9

Reactive Programming

One of the many interesting applications of programming with asynchronous tasks is reactive programming. This methodology of programming is all about asynchronously reacting to changes in state. In reactive programming, code is structured in a way that it reacts to changes. Generally, this is implemented by using asynchronous data streams, in which data and events are propagated asynchronously through a program. In fact, there are quite a few interesting variants of reactive programming. Event streams are quite similar to channels, except for the fact that channels are not implicitly asynchronous.

Reactive programming is particularly useful in designing graphical user interfaces, where changes in internal state of an application must asynchronously trickle down to the user interface. A program is thus segregated into events and logic that is executed on those events. For programmers used to imperative and object-oriented programming techniques, the hardest part of reactive programming is thinking in reactive abstractions and letting go of old habits like mutating state. However, if you’ve been paying attention so far and have started thinking with immutability and functions, you’ll find reactive programming quite natural.

In this chapter, we will explore a few interesting forms of reactive programming through Clojure and ClojureScript libraries. We will also demonstrate how we can build user interfaces using reactive programming towards the end of the chapter.

# Reactive Programming with Fibers and Dataflow Variables

Dataflow programming is one of the simplest forms of reactive programming. In dataflow programming, computations are described by composing variables without bothering about when these variables are set to a value. Such variables are also called dataflow variables, and they will trigger computations that refer to them once they are set. The Pulsar library (https://github.com/puniverse/pulsar) provides a couple useful constructs for dataflow programming. These constructs can also be used with Pulsar fibers, which we briefly talked about in Chapter 8. In this section, we will explore the basics of fibers and dataflow variables from the Pulsar library.

The following library dependencies are required for the upcoming examples.

[co.paralleluniverse/quasar-core "0.7.3"]

[co.paralleluniverse/pulsar "0.7.3"]

Your project.clj file must also contain the following entries.

:java-agents [[co.paralleluniverse/quasar-core "0.7.3"]]

:jvm-opts ["-Dco.paralleluniverse.pulsar.instrument.auto=all"]

Also, the following namespaces must be included in your namespace declaration.

(ns my-namespace

(:require [co.paralleluniverse.pulsar.core :as pc]

[co.paralleluniverse.pulsar.dataflow :as pd]))

The elementary abstraction of an asynchronous task in Pulsar library is a fiber. Fibers are scheduled for execution on fork-join based thread pools, and we can create a large number of fibers without regard to the number of available processors or cores. Fibers can be created using the spawn-fiber and fiber macros from the co.paralleluniverse.pulsar.core namespace. The spawn-fiber macro must be passed a function that takes no arguments, and the fiber form must be passed a body of expression. Both of these forms will create a fiber on which the function or body of expressions passed to them is executed. The join function from the co.paralleluniverse.pulsar.core namespace can be used to retrieve the value returned by a fiber.

An important rule we must keep in mind while dealing with fibers is that we must never call methods or functions that manipulate the current thread of execution from within a fiber. Instead, we must use fiber-specific functions from the co.paralleluniverse.pulsar.core namespace to perform these operations. For example, calling the java.lang.Thread/sleep method in a fiber must be avoided. Instead, the sleep function from the co.paralleluniverse.pulsar.core namespace can be used to suspend the current fiber for a given number of milliseconds.

The following examples can be found in src/m\_clj/c9/fibers.clj of the books source code.

For example, we can add two numbers using a fiber as shown in Example 9.1 below. Of course, using a fiber for such a trivial operation has no practical use, and it is only shown here to demonstrate how we can create a fiber and obtain the value returned by a fiber.

Example 9.1. Adding two numbers with a fiber

(defn add-with-fiber [a b]

(let [f (pc/spawn-fiber

(fn []

(pc/sleep 100)

(+ a b)))]

(pc/join f)))

The add-with-fiber function shown above creates a fiber f using the spawn-fiber macro and fetches the result returned by the fiber using the join function. The fiber f will suspend itself for 100 milliseconds using the sleep function and return the sum of a and b.

Let’s talk a bit about dataflow variables. We can create dataflow variables using the df-val and df-var functions from the co.paralleluniverse.pulsar.dataflow namespace. Both of these functions require no arguments. A dataflow variable created using these functions can be set by calling it like a function and passing it a value. Also, the value of a dataflow variable can be obtained by dereferencing it using the @ operator. A dataflow variable declared using the df-val function can only be set once, whereas one created using the df-var function can be set several times. The df-var function can also be passed a function that takes no arguments and refers to other dataflow variables in the current scope. This way, the value of such a dataflow variable will be recomputed when the values of referenced variables are changed. For example, two numbers can be added using dataflow variables as shown in the df-add function defined in Example 9.2 below.

Example 9.2. Adding two numbers with dataflow variables

(defn df-add [a b]

(let [x (pd/df-val)

y (pd/df-val)

sum (pd/df-var #(+ @x @y))]

(x a)

(y b)

@sum))

The value of the dataflow variable sum declared in the df-add function shown above will be recalculated when the referenced dataflow variables x and y are set to a value. The variables x and y are set by calling them as functions. Similarly, we can add a number to each element in a range of numbers using the df-val and df-var functions as shown in Example 9.3 below.

Example 9.3. Adding a number to a range of number with dataflow variables

(defn df-add-to-range [a r]

(let [x (pd/df-val)

y (pd/df-var)

sum (pd/df-var #(+ @x @y))

f (pc/fiber

(for [i r]

(do

(y i)

(pc/sleep 10)

@sum)))]

(x a)

(pc/join f)))

The df-add-to-range function defined above defines the dataflow variables x, y and sum, where sum is dependent on x and y. The function then creates a fiber f that uses the for macro to return a sequence of values. Within the body of the for macro, the dataflow variable y is set to a value from the range r, and the value @sum is returned. The fiber thus returns the result of adding a to all elements in the range r, as shown in the output below:

user> (df-add-to-range 2 (range 10))

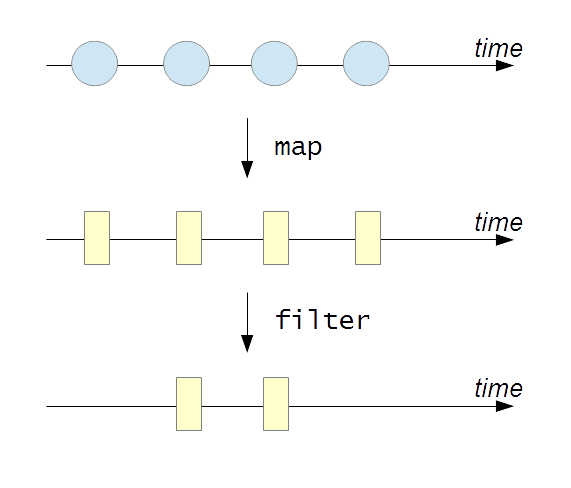
(2 3 4 5 6 7 8 9 10 11)

In conclusion, we can use the df-val and df-var functions to define dataflow variables, whose value can be recomputed when its referenced variables are changed. Effectively, changing the state of a dataflow variable may cause other dataflow variables to react to the change. Also, fibers can be used in combination with dataflow variables.

We should note that the Pulsar library also implements channels, which are analogous to channels from the core.async library. In a nutshell, channels can be used to exchange data with fibers. The Pulsar library also provides constructs for reactive programming with channels, through the co.paralleluniverse.pulsar.rx namespace. These constructs are generally termed as reactive extensions, and are very similar to transducers, in the sense that they perform some computation on the values in a channel. Reactive extensions are also implemented by the RxClojure library. We should note that one of the limitations of both the Pulsar and RxClojure libraries is that they are available only on the JVM, and can’t be used from ClojureScript programs. Thus, using core.async channels with transducers is a more feasible option in ClojureScript. Nevertheless, we will briefly explore reactive extensions through the RxClojure library in the following section.

# Using Reactive Extensions

Reactive extensions (written as Rx) are an object-oriented approach to reactive programming. In Rx, asynchronous event streams are termed as observables. An entity or object that subscribes to events from an observable is called an observer. Reactive extensions are essentially a library of functions, or methods, to manipulate observables and create objects that conform to the observer-observable pattern. For example, an observable can be transformed using the Rx variants of the map and filter functions as shown in the following illustration.



Insert Image B05024\_09\_01.png

As shown above, an observable is collection of values that vary over a period of time. It’s quite evident that observables can be treated as sequences of values using the Rx flavored versions of the map and filter functions.

We will now discuss the various constructs of the RxClojure library (https://github.com/ReactiveX/RxClojure). There are several implementations of Rx across multiple languages, such as C#, Java and PHP. The Java library for reactive extensions is RxJava, and the RxClojure library provides Clojure bindings to RxJava. As we mentioned earlier, it’s important to note that RxClojure can only be used on the JVM. Also, the RxClojure library predates the implementation of transducers in Clojure, thus making channels and transducers are a more portable and generalized approach to reactive programming.

The following library dependencies are required for the upcoming examples.

[io.reactivex/rxclojure "1.0.0"]

Also, the following namespaces must be included in your namespace declaration.

(ns my-namespace

(:require [rx.lang.clojure.core :as rx]

[rx.lang.clojure.blocking :as rxb]

[rx.lang.clojure.interop :as rxj]))

The rx.lang.clojure.core namespace contains functions for creating and manipulating observables. Observables are internally represented as collections of values. To extract values from observables, we can use functions from the rx.lang.clojure.blocking namespace. We must note that functions from the rx.lang.clojure.blocking namespace must be avoided and used only for testing. The rx.lang.clojure.interop namespace contains functions that help in performing Java interop with the underlying RxJava library.

The following examples can be found in src/m\_clj/c9/rx.clj of the books source code.

A value can be converted to an observable using the return function from the rx.lang.clojure.core namespace. An observable can be converted to a vector of values using the rx.lang.clojure.blocking/into function, and similarly we can obtain the first value of an observable using the rx.lang.clojure.blocking/first function. These functions are demonstrated in the REPL output shown below:

user> (def o (rx/return 0))

#'user/o

user> (rxb/into [] o)

[0]

user> (rxb/first o)

0

A sequence of values can be converted to an observable using the seq->o function from the rx.lang.clojure.core namespace. To convert the observable to a sequence, we pass it to the o->seq function from the rx.lang.clojure.blocking namespace. For example, we can convert the vector [1 2 3] to an observable and back to a sequence as shown below:

user> (def o (rx/seq->o [1 2 3]))

#'user/o

user> (rxb/o->seq o)

(1 2 3)

Another way to create an observable is by using the cons and empty functions from the rx.lang.clojure.core namespace. The empty function creates an observable with no values, and the cons function adds combines a value and an observable into a new observable, similar to the standard cons function. We can create an observable containing the value 0 using the cons and empty functions as shown below:

user> (def o (rx/cons 0 (rx/empty)))

#'user/o

user> (rxb/first o)

0

As we mentioned earlier, observers can subscribe to events from observables. Observers can be defined by implementing the rx.lang.clojure.Observer interface. This interface defines three methods, namely onNext, onError and onCompleted. The onNext method is called whenever an observable produces a new value, and the onCompleted method is called when an observable is done producing values. The onError method will be called in case an exception is encountered. Interestingly, all of these methods will be invoked asynchronously from an observable. For example, we can create an observer using the reify form to implement the Observer interface as shown in Example 9.4.

Example 9.4. Implementing the rx.lang.clojure.Observer interface

(def observer

(reify rx.Observer

(onNext [this v] (println (str "Got value: " v "!")))

(onError [this e] (println e))

(onCompleted [this] (println "Done!"))))

An observable can call the methods of an observer using the on-next, on-error and on- completed functions from rx.lang.clojure.core namespace. We can also define an observable using these functions and the observable\* form from the rx.lang.clojure.core namespace. The observable\* form must be passed a function that takes a single argument that represents an observer. For example, we can define a function to create an observable of two values using the observable\* form as shown in Example 9.5 below.

Example 9.5. Creating an observable using the observable\* form

(defn make-observable []

(rx/observable\* (fn [s]

(-> s

(rx/on-next :a)

(rx/on-next :b)

rx/on-completed))))

The function passed to the observable\* form shown above calls the on-next and on-completed functions to produce an observable of two values. We can convert this observable into a vector using the into function from the rx.lang.clojure.blocking namespace, as shown below:

user> (def o (make-observable))

#'user/o

user> (rxb/into [] o)

[:a :b]

An observer can also be created using the subscribe function from the rx.lang.clojure.core namespace. This function must be passed a function that takes a single value, and an observer will be created with the supplied function as its onNext method. We can also pass a second argument representing the onError method, as well as a third argument that represents the onCompleted method, to the subscribe function. For example, we can subscribe to an observable using the subscribe function, and apply a function to all values in the observable using the rx.lang.clojure.core/map function, as shown in Example 9.6 below.

Example 9.6. Subscribing to an observable using the subscribe function

(defn rx-inc [o]

(rx/subscribe o (fn [v] (println (str "Got value: " v "!"))))

(rx/map inc o))

We can create an observable and pass it to the rx-inc function defined in Example 9.6, as shown below:

user> (def o (rx/seq->o [0 1 2]))

#'user/o

user> (rx-inc o)

Got value: 0!

Got value: 1!

Got value: 2!

#<rx.Observable 0xc3fae8>

The function passed to the subscribe form in Example 9.6 is executed every time the inc function is applied to a value in the observable o. We could as well define the rx-inc function using Java interop with RxJava, as shown in Example 9.7 below.

Example 9.7. Subscribing to an observable using the Java interop

(defn rxj-inc [o]

(.subscribe o (rxj/action [v] (println (str "Got value: " v "!"))))

(.map o (rxj/fn [v] (inc v))))

It’s quite clear that working with the RxJava library through Java interop isn’t pretty, as we would have to wrap all functions in action and fn forms from the rx.lang.clojure.interop namespace. The action macro is used to represent a function that performs a side-effect, whereas the fn macro is used to wrap functions that return values. Observables can also be created using Java interop. This is done using the from static method from the rx.lang.clojure.core.Observable class. The following output demonstrates this method as well as the rxj-inc function defined in Example 9.7:

user> (def o (rx.Observable/from [0 1 2]))

#'user/o

user> (rxj-inc o)

Got value: 0!

Got value: 1!

Got value: 2!

#<rx.Observable 0x16459ef>

Of course, we should prefer to use functions from the rx.lang.clojure.core namespace, and we are using Java interop here only to show that it is indeed possible. Similar to the map function used in Example 9.6, there are several other functions in the rx.lang.clojure.core namespace that allow us to treat observables as sequences. Thus, functions such as map, filter and mapcat form the interface of observables, and describe the many ways in which we can interact with them. For example, the following output demonstrates the Rx variants of the take, cycle and range functions:

user> (rxb/into [] (->> (rx/range)

(rx/take 10)))

[0 1 2 3 4 5 6 7 8 9]

user> (rxb/into [] (->> (rx/cycle (rx/return 1))

(rx/take 5)))

[1 1 1 1 1]

The rx.lang.clojure.core namespace also provides a filter function that can be used with an observable and a predicate, as shown below:

user> (rxb/into [] (->> (rx/seq->o [:a :b :c :d :e])

(rx/filter #{:b :c})))

[:b :c]

The group-by and mapcat functions from the rx.lang.clojure.core namespace have the same semantics as the standard versions of these functions. For example, let’s define a function that uses the group-by and mapcat functions, as shown in Example 9.8 below.

Example 9.8. Using the group-by and mapcat functions

(defn group-maps [ms]

(->> ms

(rx/seq->o)

(rx/group-by :k)

(rx/mapcat (fn [[k vs :as me]]

(rx/map #(vector k %) vs)))

(rxb/into [])))

The group-maps function defined above will transform a number of maps into an observable, group the maps by the value of the key :k, and create a number of vectors using the mapcat and map functions. Of course, we wouldn’t really need such a function in the real world, and it’s only shown here to demonstrate how the group-by and mapcat functions can be used. We can pass a vector of maps to the group-maps function to produce a sequence of vectors, as shown below:

user> (group-maps [{:k :a :v 1}

{:k :b :v 2}

{:k :a :v 3}

{:k :c :v 4}])

[[:a {:k :a, :v 1}]

[:a {:k :a, :v 3}]

[:b {:k :b, :v 2}]

[:c {:k :c, :v 4}]]

Several observables can be combined using the merge function from the rx.lang.clojure.core namespace. The merge function can be passed any number of observables, as shown below:

user> (let [o1 (rx/seq->o (range 5))

o2 (rx/seq->o (range 5 10))

o (rx/merge o1 o2)]

(rxb/into [] o))

[0 1 2 3 4 5 6 7 8 9]

An observable can also be split up into two observables using the split-with function from the rx.lang.clojure.core namespace. This function must be passed an observable and a predicate function, as shown below:

user> (->> (range 6)

rx/seq->o

(rx/split-with (partial >= 3))

rxb/first

(map (partial rxb/into [])))

([0 1 2 3] [4 5])

In summary, the RxClojure library provides us with several constructs for creating and manipulating observables. We can also easily create observers that asynchronously react to observables using the subscribe function from this library. Also, the rx.lang.clojure.core namespace provides constructs that operate on observables and have similar semantics as standard Clojure functions such as map, filter and mapcat. There are several functions in the rx.lang.clojure.core namespace that we haven’t described, and you’re encouraged to find out about them on your own.

# Functional Reactive Programming

A more functional flavor of reactive programming is functional reactive programming (abbreviated as FRP). FRP was first described in the late 90s by Conal Elliot, who was a member of the Microsoft Graphics Research Group at the time, and Paul Hudak, a major contributor to the Haskell programming language. FRP is originally described as a bunch of functions to interact with events and behaviors. Both events and behaviors represent values that change over time. The major difference between these two is that events are values that change discretely over time, whereas behaviors are continuously changing values. There is no mention of an observer-observable pattern in FRP. Also, programs are written as transformations of events and behaviors. For this reason, these abstractions are also termed as a compositional event systems (or CESs).

Modern implementations of FRP provide constructs to create and transform asynchronous event streams. Also, any form of state change is represented as an event stream. In this light, a button clicked, a request made to a server, or mutating a variable, can all be thought of as event streams. The Bacon.js (https://github.com/baconjs/bacon.js/) library is a JavaScript implementation of FRP, and the Yolk library (https://github.com/Cicayda/yolk) provides ClojureScript bindings to the Bacon.js library. In this section, we will briefly study the constructs provided by the Yolk library.

The following library dependencies are required for the upcoming examples.

[yolk "0.9.0"]

Also, the following namespaces must be included in your namespace declaration.

(ns my-namespace

(:require [yolk.bacon :as y]))

In addition to the above dependencies, the following examples also use the set-html! and by-id functions from src/m\_clj/c9/common.cljs. These functions are defined as follows.

(defn ^:export by-id [id]

(.getElementById js/document id))

(defn ^:export set-html! [el s]

(set! (.-innerHTML el) s))

Ensure that the code in the following ClojureScript examples is compiled, by using the following command.

$ lein cljsbuild once

The yolk.bacon namespace provides several functions to create event streams, such as later and interval functions. The later function creates an event stream with a single value that must produce a value after a given delay. The interval function can infinitely repeat a value in a given time interval. Both these functions must be passed a number of milliseconds as the first argument and a value to produce as the second argument.

Event streams in the Bacon.js library may produce an infinite number of values. We can limit the number of values produced by an event stream by using the yolk.bacon/sliding-window function, which creates an event stream that drops older values once it’s full. This function must be passed an event stream and a number indicating the capacity of the returned event stream.

We can also create an event bus, onto which we can arbitrarily push values, using the bus function from the yolk.bacon namespace. The push function puts a value onto an event bus, and the plug function connects an event bus to another event stream. We should note that event buses are not really functional in design, as the push function can be called with an event bus arbitrarily. For this reason, an event stream must always be preferred over an event bus.

To listen to values produced from event streams, we can use the on-value, on-error and on-end functions. The on-value and on-error functions will call a supplied a 1-arity function whenever a given event stream produces a value or an error, respectively. The on-end function will call a supplied function that takes no arguments whenever a stream ends. This function is often used with the yolk.bacon/never function, which creates an event stream that ends immediately without producing a value.

Event streams can also be combined in several ways. The merge-all function combines a vector of several event streams into a single one. Another function that can collect values from several event streams in this way is the flat-map function. Alternatively, the combine-array function can be used to create a single event stream that produces arrays of the values from the supplies streams. The yolk.bacon/when function can be used to conditionally combine several channels. This function must be passed a number of clauses, similar to the cond form. Each clause must have two parts – a vector of event streams and a 1-arity function that will be invoked when all the supplied event streams produce values.

The yolk.bacon namespace also provides event stream based variants of the standard map, filter and take functions. These variants of the standard functions take an event stream as the first argument, which is different from the semantics of the standard versions.

Using these functions from the Yolk library, we can implement a simplified ClojureScript based solution to the dining philosophers problem, which we described in the previous chapters. For a detailed explanation of the dining philosophers problem and its solution, refer to Chapter 2 and Chapter 8.

The following examples can be found in src/m\_clj/c9/yolk/core.cljs of the books source code. Also, the HTML page for the following ClojureScript examples can be found in resources/html/yolk.html. The following scripts will be included in this page.

<script type="text/javascript" src="../js/bacon.js"></script>

<script type="text/javascript" src="../js/out/yolk.js"></script>

In this implementation of the dining philosophers problem, we shall represent the state of the philosophers and the forks on the table using event buses. We won’t maintain much state of the philosophers for the sake of simplicity. The event buses can then be combined using the when function from the Yolk library. Let’s first define functions to render the philosophers and represent the routine of a philosopher, as shown in Example 9.9 below.

Example 9.9. Solving the dining philosophers problem with event streams

(defn render-philosophers [philosophers]

(apply str

(for [p (reverse philosophers)]

(str "<div>" p "</div>"))))

(defn philosopher-fn [i n forks philosophers wait-ms]

(let [p (nth philosophers i)

fork-1 (nth forks i)

fork-2 (nth forks (-> i inc (mod n)))]

(fn []

(js/setTimeout

(fn []

(y/push fork-1 :fork)

(y/push fork-2 :fork)

(y/push p {}))

wait-ms)

(str "Philosopher " (inc i) " ate!"))))

The render-philosophers function shown above will wrap each philosopher in a div tag, which will be displayed on a web page. The philosopher-fn function returns a function that represents the routine of a philosopher. The function returned by the philosopher-fn function sets off a task, using the setTimeout function, to push more values into the event buses representing a particular philosopher and the forks on her left and right hand side. This function will finally return a string indicating that the given philosopher was able to eat the food supplied to him. Using these functions, we can create a simulation of the dining philosophers in a web page, as shown in Example 9.10 below.

Example 9.10. Solving the dining philosophers problem with event streams (continued)

(let [out (by-id "ex-9-10-out")

n 5

[f1 f2 f3 f4 f5 :as forks] (repeatedly n #(y/bus))

[p1 p2 p3 p4 p5 :as philosophers] (repeatedly n #(y/bus))

eat #(philosopher-fn % n forks philosophers 1000)

events (y/when [p1 f1 f2] (eat 0)

[p2 f2 f3] (eat 1)

[p3 f3 f4] (eat 2)

[p4 f4 f5] (eat 3)

[p5 f5 f1] (eat 4))]

(-> events

(y/sliding-window n)

(y/on-value

#(set-html! out (render-philosophers %))))

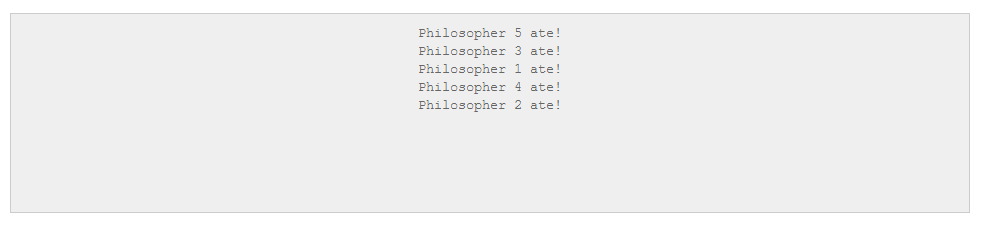
(doseq [f forks]

(y/push f :fork))

(doseq [p philosophers]

(y/push p {})))

In the let form shown in Example 9.10 above, we create the philosophers and forks in our simulation using the bus function from the Yolk library. The values produced by these event buses are then combined using a when form. The when function in the above code will check for events from a philosopher and the forks on her left and right hand side. The combinations of philosophers and forks are, in a way, hardcoded into the clauses of the when form. Of course, we must understand that the clauses of the when form shown above could easily have been generated using a macro. Values are then placed onto the event buses representing the philosophers and forks using the push function, to start the simulation. The last five philosophers who could eat are rendered in HTML as shown below.



Insert Image B05024\_09\_02.png

In summary, the Yolk library provides several constructs to handle event streams. There are several functions from this library that we haven’t discussed, and you should explore them on your own. In the following section, we will provide examples that demonstrate the other functions from the Yolk library.

# Building Reactive User Interfaces

One of the primary applications of reactive programming is user interface design, where we must create UI components that react asynchronously to changes in state. In this section, we will describe a few examples implemented using the core.async library and the Yolk library. This is meant to give us a comparison between channels and event streams, and how we can design solutions using both these concepts. Note that only the overall design and code for these examples will be described, and you should be able to fill in the details on your own.

The following library dependencies are required for the upcoming examples.

[yolk "0.9.0"]

[org.clojure/core.async "0.1.346.0-17112a-alpha"]

Also, the following namespaces must be included in your namespace declaration.

(ns my-namespace

(:require [goog.events :as events]

[goog.events.EventType]

[goog.style :as style]

[cljs.core.async :as a]

[yolk.bacon :as y])

(:require-macros [cljs.core.async.macros :refer [go go-loop alt!]]))

In addition to the above dependencies, the following examples also use the set-html! And by-id functions from src/m\_clj/c9/common.cljs. Ensure that the code in the following ClojureScript examples is compiled, by using the following command.

$ lein cljsbuild once

As a first example, let’s create three asynchronous tasks that each produce values at different time intervals. We must fetch all the values produced by these tasks and render them into a web page in the same order.

The following examples can be found in src/m\_clj/c9/reactive/core.cljs of the books source code. Also, the HTML page for the following ClojureScript examples can be found in resources/html/reactive.html. The following scripts will be included in this page.

<script type="text/javascript" src="../js/bacon.js"></script>

<script type="text/javascript" src="../js/out/reactive.js"></script>

We could implement this using processes and channels from the core.async library. Channels will convey the values produced by three processes, and we will use a merge operation to combine these channels, as shown in Example 9.11.

Example 9.11. Three asynchronous tasks using channels

(defn render-div [q]

(apply str

(for [p (reverse q)]

(str "<div class='proc-" p "'>Process " p "</div>"))))

(defn start-process [v t]

(let [c (a/chan)]

(go (while true

(a/<! (a/timeout t))

(a/>! c v)))

c))

(let [out (by-id "ex-9-11-out")

c1 (start-process 1 250)

c2 (start-process 2 1000)

c3 (start-process 3 1500)

c (a/merge [c1 c2 c3])

firstn (fn [v n]

(if (<= (count v) n)

v

(subvec v (- (count v) n))))]

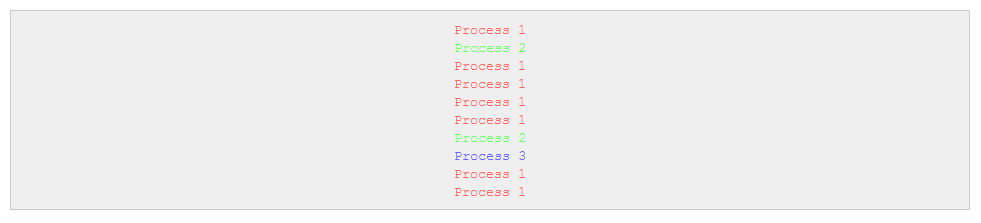
(go-loop [q []]

(set-html! out (render-div q))

(recur (-> (conj q (a/<! c))

(firstn 10)))))

The start-process function shown above will create a process that periodically produces values using the go form, and returns a channel from which the values can be read. The render-div function will create div tags for the values produced by the three tasks. Only the ten most recent values will be shown. This code will produce the following output.



Insert Image B05024\_09\_03.png

We could also implement the above example using FRP, in which values produced each of the three tasks are event streams. The merge-all function from the yolk.bacon namespace can be used to combine the event streams, and the sliding-window function can obtain the ten most recent values produced by the resulting stream. The render-div function from Example 9.11 can be reused here to render the values. This is implemented as shown in Example 9.12 below, and produces the same output as Example 9.11.

Example 9.12. Three asynchronous tasks using FRP

(let [out (by-id "ex-9-12-out")

events [(y/interval 250 1)

(y/interval 1000 2)

(y/interval 1500 3)]]

(-> events

y/merge-all

(y/sliding-window 10)

(y/on-value

#(set-html! out (render-div %)))))

Next, let’s try to capture mouse events from a particular div tag, and display the page offset values of the locations of these events. We can do this with channels, but we would first need a function to convey events onto a channel. We can implement this using the goog.events/listen and cljs.core.async/put! functions, as shown in Example 9.13 below.

Example 9.13. A function to convey events onto a channel

(defn listen

([el type] (listen el type nil))

([el type f] (listen el type f (a/chan)))

([el type f out]

(events/listen el type

(fn [e] (when f (f e)) (a/put! out e)))

out))

We can now use the listen function defined above to listen to the goog.events.EventType.MOUSEMOVE event type from a particular div tag. The values will have to be converted to page offsets, and this can be done using the getPageOffsetLeft and getPageOffsetTop functions from the goog.style namespace. This implementation is described in Example 9.14 below.

Example 9.14. Mouse events using channels

(defn offset [el]

[(style/getPageOffsetLeft el) (style/getPageOffsetTop el)])

(let [el (by-id "ex-9-14")

out (by-id "ex-9-14-out")

events-chan (listen el goog.events.EventType.MOUSEMOVE)

[left top] (offset el)

location (fn [e]

{:x (+ (.-offsetX e) (int left))

:y (+ (.-offsetY e) (int top))})]

(go-loop []

(if-let [e (a/<! events-chan)]

(let [loc (location e)]

(set-html! out (str (:x loc) ", " (:y loc)))

(recur)))))

We can also implement this using from-event-stream and map functions from the Yolk library. Interestingly, the events produced by the stream returned by the from-event-target function will have page offsets of the event stored as the pageX and pageY properties. This allows us to have a much simpler implementation, as shown in Example 9.15 below.

Example 9.15. Mouse events using FRP

(let [el (by-id "ex-9-15")

out (by-id "ex-9-15-out")

events (y/from-event-target el "mousemove")]

(-> events

(y/map (juxt (fn [e] (.-pageX e))

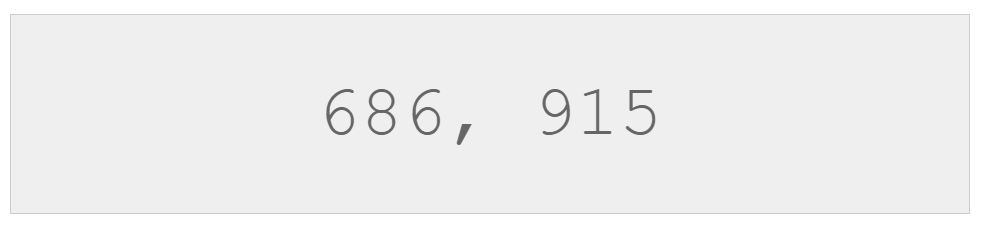
(fn [e] (.-pageY e))))

(y/map (fn [[x y]] (str x ", " y)))

(y/on-value

#(set-html! out %))))

Both of the implementations shown in Example 9.14 and Example 9.15 work as expected, and produce the following output.



Insert Image B05024\_09\_04.png

As a final example, we will simulate several search queries and display the results from the first three queries that return results. The queries can be described as follows - there are two queries for web results, two queries for image results, and two queries for video results. We can implement these simulated queries as shown in Example 9.16 below.

Example 9.16. Simulating search queries with channels

(defn chan-search [kind]

(fn [query]

(go

(a/<! (a/timeout (rand-int 100)))

[kind query])))

(def chan-web1 (chan-search :web1))

(def chan-web2 (chan-search :web2))

(def chan-image1 (chan-search :image1))

(def chan-image2 (chan-search :image2))

(def chan-video1 (chan-search :video1))

(def chan-video2 (chan-search :video2))

The chan-search function returns a function that uses the cljs.core.async/timeout function to simulate a search query by parking for a random number of milliseconds. Using the chan-search function, we create several queries for the different kinds of results we are interested in. Using these functions, we can implement a function to perform all the queries and return the first three results, as shown in Example 9.17 below.

Example 9.17. Simulating search queries with channels (continued)

(defn chan-search-all [query & searches]

(let [cs (for [s searches]

(s query))]

(-> cs vec a/merge)))

(defn chan-search-fastest [query]

(let [t (a/timeout 80)

c1 (chan-search-all query chan-web1 chan-web2)

c2 (chan-search-all query chan-image1 chan-image2)

c3 (chan-search-all query chan-video1 chan-video2)

c (a/merge [c1 c2 c3])]

(go (loop [i 0

ret []]

(if (= i 3)

ret

(recur (inc i)

(conj ret (alt!

[c t] ([v] v)))))))))

As shown in the above example, the merge function can be used to combine channels that produce the results of the search queries. Note that the queries to the three soulrces of results, namely web, images and videos, are timed out after 80 milliseconds. We can bind the chan-search-fastest function to the click of a mouse button using the listen function we defined earlier, as shown in Example 9.18 below.

Example 9.18. Simulating search queries with channels (continued)

(let [out (by-id "ex-9-18-out")

button (by-id "search-1")

c (listen button goog.events.EventType.CLICK)]

(go (while true

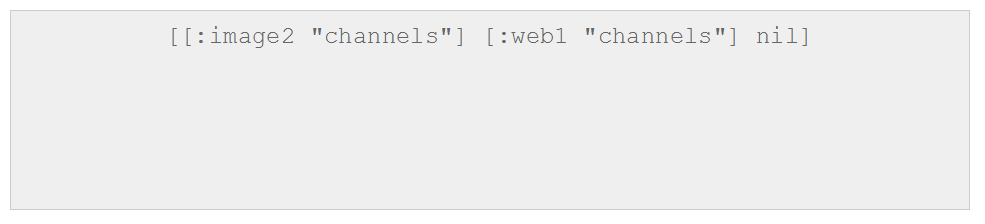
(let [e (a/<! c)

result (a/<! (chan-search-fastest "channels"))

s (str result)]

(set-html! out s)))))

Clicking on the button bound to the chan-search-fastest function will show the following output. Note that the nil value in the below output indicates a timeout of queries to a given source of results.



Insert Image B05024\_09\_05.png

We can implement an FRP version of the simulation of search queries that was previously described. The queries for the various sources of data are defined as shown in Example 9.19 below.

Example 9.19. Simulating search queries with FRP

(defn frp-search [kind]

(fn [query]

(y/later (rand-int 100) [kind query])))

(def frp-web1 (frp-search :web1))

(def frp-web2 (frp-search :web2))

(def frp-image1 (frp-search :image1))

(def frp-image2 (frp-search :image2))

(def frp-video1 (frp-search :video1))

(def frp-video2 (frp-search :video2))

The functions to shown above all return event streams for search results. The search results produced can combined with timeouts using the later, merge and combine-as-array functions from the yolk.bacon namespace, as shown in Example 9.20 below.

Example 9.20. Simulating search queries with FRP (continued)

(defn frp-search-all [query & searches]

(let [results (map #(% query) searches)

events (cons (y/later 80 "nil") results)]

(-> (apply y/merge events)

(y/take 1))))

(defn frp-search-fastest [query]

(y/combine-as-array

(frp-search-all query frp-web1 frp-web2)

(frp-search-all query frp-image1 frp-image2)

(frp-search-all query frp-video1 frp-video2)))

The frp-search-fastest function can be invoked on clicking a button, as shown below in Example 9.21.

Example 9.21. Simulating search queries with FRP (continued)

(let [out (by-id "ex-9-21-out")

button (by-id "search-2")

events (y/from-event-target button "click")]

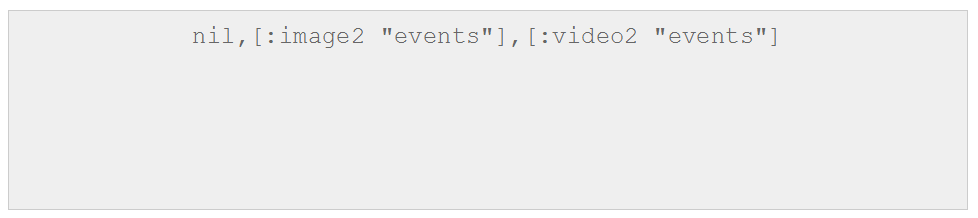
(-> events

(y/flat-map-latest #(frp-search-fastest "events"))

(y/on-value

#(set-html! out %))))

The above example produces the following output when the button shown is clicked on.



Insert Image B05024\_09\_06.png

In conclusion, we can use both channels and event streams to implement interactive interfaces in web pages. Although the FRP implementations of the above examples are slightly shorter, we can say that both core.async and Yolk libraries have their own elegance.

# Summary

So far, we have discussed reactive programming through the Pulsar, RxClojure and Yolk libraries. We have also described several ClojureScript examples that compare channels from the core.async library to reactive event streams from the Yolk library.

In the following chapter, we will explore how we can test our Clojure programs.