Contact Us

FEEDS

Atom
RSS

TOPICS

8th Light University
AWS
Android
Ansible
Apprentice Blog of the Week
Apprenticeship
Architecture
Business
Clojure
ClojureScript
Coding
Communications
Community
Consulting
Craftsmanship
Design
DevOps
Elixir
Front-end
Inspiration
Java
JavaScript
Learning
Microservices
Mobbing

TDD in ClojureScript

Eric Smith / 05 Oct 2016 Coding ClojureScript

I recently started a ClojureScript project, for fun not pay, which might surprise those who know me well. You see, when I had used ClojureScript in the past I found quite a few issues:

- The feedback loop was incredibly slow, because the compile times were slow.
- Compilation errors were often ignored, and the tests wouldn't fail, because "wee JavaScript!"
- DOM Manipulation was way behind what JQuery could already do.
- Etc., etc.

The truth is the problem wasn't the language but an ecosystem that was immature, and thanks to several improvements on the ClojureScript side—including incremental compilation and the development of Figwheel—I'm giving it another try. When it came to my testing setup, I had a few requirements:

- TDD *quickly* both in and out of the browser.
- Working sourcemaps and live reloading.
- Keep all the tools in Clojure/ClojureScript.

Setting this up was surprisingly difficult, as it is not documented in one place anywhere. So if you don't want to spend hours (or days) searching Google, you'll want to follow

Pairing
Principles
Process
Quality
React
Ruby
Testing
Tools
UX Design
Web Development
iOS

along. When you're done you'll have a ClojureScript application that's enjoyable to work with.

Creating The App

We'll use Leiningen to create the app. Quite simple to get started:

```
$ lein new app clojurescript-tdd-application
```

This will create a Clojure application, not a ClojureScript one. We'll need to modify `project.clj` to make it a ClojureScript app. I'm going to try to make these modifications slowly and step-wise, because I found that other directions on the web frequently skipped steps. Hopefully I won't make the same mistake. Your initial `project.clj` should look like this:

```
(defproject clojurescript-tdd-application "0.1.0-SNAPSH
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.
  :dependencies [[org.clojure/clojure "1.8.0"]]
  :main ^:skip-aot clojurescript-tdd-application.core
  :target-path "target/%s"
  :profiles {:uberjar {:aot :all}})
```

You'll want to update the URL field to a GitHub repository or other website, and pick your own license. The first thing you want to change is the dependencies to add ClojureScript.

```
:dependencies [[org.clojure/clojure "1.8.0"]
               [org.clojure/clojurescript "1.9.227"]]
```

Do make sure to look up the latest versions of Clojure and ClojureScript. Now since this is a ClojureScript app and not a Clojure app, you can remove a couple of the fields. In particular, you can get rid of `:profiles`, `:main`, and `:target-path`. If you plan to make a Clojure/ClojureScript application then you may want to restore those, but that's outside the scope of this tutorial.

We're going to start by getting an app up and running using Figwheel. Figwheel is a tool that allows live reloading of your ClojureScript application while you develop it. It has several fantastic features, including a ClojureScript REPL, really big obvious messages if your code has errors, and it works with an unoptimized build so debugging is a lot easier. Let's bring the `lein-figwheel` plugin into our project. You'll need to add the new `:plugins` key:

```
:dependencies [[org.clojure/clojure "1.8.0"]
               [org.clojure/clojurescript "1.9.227"]]
:plugins [[lein-cljsbuild "1.1.4" :exclusions [[org.clo
          [lein-figwheel "0.5.6"]]
```

Note that I also had to bring in `lein-cljsbuild` as well. It's not technically required, but it makes it a lot easier to use Figwheel. In order to get an app started we'll create our first cljsbuild configuration:

```
  :plugins [[lein-cljsbuild "1.1.4" :exclusions [[org.c
            [lein-figwheel "0.5.6"]]
  :cljsbuild {
    :builds [{:id "dev"              ; development confi
              :source-paths ["src"] ; Paths to monitor
              :figwheel true        ; Enable Figwheel
              :compiler {:main clojurescript_tdd_applic
                         :asset-path "cljs/out"
                         :output-to "resources/public/c
                         :output-dir "resources/public/
```

```
                                :source-map-timestamp true}
                 }]})
```

Keyword order doesn't matter but the `:cljsbuild` key needs to be in the main project hashmap, just like `:plugins`. After adding this configuration, you now have enough to run a ClojureScript application using Figwheel. Sort of. At the command line, run:

```
$ lein figwheel dev
```

If you do, you'll see something like:

```
Figwheel: Cutting some fruit, just a sec ...
...
Compiling "resources/public/cljs/main.js" from ["src"].
Failed to compile "resources/public/cljs/main.js" in 1
----  Could not Analyze    ----

  No such namespace: clojurescript_tdd_application.core

----  Analysis Error  ----
Figwheel: inital compile failed - outputing temporary h
```

That shouldn't be surprising if you think about it—you haven't created a namespace by that name in that directory. If you look in the directory `src/closurescript_tdd_application`, you'll see there is a `core.clj` file. Since you're running a ClojureScript app, not a Clojure one, we'll need to change the extension to cljs. In addition you'll want to modify it to fit a ClojureScript app:

```
; core.cljs

(ns clojurescript-tdd-application.core)
```

```
(defn main []
  (enable-console-print!)
  (prn "Hello, World!"))

(main)
```

It's idiomatic to create a function and call that as the last line of your main ClojureScript namespace, although technically anything in the main namespace will be executed. Now let's see what happens if I start the app:

```
cljs.user=> :cljs/quit
Choose focus build for CLJS REPL (devcards-test, dev, t
$ lein figwheel dev
Figwheel: Cutting some fruit, just a sec ...
...
Figwheel: Starting server at http://0.0.0.0:3449
...
Launching ClojureScript REPL for build: dev
...
Prompt will show when Figwheel connects to your applica
```

I've removed some of the other messages for clarity, but no matter how long you wait here you won't have a prompt. You won't get one if you browse to `http://localhost:3449`, and that's because nothing is executing your built ClojureScript file. Remember this in your `project.clj`?:

```
:compiler {:main clojurescript_tdd_application.core
           :asset-path "cljs/out"
           :output-to "resources/public/cljs/main.js"
           :output-dir "resources/public/cljs/out"
           :source-map-timestamp true}
```

Well, there's no file executing `resources/public/cljs/main.js`. You need an `index.html`

just like any other web app. Figwheel should have created a `resources/public` directory for you; it's where your compiled ClojureScript files are being placed. In `resources/public` create an `index.html` file. It can look like this:

```
<!DOCTYPE html>
<html>
  <head></head>
  <body>
    <script src="cljs/main.js" type="text/javascript"><
  </body>
</html>
```

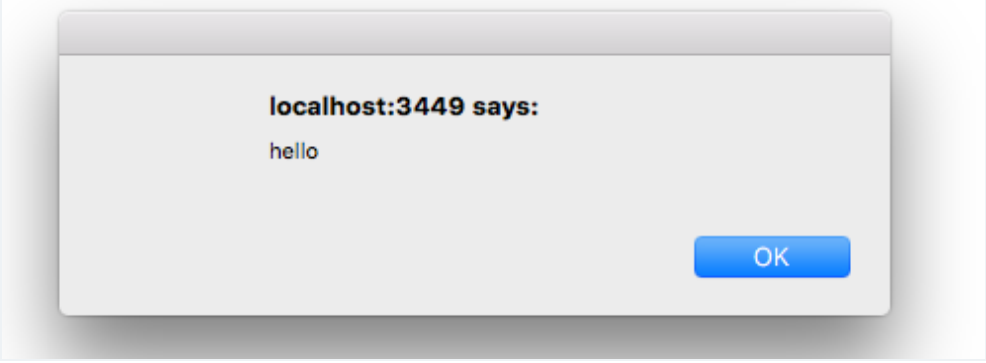
Eventually you're probably going to want to put something on the page, but this is the bare minimum you need to have a page up and running. Now if you browse to `http://localhost:3449/`, the "prompt" will show back in your command line. You did not have to restart Figwheel for this to work, so if you already stopped it, start it again. If your web browser is pointed at `http://localhost:3449`, then the prompt should load like so:

```
To quit, type: :cljs/quit
cljs.user=>
```

To really prove that Figwheel is attached to your application, you can type the following in the REPL:

```
To quit, type: :cljs/quit
cljs.user=> (js/alert "hello")
```

You should see this alert pop up in the browser:



Take a moment to think about how awesome that is (a REPL connected to your web browser!), then congratulate yourself. You have Figwheel running in development! Before we move on to testing, let's clear up a few things. You might remember I said that your files will be compiled to `resources/public/cljs/out`. Make sure you add that directory to your source control's ignore list so you don't accidentally check them in, then add `:clean-targets` to your project.clj:

```
:clean-targets ^{:protect false} [:target-path "out" "res
:cljsbuild {
  ;...
```

Once again the keyword order doesn't matter, but make sure `:clean-targets` is in the main project hashmap, not in any of the maps inside those maps. Now you can clean up the build with the `lein clean` command.

```
$ lein clean
$ ls resources/public/
index.html
```

Finally, once you build a real app you're probably going to want to reload CSS as well, which is easily done inside the `project.clj` with Figwheel options. Note the Figwheel key.

```
:clean-targets ^{:protect false} [:target-path "resourc
:figwheel { :css-dirs ["resources/public/css"] }
:cljsbuild {
```

You can also reload HTML on the fly, but that's more complicated and I don't actually do that yet, I just hit refresh. You can see how to do that on the Figwheel wiki.

Unit Testing - Command Line

Remember when I said I was gonna do TDD? Now that we've got an empty app setup, it's time to start writing tests for it. We'll start by setting up unit testing at the command-line so that we can have it working in CI quickly. This means you're going to need to install PhantomJS if you haven't already. If you're unaware, PhantomJS is a headless WebKit browser that's great for running unit tests but difficult to set up. Fortunately there is a popular test runner called `doo` that does most of the work for us.

After you've installed PhantomJS you can add doo to your dependency list and the doo plugin:

```
:dependencies [[lein-doo "0.1.7"]
               [org.clojure/clojure "1.8.0"]
               [org.clojure/clojurescript "1.9.227"]]
:plugins [[lein-cljsbuild "1.1.4" :exclusions [[org.clo
          [lein-doo "0.1.7"]
          [lein-figwheel "0.5.6"]]
```

The plugin won't do much on its own. You'll need to create another build configuration. See the configuration with the build id "test". I've left the `:dev` configuration in the example below for context, but it's unchanged:

```
:cljsbuild {
  :builds [{:id "dev"              ; development configu
            :source-paths ["src"] ; Paths to monitor fo
            :figwheel true        ; Enable Figwheel
            :compiler {:main clojurescript_tdd_applicat
                       :asset-path "cljs/out"
                       :output-to "resources/public/clj
                       :output-dir "resources/public/cl
                       :source-map-timestamp true}
            }
           {:id "test"
            :source-paths ["src" "test"]
            :compiler {:main runners.doo
                       :optimizations :none
                       :output-to "resources/public/clj
```

Having added the build configuration, you can try to run the unit tests at the command line using the `lein doo` command:

```
$ lein doo phantom test once

;; ==============================================================
;; Testing with PhantomJS:

goog.require could not find: runners.doo

  phantomjs://code/phantom6698296083052361156.js:81 in
```

That's to be expected as you haven't created a namespace named `runners.doo`. Go ahead and create the `test/runners` directory and a file there named `doo.cljs`. Inside that file you should have:

```
(ns runners.doo
  (:require [doo.runner :refer-macros [doo-all-tests]]
                                      [runners.tests]))

(doo-all-tests #"(clojurescript-tdd-application)\..*-te
```

Note the line `(doo-all-tests #"(clojurescript-tdd-application)\..*-test")` should match all namespaces beginning with `clojurescript-tdd-application` and ending with `-test`. Should you need more than one top-level namespace then you can solve that with a `|` in the regular expression. Running this now will cause:

```
$ lein doo phantom test once
clojure.lang.ExceptionInfo: failed compiling file:test/
...
Caused by: clojure.lang.ExceptionInfo: No such namespac
```

Yup, there's no `runners.tests.cljs` file, because we haven't created it. Go ahead and create a file named `tests.cljs` in runners that looks like this:

```
(ns runners.tests)
```

Why do we have this seemingly pointless namespace? `doo-all-tests` can only find namespaces that have been loaded, and the tests won't be unless you require them. As we add tests to the system we'll need to add them to this namespace. For example, my current side project looks like this:

```
(ns runners.tests
  (:require [space-invaders.game-test]
            [space-invaders.view-test]
            [util.game-loop-test]
            [util.image-loader-test]))
```

Now when you run the tests you should see this:

```
$ lein doo phantom test once

;; ======================================================
;; Testing with PhantomJS:


Testing runners.doo

Ran 0 tests containing 0 assertions.
0 failures, 0 errors.
```

It's glorious, I know. When you're done telling everybody around you about this, you can also try out all your tests in an auto-runner by calling this:

```
$ lein doo phantom test auto
Building ...
... done. Elapsed 0.739818844 seconds

;; ======================================================
;; Testing with Phantom:


Testing runners.doo

Ran 0 tests containing 0 assertions.
0 failures, 0 errors.
Watching paths: /Users/eric/Projects/clojurescript-tdd-
```

This can work, but PhantomJS is a little slow—not to run the tests, but to load and get started. Furthermore, debugging a test in PhantomJS is a huge pain in the butt. I prefer to run my tests in the browser, and we'll get that set up in a moment. First, though, let's get rid of that overly verbose test command. `lein cljsbuild test` should run all your tests provided you configure it in the cljsbuild configuration. Add this to your project.clj:

```
:cljsbuild {
  :test-commands {"test" ["lein" "doo" "phantom" "test"
```

You can of course add this anywhere in the `:cljsbuild` hash, but it's easiest to read at the very top. Now instead of the cumbersome command, just run `lein cljsbuild test` and tests should run nicely.

But about the browser...

Browser Testing

It's going to be pretty hard for me to write about testing ClojureScript in the browser without ranting for a little bit. If you've read the `doo` README then you know that `doo` uses Karma for testing in the browser. Sorry, but if I wanted to type `npm install [anything]` I'd go ahead and write plain old JavaScript. What does Karma give you? It runs tests automatically in a browser, rerunning them on changes. In other words, it does exactly what Figwheel already does, only with an awkward interface and a dependency on npm for your ClojureScript project. No thank you.

After watching several videos and trying out several options to avoid Karma, this state of affairs led to me writing my own test runner. I got as far as running several tests using Figwheel as my runner.

```
8 ran successfully, 0 failed.
18 assertions passed, 0 assertions failed, and there were 0 errors

test-update-game
 "increments ticks"
test-initial-app-state
 "each row of invaders is 11 long"
test-toggle-back-and-forth
 "the invaders toggle between two states based on the velocity"
test-invaders-to-position
 "no invaders, without any tics"
 "one invader, without any tics"
 "an invader on the second row, without any tics"
 "two invaders in the same row, without any tics"
```

It turns out that I wasn't the only person who realized Figwheel did exactly what I wanted—so did Bruce Hauman, the creator of Figwheel. He created yet another awesome ClojureScript tool—Devcards. Devcards will do a lot more than run tests, but it has that built in and it's what we'll use for running tests in the browser. It's time for more changes to project.clj. First, let's add Devcards to our dependencies list (that's dependencies, not plugins):

```
:dependencies [[lein-doo "0.1.7"]
               [devcards "0.2.1-7"]
               [org.clojure/clojure "1.8.0"]
               [org.clojure/clojurescript "1.9.227"]]
```

As always, make sure you double check that the dependency version is the latest one. You'll need to set up another cljsbuild profile that runs Devcards.

```
{:id "devcards-test"
 :source-paths ["src" "test"]
 :figwheel {:devcards true}
 :compiler {:main runners.browser
            :optimizations :none
            :asset-path "cljs/tests/out"
            :output-dir "resources/public/cljs/tests/ou
```

```
            :output-to "resources/public/cljs/tests/all
            :source-map-timestamp true}}
```

You might wonder why we can't reuse the existing test config. The reason is that the test runners are different. Much like before, we'll run tasks at the command line after we make changes, so we can verify how things work so far. To run the Devcards configuration, the command is `lein figwheel devcards-test`:

```
$ lein figwheel devcards-test
...
Failed to compile "resources/public/cljs/tests/all-test
----  Could not Analyze    ----

  No such namespace: runners.browser, could not locate
```

You probably saw that coming. In `runners/`, create a file `browser.cljs` that looks like this:

```
(ns runners.browser
  (:require [runners.tests]))
```

That will be enough Clojure to get tests showing up in the browser, but just like with the dev configuration you'll need an HTML page to run that code. Create a file called `tests.html` in `resources/public` that looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Tests</title>
        <meta name="viewport" content="width=device-wid
        <meta charset="UTF-8">
  </head>
```

```
  <body>
    <script src="cljs/tests/all-tests.js" type="text/ja
  </body>
</html>
```

Now you can start the app in the Devcards configuration:

```
$ lein figwheel devcards-test
```

To go to your tests, browse to `http://localhost:3449/tests.html` and you should see this:

There are no tests yet. Let's write a couple tests and see how well this works.

Workflow

Go ahead and quit Figwheel, then start it again by running:

```
$ lein figwheel devcards-test dev
```

Running both configurations at once means we can see both the tests and the application at the same time, and both will benefit from Figwheel's auto-reloading. Note that you'll start "connected" to the dev environment, I believe because dev is first in the cljsbuild config. When Figwheel is loaded you'll see text like this:

```
Launching ClojureScript REPL for build: dev
Figwheel Controls:
          (stop-autobuild)                ;; stops Figw
          (start-autobuild [id ...])      ;; starts aut
          (switch-to-build id ...)        ;; switches a
          (reset-autobuild)               ;; stops, cle
          (reload-config)                 ;; reloads bu
          (build-once [id ...])           ;; builds sou
          (clean-builds [id ..])          ;; deletes cc
          (print-config [id ...])         ;; prints out
          (fig-status)                    ;; displays c
  Switch REPL build focus:
          :cljs/quit                      ;; allows you
    Docs: (doc function-name-here)
    Exit: Control+C or :cljs/quit
 Results: Stored in vars *1, *2, *3, *e holds last exce
Prompt will show when Figwheel connects to your applica
To quit, type: :cljs/quit
```

Note that line at the top that says it's launching the REPL for *dev*. If you want to launch your REPL for devcards-test you can do that by using the `:cljs/quit` command, which will prompt you to switch if needed. I usually leave this as-is. At this point open one browser tab and point it at `http://localhost:3449`, and a second tab and point it at `http://localhost:3449/tests.html`. If you've got the screen space, you may want to use two separate browser windows so that both the application and tests are auto-reloading.

Let's create our first test. In `test/clojurescript_tdd_application` there is a file named `core_test.clj`. This is just a leftover from the original `lein new` command. Rename it to `core_test.cljs` and add it to `runner.tests`.

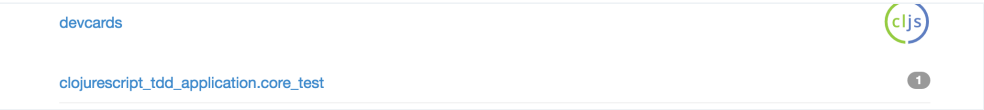
```
(ns runners.tests
  (:require [clojurescript-tdd-application.core-test]))
```

You'll start to see errors on the screen in the browser and the ClojureScript REPL, because this code is Clojure not ClojureScript. Let's write our first ClojureScript test.

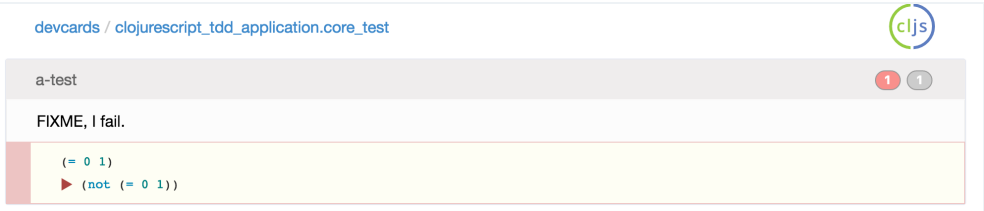
```
(ns clojurescript-tdd-application.core-test
  (:require [cljs.test :refer-macros [is testing]]
            [devcards.core :refer-macros [deftest]]))

(deftest a-test
  (testing "FIXME, I fail."
    (is (= 0 1))))
```

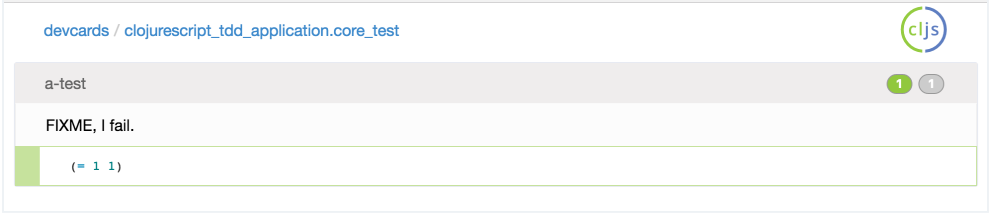
The minute you add this code and save it, you should see a link appear in Devcards, and the errors should go away.



Note the namespaces used in the test above. In order for a test to show up in Devcards, it needs to use the `devcards.core` version of `deftest`, not the `cljs.test` version. Fortunately, the `devcards.core` version will run the `cljs.test` when run at the command line, so the test will run in both environments. Clicking the link to `clojurescript.tdd-application.core-test` will run all the tests in that namespace, so let's do that.



This gives you a nice, clear reading of what's failing and why it fails. Make the test pass and *without refreshing* you should see this.



Pretty awesome. Just in case you don't trust me, go ahead and run your tests at the command line.

```
$ lein cljsbuild test
Compiling ClojureScript...
Running ClojureScript test: test

;; ================================================
;; Testing with PhantomJS:


Testing clojurescript-tdd-application.core-test

Ran 1 tests containing 1 assertions.
0 failures, 0 errors.
```

There you can see the one test running in this namespace. Notice how much slower it is than the browser. It's this browser-based workflow that keeps me using Devcards and by extension `cljs.test`, as there isn't a Devcards runner for speclj yet.

Browser-based applications tend to be asynchronous, so we'll need a way to write asynchronous tests. The combination of `cljs.test` asynchronous facilities and the Devcards UI really shine here. To make this test asynchronous you'll need to use the `cljs.test` `async` macro like so:

```
(ns clojurescript-tdd-application.core-test
  (:require [cljs.test :refer-macros [is testing async]
            [devcards.core :refer-macros [deftest]]))
```

```
(deftest a-test
  (testing "FIXME, I fail."
    (async done
      (js/setTimeout
        (fn []
          (is (= 1 0))
          (done))
        100))))
```

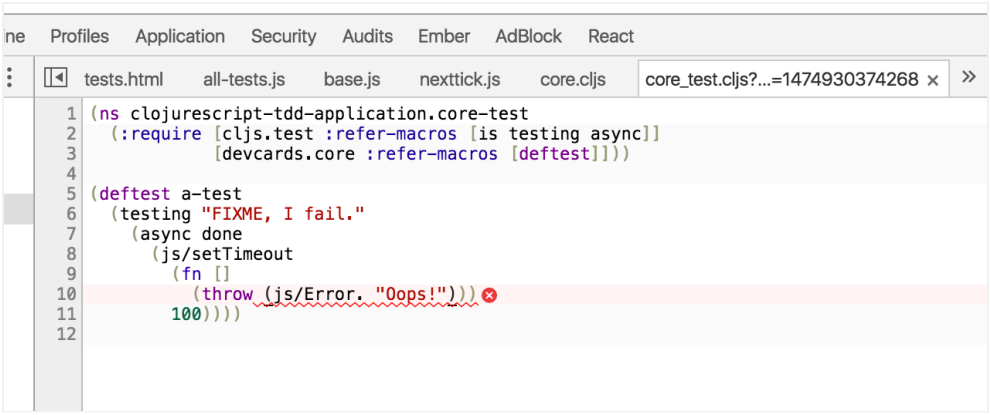
Don't forget to add `async` to the list of macros when you require `cljs.test` (see line 2). What that test does is check if `(= 1 0)` after a 100 second timeout. The `async` macro yields a function (called by convention) that, when called, signals that the test is... done. Remove that call and what does the test do?

You might ask, what's the big deal about a message saying that the test timed out? Well by default `cljs.test` doesn't do that. It just hangs, and in a browser that can leave you with the impression the tests didn't run. Note the error message asking about exceptions. Let's take a look at what happens if an exception was thrown.

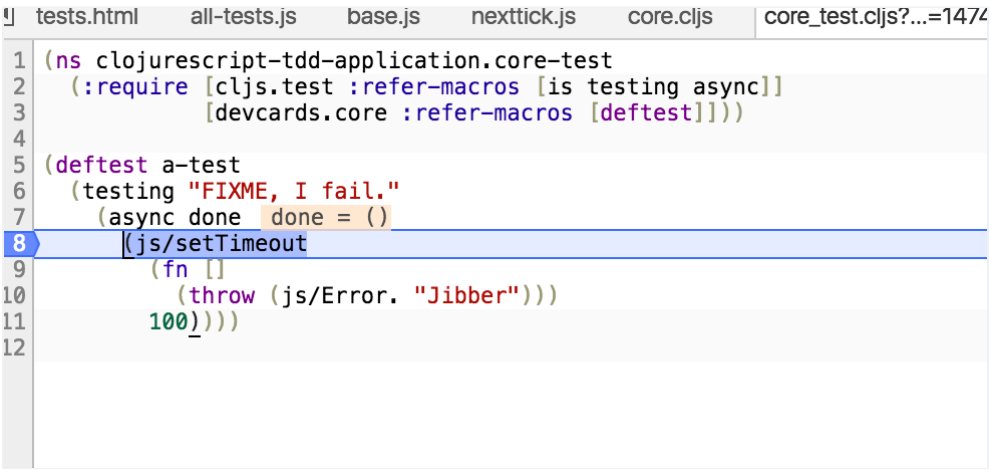
```
(deftest a-test
  (testing "FIXME, I fail."
    (async done
      (js/setTimeout
        (fn []
          (throw (js/Error. "Oops!")))
        100))))
```

```
("../clojurescript_tdd_application/core_test.js" "../runners/tests.js" "../ru
Figwheel: NOT loading these files
not required: (nil)
"Running tests!!"
▼Uncaught Error: Oops!
    (anonymous function) @ core_test.cljs?rel=1474930374268:10
```

There's the stack trace, and when I click the file location:



Well I'll be darned—that's ClojureScript code. In the browser! And yes, you can set breakpoints in it.



If you don't see that, then sourcemaps are probably not enabled in your browser, which you can enable in virtually any browser. I currently have `:source-map-timestamp` set to true so that the maps won't be cached to old versions of cljs files. But that means you lose breakpoints when a file changes. You can change that setting to `:source-map`, but browser caching is likely to hold the wrong sourcemap without a full refresh.

There you have it! Follow the "simple" directions above and you're up and running with a testable ClojureScript development environment. Okay, it took a while, but I swear it's worth it. If you want to see the full source for the skeleton project we just developed, it's on github. If you want to watch an application being developed this way, you can look at my hobby project, Space Invaders in ClojureScript.

Gotchas

The setup isn't quite perfect. I generally find I need to refresh a DevCards page after exceptions, because the automatic refreshing stops, and when you modify macros that appear in .cljc files. You should really read the Figwheel Quick Start paying particular attention to "writing reloadable code", which will be necessary in your application and your tests. In addition asynchronous tests lose their pretty `testing` descriptions which isn't a big deal unless you have several in a row. In that case it becomes hard to tell what test is failing, and there is a github issue for it. Devcards also don't support `cljs.test` fixtures yet, or running all tests but I don't use those features often anyway.

While imperfect, this is still far far better than the situation a few years ago. With transpilers and frameworks I find that "pure" JavaScript can actually have a much slower feedback loop, without a Clojure experience.

References

None of the above is my original work; my hope is that by putting step-by-step directions other developers won't have to spend as much time as I did with README's, Google searches, and swearing.

- Figwheel and Devcards are both the creations of Bruce Hauman, who has done more for the ClojureScript

community than anybody not directly involved with creating the language. You should really check out his talks on both of these tools, as I have not demonstrated all of their functionality.

- Rafik Naccache wrote an excellent article on getting set up with doo for running command line tests. Like most ClojureScript developers, they run all their tests in console. As you can see, I prefer using Devcards.
- doo is the library we are using for PhantomJS testing.
- cljs.test is the library we're using for writing tests.

Eric Smith has a Master's Degree in Video Game Development from DePaul University.

Follow @paytonrules

RELATED POSTS

Advice for early-career developers Colin Jones
Legacy Code: Spying On Global Functions Christoph Gockel
Framework Seams Mike Knepper
Myths about Unit Tests Fabien Townsend
AWS Do's and Don'ts Sarah Sunday
How to Grow a User Group, the Remix Ray Hightower
New Leadership for ChicagoRuby Ray Hightower
How JSON decoding works in Elm—Part 3 Kofi Gumbs
Refactoring React Josh Mccormick
Diversity, Inclusion, and 8th Light Paul Pagel

MORE POSTS BY THIS AUTHOR

Test-Driving the Game Loop pt. 2
Test-Driving the Game Loop pt. 1
Stand-ups are Broken, but Should They be Fixed?
Yes, you can speak at a conference
What is a Bug?

We are Principled: 6th Edition
Mind Your Own Business Rails
That's Not Yours
Pointers, Schmointers

Interested in 8th Light's services? Let's talk.

Contact Us

Contact
Privacy Policy
Apprenticeship
Blog
Twitter
Sitemap

Chicago
London
Los Angeles
New York