all async all the time

@mccraigmccraig

what

- i decided to implement my latest system in fully asynchronous fashion (almost - local file access isn't async yet)
- i played around with some different ways of structuring the system, and i'm going to explore these here
- i'm not going to try and teach anyone how to grok monads, but there will be examples which include some monads
- but don't be afraid, they are quite friendly

why async ?

- optimise thread/memory use
- underlying i/o mechanism is async
- increased throughput
- easier throttling / control / backpressure when needed
- i wanted to dammit

how to do it

- callbacks / CPS
- CSP / core.async
- promises
- various promise-like monads

some code

- we will model an api which gathers results from other upstream apis
- all code examples are working
- https://github.com/mccraigmccraig/all-async

platform

- all the examples are implemented on juxt/yada
- yada is based on ztellmnan/aleph & ztellman/manifold
- manifold supports multiple async paradigms, so is a good choice for exploring the differences
- funcool/cats is used for monads / applicatives etc

building an api which consumes upstream services

- some simple apis which consume the upstream services

- looking in particular at comprehensibility, composition, error-handling

upstream services

- /api/[un]reliable-random-number
- /api/[un]reliable-random-letter
- /api/slow-random-number

an upstream service

```
(defn random-number-handler
  [ctx]
  (let [r (rand-int 100)]
    (info "random-number" r)
    (generate-string r)))

(defn reliable-random-number-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response random-number-handler}}})))
```

an unreliable upstream service

```
(defn unreliable-handler
  [handler]
  (fn [ctx]
    (info "unreliable")
    (if (> (rand) 0.25)
      (handler ctx)
      (d/error-deferred
       (ex-info "i'm a teapot"
                {:status 418
                 :yada.core/http-response true})))))

(defn random-number-handler
  [ctx]
  (let [r (rand-int 100)]
    (info "random-number" r)
    (generate-string r)))

(defn unreliable-random-number-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (unreliable-handler
                                random-number-handler)}}})))
```

callbacks

callbacks - setup

```
(defn callback-handler
  [handler]
  (fn [ctx]
    (let [r (d/deferred)
          on-success (fn [v] (d/success! r (generate-string v)))
          on-error (fn [e] (d/error! r e))]
      (handler on-success on-error)
      r)))

(defn http-get-with-callbacks
  [url on-success on-error]
  (let [r (http/get url)]
    (d/on-realized r
                   #(on-success (-> % :body slurp parse-string))
                   on-error)
    nil))
```

a single callback

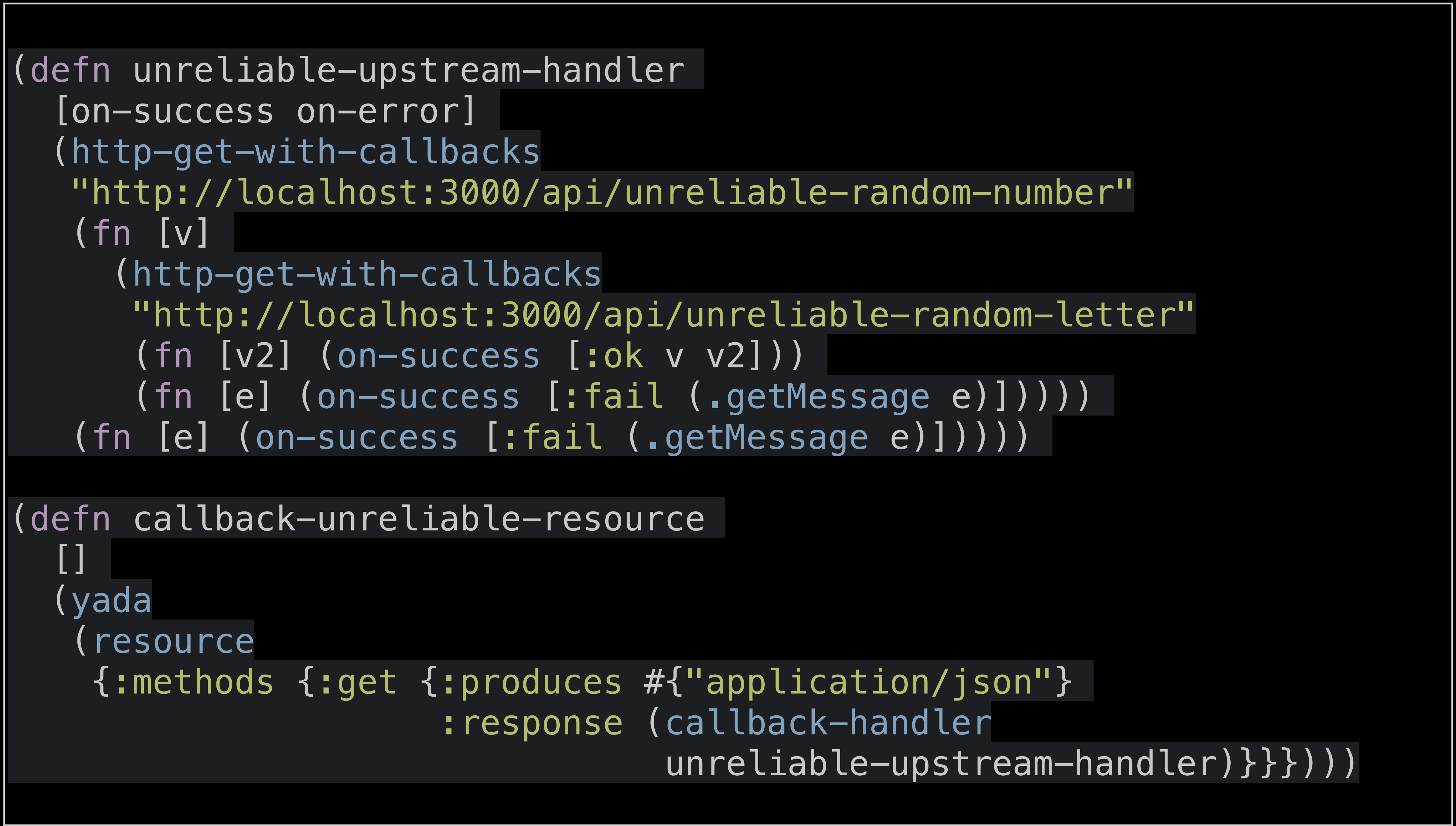
```
(defn reliable-upstream-handler
  [on-success on-error]
  (http-get-with-callbacks
   "http://localhost:3000/api/reliable-random-number"
   (fn [v]
     (on-success [:ok v]))
   (fn [e]
     (on-success [:fail (.getMessage e)]))))

(defn callback-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (callback-handler
                                reliable-upstream-handler)}}})))
```

callbacks

- well, they are simple
- the intent is quite well hidden amongst the boilerplate

callbacks - composition & error handling



```
(defn unreliable-upstream-handler
  [on-success on-error]
  (http-get-with-callbacks
   "http://localhost:3000/api/unreliable-random-number"
   (fn [v]
     (http-get-with-callbacks
      "http://localhost:3000/api/unreliable-random-letter"
      (fn [v2] (on-success [:ok v v2]))
      (fn [e] (on-success [:fail (.getMessage e)]))))
   (fn [e] (on-success [:fail (.getMessage e)]))))

(defn callback-unreliable-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (callback-handler
                                unreliable-upstream-handler)}}})))
```

callbacks - composition

- they don't compose easily
- stages in the computation have to be concerned with results of other stages
- gets very messy very quickly - great discipline required
- move along

promises

promises

- promises offer a nicer way of dealing with callbacks
- manifold calls them "Deferred" values
- the idea is to capture the state of the computation at a given stage, and register callbacks against that
- this example is semantically different from the callback example, demonstrating both coordination and chaining with manifold

promises-setup

```
(defn http-get-promise
  [url]
  (let [dr (http/get url)]
    (d/chain dr
             (fn [v]
               (-> v :body slurp parse-string)))))

(defn encode-error-handler
  [handler]
  (fn [ctx]
    (let [dv (handler ctx)]
      (-> dv
          (d/chain (fn [v] [:ok v]))
          (d/catch Exception (fn [e] [:fail (.getMessage e)]))))))
```

promises

```
(defn promise-handler
  [ctx]
  (let [r1 (http-get-promise "http://localhost:3000/api/reliable-random-number")
        r2 (http-get-promise "http://localhost:3000/api/reliable-random-letter")
        comb (d/zip r1 r2)]
    (d/chain comb
             (fn [[v1 v2]]
               [v1 v2]))))
```

```
(defn promise-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response promise-handler}}})))
```

promises - error handling

```
(defn promise-unreliable-handler
  [ctx]
  (let [r1 (http-get-promise "http://localhost:3000/api/unreliable-random-number")
        r2 (http-get-promise "http://localhost:3000/api/unreliable-random-letter")
        comb (d/zip r1 r2)]
    (d/chain comb
             (fn [[v1 v2]]
               [v1 v2]))))
```

```
(defn promise-unreliable-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (encode-error-handler
                                promise-unreliable-handler)}}})))
```

promises

- much cleaner than callbacks
- have built-in error handling, which is nice, and helps composition
- every promise library has it's own way of combining results from promises
- manifold lets us do better still

promises - flow

```
(defn promise-unreliable-flow-handler
  [ctx]
  (d/let-flow [v1 (http-get-promise "http://localhost:3000/api/unreliable-random-number")
               v2 (http-get-promise "http://localhost:3000/api/unreliable-random-letter")
    [v1 v2]))

(defn promise-unreliable-flow-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (encode-error-handler
                                promise-unreliable-flow-handler)}}})))
```

promises - flow

- this is starting to look really nice
- the async code doesn't look much different from similar sync code
- if you use raw promises they are going to infect your codebase

CSP / core.async

core.async setup

```
(defn async-handler
  [handler]
  (fn [ctx]
    (let [d (d/deferred)]
      (go
        (let [v (<! (handler ctx))]
          (if-not (instance?  Throwable v)
            (d/success! d v)
            (d/error! v))))
      d)))

(defn http-get-with-core-async
  [url]
  (let [dr (http/get url)
        ch (chan)]
    (d/on-realized dr
                   (fn [r]
                     (let [v (-> r :body slurp parse-string)]
                       (put! ch v)))
                   (fn [e]
                     (put! ch e)))
    ch))
```

core.async

```clojure
(defn reliable-upstream-handler
  [ctx]
  (go
    (let [rn (<! (http-get-with-core-async
                  "http://localhost:3000/api/reliable-random-number"))
          rl (<! (http-get-with-core-async
                  "http://localhost:3000/api/reliable-random-letter"))]
      [:ok [rn rl]])))

(defn core-async-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (async-handler reliable-upstream-handler)}}})))
```

core.async composition & error handling

```
(defn unreliable-upstream-handler
  [ctx]
  (go
    (let [rn (<! (http-get-with-core-async
                  "http://localhost:3000/api/unreliable-random-number"))
          rl (<! (http-get-with-core-async
                  "http://localhost:3000/api/unreliable-random-letter"))]
      (cond
        (not (or (instance? Exception rn)
                 (instance? Exception rl)))
        [:ok [rn rl]]

        :else
        [:fail (if (instance? Exception rn)
                 (.getMessage rn)
                 (.getMessage rl))]))))

(defn core-async-unreliable-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (async-handler unreliable-upstream-handler)}}})))
```

core.async

- go blocks are quite nice
- async code looks like sync code
- core.async doesn't help out with error handling
- it does do a tonne of cool stuff though
- but you are going to need a lot of discipline or to hide it somehow

assorted promise-like monads

what have monads ever done for me

- nothing
- except manage the machinery of the steps of a stepwise computation
- leaving the interesting, task-specific, part of the computation

do syntax / mlet

- the monad examples will look very clean because of the "mlet" or "do" syntax
- it's a macro which re-arranges the nested calls of monadic functions (which look pretty grungy) into a nice let-like list
- you can mostly forget about that once you get a feel for it

promise-monad setup

```
(defn http-get-promise-monad
  [url]
  (with-context deferred-context
    (mlet [r (http/get url)
           :let [v (-> r :body slurp parse-string)]]
      (return v))))

(defn encode-error-handler
  [handler]
  (fn [ctx]
    (let [dv (handler ctx)]
      (-> dv
          (d/chain (fn [v] [:ok v]))
          (d/catch Exception (fn [e] [:fail (.getMessage e)]))))))
```

promise monad

```
(defn monad-handler
  [ctx]
  (with-context deferred-context
    (mlet [r1 (http-get-promise-monad
               "http://localhost:3000/api/reliable-random-number")
           r2 (http-get-promise-monad
               "http://localhost:3000/api/reliable-random-letter")]
      (return [r1 r2]))))

(defn monad-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response monad-handler}}})))
```

promise monad

- this looks pretty good
- very comprehensible, if you can forget about the m in mlet for a moment
- flexible wrt the datatype (cats has manifold deferred, core.async chan, promesa, promissum)
- composes straightforwardly
- inherits whatever error-handling the underlying structure has

promise monad - error handling

```clojure
(defn monad-unreliable-handler
  [ctx]
  (with-context deferred-context
    (mlet [r1 (http-get-promise-monad
               "http://localhost:3000/api/unreliable-random-number")
           r2 (http-get-promise-monad
               "http://localhost:3000/api/unreliable-random-letter")]
      (return [r1 r2]))))

(defn monad-unreliable-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (encode-error-handler
                                monad-unreliable-handler)}}})))
```

benefits

- removes step-machinery-related code from your codebase
- easy to change the step machinery
- easier comprehension because you see only the problem-related code

so far, so much like let-flow

- yes, but
- let-flow is a manifold construct - other promise implementations do it differently, so the promise-monad frees your code from infective structures
- and infects your code with monadic calls
- but you are quite free to change monadic types - i ported a 10kloc ClojureScript application based on one-shot channels to promises in ~2hrs
- cats' implementation works fine on ClojureScript
- and you can do more...

but wait, i wanna do more than just one thing at a time

- this computation isn't a list of steps dammit, it's a graph
- monads cannot help you now
- applicative functors to the rescue

e.g. slow calls

```clojure
(defn timer-handler
  [handler]
  (fn [ctx]
    (let [st (t/now)
          r (handler ctx)]
      (-> r
          (d/chain (fn [v]
                     (let [et (t/now)
                           d (t/in-millis (t/interval st et))]
                       [:ok v [d :millis]]))))))))

(defn monad-slow-handler
  [ctx]
  (with-context deferred-context
    (mlet [r1 (http-get-promise-monad
               "http://localhost:3000/api/slow-random-number")
           r2 (http-get-promise-monad
               "http://localhost:3000/api/slow-random-number")]
      (return [r1 r2]))))

(defn monad-slow-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (timer-handler
                                monad-slow-handler)}}}))))
```

with applicatives instead

```
(defn applicative-slow-handler
  [ctx]
  (with-context deferred-context
    (alet [r1 (http-get-promise-monad
               "http://localhost:3000/api/slow-random-number")
           r2 (http-get-promise-monad
               "http://localhost:3000/api/slow-random-number")]
      [r1 r2])))

(defn applicative-slow-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (timer-handler
                                applicative-slow-handler)}}})))
```

alet

- another macro, does a different transformation to mlet
- includes an analysis of which steps of a computation depend on other steps
- issues optimal batches of "parallel" calls to satisfy the computation in the minimum number of steps
- note there is no "return"
- don't look at the macro-expansion

side-channels

- i want to collect some meta-information about a computation. timings, validation info etc

- maybe from deeply nested calls

- without every single call in the stack managing meta-info parameters and return values

monad-transformer

```
(def deferred-writer-context (writer/writer-t deferred-context))
```

writer setup

```
(defn http-get-log-promise
  [url]
  (with-context deferred-writer-context
    (mlet [:let [st (t/now)]

           r (lift (http/get url))

           :let [et (t/now)
                 d (t/in-millis (t/interval st et))

                 v (-> r :body slurp parse-string)]

           _ (writer/tell {:url url :duration d})]
      (return v))))

(defn encode-error-log-handler
  [handler]
  (fn [ctx]
    (let [dv (handler ctx)]
      (-> dv
          (d/chain (fn [[v log]]
                     (let [d (->> log
                                  (map :duration)
                                  (filter identity)
                                  (reduce +))]
                       (generate-string
                        [:ok v d log]))))
          (d/catch Exception
              (fn [e] [:fail (.getMessage e)])))))))
```

side-channel handlers

```clojure
(defn monad-unreliable-log-handler
  [ctx]
  (with-context deferred-writer-context
    (mlet [r1 (http-get-log-promise
               "http://localhost:3000/api/unreliable-random-number")
           r2 (http-get-log-promise
               "http://localhost:3000/api/unreliable-random-letter")]
      (return [r1 r2]))))

(defn monad-unreliable-log-resource
  []
  (yada
   (resource
    {:methods {:get {:produces #{"application/json"}
                     :response (encode-error-log-handler
                                monad-unreliable-log-handler)}}})))
```

side-channels

- the side-channel handlers look the same as the vanilla promise-monad handlers

- you can "tell" the side-channel anything from anywhere, however deeply nested, and it gets added to the log

- only the context changed, and the beginning and end of any processing stack

some bad points

- promise-monads are definitely invasive - but then so are core.async, plain promises etc
- no help from static type-checking - weird errors if you forget to wrap, perhaps by letting a branch return nil. but hey, no weirder than the errors you get from chains of promises, or callbacks, or unintended lazy side-effects

summary

- async hasn't been any (well, much) harder to implement than sync
- there is less choice, but there are async clj/cljs clients around for dbs, qs etc
- cool stuff like fully async multipart upload handling can be yours
- monads/applicatives remove infective goop from your codebase (and replace it with nicer infective goop)
- and make things a bit more comprehensible
- and a bit easier to change
- and can give you a very clean solution for side-channels
- and applicatives can straightforwardly solve some nasty asynchronous graph dependency problems