defn make-router
  "Create react-router/BrowserRouter from given routes."
  [routes]
  (createBrowserRouter (clj->js routes)))

(defnc WrapRouter
  "Make routes and wrap in a RouterProvider"
  [{:keys [routes]}]
  ($ RouterProvider
     {:router (make-router routes)}))

(defnc Link
  "Wrapper react-router/NavLink"
  [{:keys [to label] :as props}]

  ($ NavLink {:to to
              & props}
     label))

(defnc RouteContent
  "Render the route content."
  []
  ($ Outlet))
```

Now that we have dealed with the interop part we are ready to define the routes and start the work on our first component, the Navbar.

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

(defnc Navbar [{:keys [navlinks]}]
```

```
  (d/nav
   (for [{:keys [to label]} navlinks]
     ($ router/Link {:key to
                     :to to
                     :label label}))))
```

Here we have a simple renderer for Navlinks. Next, Put this into play and define the routes in the `core.cljs` file.

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

(def routes
  [{:to "/movies"
    :label "movies"
    :element (d/div "movies")}
   {:to "/people"
    :label "people"
    :element (d/div "people")}])

(defnc Layout []
  (d/div
   ($ nav/Navbar {:navlinks routes})
   ($ router/RouteContent)))

(defnc App []
  ($ router/WrapRouter
     {:routes [{:path "/"
                :element ($ Layout)
                :children (for [{:keys [to element]} routes]
                            {:path to
                             :element element})}]}))

(defn ^:dev/after-load init
  "This function is used in the `index.html`
  to load the application."
```

```
32    []
33    (utils/render App))
```

Now you should be able to have a working frontend navigation in your application between the movies and people pages.

Let's get some data for rendering the page content. For that we need a function to make http requests. We'll be using the browser `fetch` for this.

```
1   (ns app.http)
2
3   (defn fetch
4     "Browser fetch wrapper
5
6     Same as:
7
8     fetch(URL)
9        .then((response) => response.json())
10       .then((data) => console.log(data))"
11    ([url] (fetch url {}))
12    ([url opts]
13     (-> (js/fetch url (clj->js opts))
14         (.then (fn [response]
15                  (.json response)))
16         (.then (fn [data]
17                  (js->clj data :keywordize-keys true))))))
```

Now we are ready to make our first HTTP query!

```
1   (ns app.movies
2     (:require [app.http :as http]
3               [helix.core :refer [defnc $]]
4               [helix.dom :as d]
5               [helix.hooks :as hooks]
6               ["react" :as react]))
7
8   (defnc MovieList []
9     (let [[movies set-movies] (react/useState [])]
10
11      (hooks/use-effect
12       :once
13       (-> (http/fetch "https://swapi.dev/api/films")
```

```
          (.then (fn [data]
                   (set-movies (:results data))))))

    (d/ul
     (for [film movies]
       (d/li
        (d/strong (:title film))
        (d/p " by " (:director film)))))))
```

Nobody wants to deal with managing the queries themselves so let's just create a wrapper for the `react-query` library to make our lives a lot easier in the long term.

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

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

(defn use-query
  "create clojure wrapper for useQuery"
  [{:keys [query-key query-fn]}]
  (let [result (react-query/useQuery
                #js {:queryFn  query-fn
                     :queryKey (into-array query-key)})]
    (js/console.log result)
    {:data result.data
     :loading? result.isLoading}))

(defnc WrapQueryClientProvider [{:keys [children]}]
  ($ react-query/QueryClientProvider {:client query-client}
     children))
```

And now lets update the movielist component to use the new query wrapper

```
(ns app.movies
  (:require [app.http :as http]
            [app.query :as query]
            [helix.core :refer [defnc $]]
```

```
            [helix.dom :as d]
            ["react" :as react]))

(def url  "https://swapi.dev/api/films")

(defnc MovieList []
  (let [{:keys [data]} (query/use-query {:query-key [url]
                                         :query-fn #(http/fetch url)})]
    (d/ul
     (for [film (:results data)]
       (d/li
        (d/strong (:title film))
        (d/p " by " (:director film)))))))
```

Let's add a detail view to make a dynamic view.

```
(ns app.movies
  (:require [app.http :as http]
            [app.query :as query]
            [clojure.string :as str]
            [cljs-bean.core :refer [bean]]
            [helix.core :refer [defnc $]]
            [helix.dom :as d]
            ["react" :as react]
            ["react-router-dom" :refer [useNavigate
                                        useParams]]))

(def url  "https://swapi.dev/api/films")

(defn movie-detail-url [id]
  (str url "/" id "/"))

(defnc MovieList []
  (let [navigate (useNavigate)
        {:keys [data]} (query/use-query {:query-key [url]
                                         :query-fn #(http/fetch url)})]
    (d/ul
     (for [film (:results data)]
       (let [resource-id (last (str/split (:url film) #"/"))]
         (d/li {:key (:id film)
```

```
                 :on-click #(navigate (str "/movies/" resource-id))}
                (d/strong (:title film))
                (d/p " by " (:director film))))))))

(defnc MovieDetail []
  (let [params (bean (useParams))
        movie-detail-url (movie-detail-url (:id params))
        query-params {:query-key [movie-detail-url]
                      :query-fn #(http/fetch movie-detail-url)}
        {movie :data} (query/use-query query-params) ]
    (d/pre (str movie))))
```