# ClojureScript

## Contents

---

ClojureScript is an excellent alternative to JavaScript for client side application logic. Some of the advantages of using ClojureScript include:

use the same language on both the client and the server
share common code between the front-end and back-end
cleaner and more consistent language
dependency management via Leiningen
immutable data structures
powerful standard library

### Adding ClojureScript Support

The easiest way to add ClojureScript support is by using the one of the ClojureScript profile flags when creating a new project.

However, it's quite easy to add it to an existing project as well.

Build Tool: lein

Topics

Your First Application
REPL Driven Developme
Application Profiles
HTML Templating
Static Assets
• **ClojureScript**
Routing
RESTful Services
Request types
Response types
Websockets
Middleware
Sessions and Cookies
Input Validation
Security
Component Lifecycle
Database Access
Database Migrations
Logging
Internationalization
Testing
Server Tuning
Environment Variables
Deployment
Useful Libraries
Sample Applications
Upgrading
Clojure Resources

Books

First, add the lein-cljsbuild plugin and `:cljsbuild` key to the project as seen below:

```
:plugins [... [lein-cljsbuild "1.1.3"]]

:resource-paths ["resources" "target/cljsbuild"]
:cljsbuild
{:builds
   {:app
    {:source-paths ["src/cljs"]
     :compiler
                  {:main          (str project-ns ".app")
                   :asset-path    "/js/out"
                   :output-to     "target/cljsbuild/public/js/app.js"
                   :output-dir    "target/cljsbuild/public/js/out"
                   :optimizations :none
                   :source-map    true
                   :pretty-print  true}}
    :min
    {:source-paths ["src/cljs"]
     :compiler
                  {:output-to     "target/cljsbuild/public/js/app.js"
                   :output-dir    "target/uberjar"
                   :externs       ["react/externs/react.js"]
                   :optimizations :advanced
                   :pretty-print  false}}}}
```

The ClojureScript sources are expected to be found under the `src/cljs` source path in the above configuration. Note that ClojureScript files **must** end with the `.cljs` extension. If the file ends with `.clj` it will still compile, but it will not have access to the `js` namespace.

The compiled JavaScript file will be available in the `/js/app.js` resource path and can be referenced on the page as follows:

```
{% script "/js/app.js" %}
```

Next, update the `:uberjar` profile with the following options:

```
:prep-tasks ["compile" ["cljsbuild" "once" "min"]]
```

The above will add the `lein-cljsbuild` hook to the `:uberjar` profile so that ClojureScript is compiled when `lein uberjar` is run.

### Using Libraries

One advantage of using ClojureScript is that it allows managing your client-side libraries using Leiningen. ClojureScript libraries are included under dependencies in the `project.clj` just like any other library.

### Running the Compiler

The easiest way to develop ClojureScript applications is to run the compiler in `auto` mode. This way any changes you make in your namespaces will be recompiled automatically and become immediately available on the page. To start the compiler in this mode simply run:

```
lein cljsbuild auto
```

Make sure to run the `clean` option before packaging the application for production using `lein uberjar`. This will ensure that any existing artifacts are removed before the production JavaScript is compiled:

```
lein cljsbuild once
```

### Live Code Reloading

A more advanced approach is to setup Figwheel to hot load the code in the browser. The easiest way to get Figwheel support is by using a ClojureScript profile when creating your Luminus project.

Figwheel requires that the server to be running:

```
lein run
```

Once the server starts simply run:

```
lein figwheel
```

This will start Figwheel and connect a browser REPL. Any changes you make in ClojureScript source will now be automatically reloaded on the page.

### ClojureScript with nREPL

To connect the IDE to a ClojureScript REPL make sure that you have the `:nrepl-port` key in your `:figwheel` config in `project.clj`. This key defaults to port `7002`. When Figwheel starts, it will open nREPL on the specified port.

Luminus also sets up the scaffolding for running the Figwheel compiler from the REPL. When you generate a project using one of the ClojureScript flags, then a `env/dev/clj/<app>/figwheel.clj` namespace will be generated. This namespace provides functions to manage the Figwheel compiler and run the ClojureScript REPL. This allows you to connect any REPL aware editor to the ClojureScript REPL.

Once you run `lein figwheel`, then you'll be able to connect to its nREPL at `localhost:7002`. Once connected, you simply have to run `(cljs)` and the ClojureScript nREPL will become available. You can test that everything is working correctly by running `(js/alert "Hi")` in the REPl. This should pop up an alert in the browser.

Alternatively, the compiler can be started from a regular REPL using the `start-fw` function and stopped using the `stop-fw` function. The ClojureScript REPL is started by running the `cljs` function after `start-fw` has run successfully. These functions will be available in the `user` namespace. The REPL will default to it when it starts:

```
user=> (start-fw)
Figwheel: Starting server at http://localhost:3449
Figwheel: Watching build - app
Compiling "target/cljsbuild/public/js/app.js" from ("src/cljc" "src/cl
Successfully compiled "target/cljsbuild/public/js/app.js" in 7.583 sec
Figwheel: Starting CSS Watcher for paths  ["resources/public/css"]
Figwheel: Starting nREPL server on port: 7002
nil
user=> (cljs)
Launching ClojureScript REPL for build: app
Figwheel Controls:
          (stop-autobuild)                ;; stops Figwheel autobuilde
          (start-autobuild [id ...])      ;; starts autobuilder focuse
          (switch-to-build id ...)        ;; switches autobuilder to d
          (reset-autobuild)               ;; stops, cleans, and starts
          (reload-config)                 ;; reloads build config and
          (build-once [id ...])           ;; builds source one time
          (clean-builds [id ..])          ;; deletes compiled cljs tar
          (print-config [id ...])         ;; prints out build configur
          (fig-status)                    ;; displays current state of
  Switch REPL build focus:
          :cljs/quit                      ;; allows you to switch REPL
    Docs: (doc function-name-here)
    Exit: Control+C or :cljs/quit
 Results: Stored in vars *1, *2, *3, *e holds last exception object
Prompt will show when Figwheel connects to your application
```

### Advanced Compilation and Exports

During advanced compilation variable names will be munged by the compiler to shorten the code. If we wish to expose any functions to JavaScript we have to ensure that their names are protected. This is done by using the `^:export` annotation, eg:

```
(ns main)

(defn ^:export init []
  (js/alert "hello world"))
```

We can now call this function from our page like any other:

```
<script>
main.init();
</script>
```

If we use a Js library in our code we must protect the names of any functions we call from it as well. For example, if we wanted to use the AlbumColors library, we could write the following:

```
(defn ^:export init []
  (.getColors (js/AlbumColors. "/img/foo.jpg")
    (fn [[background]]
     (.log js/console background))))
```

However, when the script is compiled with the `:advanced` flag, the `AlbumColors` and `getColors` will be munged.

To protect them we have to create a Js file with the names we'd like to protect and reference it in our build:

```
var AlbumColors = {};
AlbumColors.getColors = function() {};
```

Note that in most cases it's possible to simply use the JavaScript library as its own externs file without the need to manually write out each function used.

If we put the above code in a file called `externs.js` under the `resources` directory then we would reference it in our `cljsbuild` section as follows:

```
{:id "release"
 :source-paths ["src/cljs"]
 :compiler
 {:output-to "target/cljsbuild/public/js/app.js"
  :optimizations :advanced
  :pretty-print false
  :output-wrapper false
  ;;specify the externs file to protect function names
  :externs ["resources/my-externs.js"]
  :closure-warnings {:externs-validation :off
                     :non-standard-jsdoc :off}}}
```

A useful site for extracting externs can be found here.

Please see the official documentation for more information.

## Interacting with JavaScript

All the global JavaScript functions and variables are available via the `js` namespace.

#### METHOD CALLS

```
(.method object params)

(.log js/console "hello world!")
```

#### ACCESSING PROPERTIES

```
(.-property object)

(.-style div)
```

#### SETTING PROPERTIES

```
(set! (.-property object))

(set! (.-color (.-style div) "#234567"))
```

For more examples of ClojureScript synonyms of common JavaScript operations see the ClojureScript Synonyms.

## Reagent

Reagent is the recommended approach for building ClojureScript applications with Luminus. Using the `+reagent` profile in Luminus will create an application with it configured.

Reagent is backed by React and provides an extremely efficient way to manipulate the DOM using Hiccup style syntax. In Reagent, each UI component is simply a data structure that represents a particular DOM element. By taking a DOM-centric view of the UI, Reagent makes writing composable components simple and intuitive.

A simple Reagent component looks as follows:

```
[:label "Hello World"]
```

Components can also be functions:

```
(defn label [text]
  [:label text])
```

The values of the components are stored in Reagent atoms. These atoms behave just like regular Clojure atoms, except for one important property. When an atom is updated it causes any components that dereference it to be repainted. Let's take a look at an example.

**Important:** Make sure that you require Reagent atom in the namespace, otherwise regular Clojure atoms will be used and components will not be repainted on change.

```
(ns myapp
  (:require [reagent.core :as reagent :refer [atom]]))

(def state (atom nil))

(defn input-field [label-text]
  [:div
    [label label-text]
    [:input {:type "text"
             :value @state
             :on-change #(reset! state (-> % .-target .-value))}]])
```

Above, the `input-field` component consists of a `label` component we defined earlier and an `:input` component. The input will update the `state` atom and render it as its value.

Notice that even though `label` is a function we're not calling it, but instead we're putting it in a vector. The reason for this is that we're specifying the component hierarchy. The components will be run by Reagent when they need to be rendered.

This is behavior makes it trivial to implement the React Flux pattern.

```
Views--->(actions) --> Dispatcher-->(callback)--> Stores---+
Ʌ                                                          |
|                                                          V
+--(event handlers update)--(Stores emit "change" events)--+
```

Our view components dispatch updates to the atoms, which represent the stores. The atoms in turn notify any components that dereference them when their state changes.

In the previous example, we used a global atom to hold the state. While it's convenient for small applications this approach doesn't scale well. Fortunately, Reagent allows us to have localized states in our components. Let's take a look at how this works.

```
(defn input-field [label-text id]
  (let [value (atom nil)]
    (fn []
      [:div
        [label "The value is: " @value]
        [:input {:type "text"
                 :value @value
                 :on-change #(reset! value (-> % .-target .-value))}]]
```

All we have to do is create a local binding for the atom inside a closure. The returned function is what's going to be called by Reagent when the value of the atom changes.

Finally, rendering components is accomplished by calling the `render-component` function:

```
(defn render-simple []
  (reagent/render-component [input-field]
                            (.-body js/document))
```

A working sample project can be found here. For a real world application using Reagent see the Yuggoth blog engine.

### Client Side Routing

Secretary is the recommended ClojureScript routing library. It uses Compojure-inspired syntax for route definitions. To use the library, We'll add the dependency to your project, if you created the project using a ClojureScript profile, then it will be included by default.

```
[secretary "1.2.0"]
```

Next, we have to reference the library in our ClojureScript namespace to use it.

```
(ns app
  (:require [secretary.core :as secretary
             :include-macros true
             :refer [defroute]]
            [goog.events :as events])
  (:import goog.History
           goog.history.EventType))
```

With the library imported we can create routes that will set the content of the specified DOM element when triggered.

```
(defn home []
  [:div [:h1 "Home"]])

(defn info []
  [:div [:h1 "About this app"]])

(defn not-found []
  [:div [:h1 "404: Page doesn't exist"]])

(defn page [page-component]
  (reagent/render-component
    [page-component]
    (.getElementById js/document "appContainer")))

(defroute home-path "/" [] (page home))
(defroute home-path "/about" [] (page info))
(defroute "*" [] (page not-found))
```

Please refer to the official documentation for further details.

### Working With the DOM directly

**WARNING**

Since Reagent uses a virtual DOM and renders components as necessary, direct manipulation of the DOM is highly discouraged. Updating DOM elements outside the Reagent components can result in unpredictable behavior.

That said, there are several libraries available for accessing and modifying DOM elements. In particular, you may wish to take a look at the Domina and Dommy. Domina is a lightweight library for selecting and manipulating DOM elements as well as handling events. Dommy is a templating library similar to Hiccup.

### Ajax

Luminus uses cljs-ajax for handling Ajax operations. The library provides an easy way to send Ajax queries to the server using `ajax-request`, `GET`, and `POST` functions.

### AJAX-REQUEST

The `ajax-request` is the base request function that accepts the following parameters:

uri - the URI for the request
method - a string representing the HTTP request type, eg: "PUT", "DELETE", etc.
format - a keyword indicating the response format, can be either `:raw`, `:json`, `:edn`, or `:transit` and defaults to `:transit`
handler - success handler, a function that accepts the response as a single argument
error-handler - error handler, a function that accepts a map representing the error with keys `:status` and `:status-text`
params - a map of params to be sent to the server

### GET/POST HELPERS

The `GET` and `POST` helpers accept a URI followed by a map of options:

`:handler` - the handler function for successful operation should accept a single parameter which is the deserialized response
`:error-handler` - the handler function for errors, should accept a map with keys `:status` and `:status-text`
`:format` - the format for the request can be either `:raw`, `:json`, `:edn`, or `:transit` and defaults to `:transit`
`:response-format` - the response format. If you leave this blank, it will detect the format from the Content-Type header
`:params` - a map of parameters that will be sent with the request
`:timeout` - the ajax call's timeout. 30 seconds if left blank
`:headers` - a map of the HTTP headers to set with the request
`:finally` - a function that takes no parameters and will be triggered during the callback in addition to any other handlers

```
(ns foo
  (:require [ajax.core :refer [GET POST]]))

(defn handler [response]
  (.log js/console (str response)))

(defn error-handler [{:keys [status status-text]}]
  (.log js/console
    (str "something bad happened: " status " " status-text)))

(GET "/hello")

(GET "/hello" {:handler handler})

(POST "/hello")
```

```
(POST "/send-message"
        {:headers {"Accept" "application/transit+json"}
         :params {:message "Hello World"
                  :user    "Bob"}
         :handler handler
         :error-handler error-handler})
```

In the example above, the `handler` will be invoked when the server responds with a success status. The response handler function should accept a single parameter. The parameter will contain the deserialized response from the server.

The library attempts to automatically discover the encoding based on the response headers, however the response format can be specified explicitly using the `:response-format` key.

The `error-handler` function is expected to to accept a single parameter that contains the error response. The function will receive the entire response map that contains the status and the description of the error along with any data returned by the server.

`:status` - contains the HTTP status code

`:status-text` - contains the textual description of the status

`:original-text` - contains the server response text

`:response` - contains the deserialized response when if deserialization was successful

When no handler function is supplied then no further action is taken after the request is sent to the server.

The request body will be interpreted using the ring-middleware-format library. The library will deserialize the request based on the `Content-Type` header and serialize the response using the `Accept` header that we set above.

The route should simply return a response map with the body set to the content of the response:

```
(ns myapp.routes.services
  (:require [compojure.core :refer [defroutes GET POST]]
            [ring.util.response :refer [response status]]))

(defn save-message! [{:keys [params]}]
  (println params)
  (response {:status :success}))

(defroutes services
  (POST "/send-message" request (save-message! request)))
```

Note that CSRF middleware is enabled by default. The middleware wraps the `home-routes` in the `handler` namespace of your application. It will intercept any request to the server that isn't a `HEAD` or `GET`.

```
(def app
  (-> (routes
        (wrap-routes home-routes middleware/wrap-csrf)
        base-routes)
      middleware/wrap-base))
```

The middleware is applied by `wrap-routes` after the routes are resolved and does not affect other route definitions. If we wish the services to be accessible to external client then we would update the routes to contain the `service-routes` as seen below.

```
(ns myapp.handler
  (:require ...
            [myapp.routes.services :refer [service-routes]]))

(def app
  (-> (routes
        service-routes ;; no CSRF protection
        (wrap-routes home-routes middleware/wrap-csrf)
        base-routes)
      middleware/wrap-base))
```

Alternatively, we could wrap the `service-routes` using `wrap-csrf` middleware as seen with `home-routes`:

```
(def app
  (-> (routes
        (wrap-routes service-routes middleware/wrap-csrf)
        (wrap-routes home-routes middleware/wrap-csrf)
        base-routes)
      middleware/wrap-base))
```

We would now need to pass the CSRF token along with the request. One way to do this is to pass the token in the `x-csrf-token` header in the request with the value of the token.

To do that we'll first need to set the token as a hidden field on the page:

```
<input id="csrf-token" type="hidden" value="{{csrf-token}}"></input>
```

Then we'll have to set the header in the request:

```
(POST "/send-message"
        {:headers {"Accept" "application/transit+json"
                   "x-csrf-token" (.-value (.getElementById js/documen
         :params {:message "Hello World"
                  :user    "Bob"}
         :handler handler
         :error-handler error-handler})
```