Clojure Reactive Programming

Design and implement highly reusable reactive applications by integrating different frameworks with Clojure

Leonardo Borges
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'What is Reactive Programming?'
- A synopsis of the book’s content
- More information on Clojure Reactive Programming

About the Author

Leonardo Borges is a programming languages enthusiast who loves writing code, contributing to open source software, and speaking on subjects he feels strongly about. After nearly 5 years of consulting at ThoughtWorks, where he worked in two commercial Clojure projects, among many others, he is now a software engineer at Atlassian. He uses Clojure and ClojureScript to help build real-time collaborative editing technology. This is his first full-length book, but he contributed a couple of chapters to Clojure Cookbook, O'Reilly.

Leonardo has founded and runs the Sydney Clojure User Group in Australia. He also writes posts about software, focusing on functional programming, on his website (http://www.leonardoborges.com). When he isn't writing code, he enjoys riding motorcycles, weightlifting, and playing the guitar.
Clojure Reactive Programming

Highly concurrent applications such as user interfaces have traditionally managed state through the mutation of global variables. Various actions are coordinated via event handlers, which are procedural in nature.

Over time, the complexity of a system increases. New feature requests come in, and it becomes harder and harder to reason about the application.

Functional programming presents itself as an extremely powerful ally in building reliable systems by eliminating mutable states and allowing applications to be written in a declarative and composable way.

Such principles gave rise to Functional Reactive Programming and Compositional Event Systems (CES), programming paradigms that are exceptionally useful in building asynchronous and concurrent applications. They allow you to model mutable states in a functional style.

This book is devoted to these ideas and presents a number of different tools and techniques to help manage the increasing complexity of modern systems.

What This Book Covers

Chapter 1, What is Reactive Programming?, starts by guiding you through a compelling example of a reactive application written in ClojureScript. It then takes you on a journey through the history of Reactive Programming, during which some important terminology is introduced, setting the tone for the following chapters.

Chapter 2, A Look at Reactive Extensions, explores the basics of this Reactive Programming framework. Its abstractions are introduced and important subjects such as error handling and back pressure are discussed.

Chapter 3, Asynchronous Programming and Networking, walks you through building a stock market application. It starts by using a more traditional approach and then switches to an implementation based on Reactive Extensions, examining the trade-offs between the two.

Chapter 4, Introduction to core.async, describes core.async, a library for asynchronous programming in Clojure. Here, you learn about the building blocks of Communicating Sequential Processes and how Reactive Applications are built with core.async.

Chapter 5, Creating Your Own CES Framework with core.async, embarks on the ambitious endeavor of building a CES framework. It leverages knowledge gained in the previous chapter and uses core.async as the foundation for the framework.
Chapter 6, Building a Simple ClojureScript Game with Reagi, showcases a domain where Reactive frameworks have been used for great effects in games development.

Chapter 7, The UI as a Function, shifts gears and shows how the principles of functional programming can be applied to web UI development through the lens of Om, a ClojureScript binding for Facebook's React.

Chapter 8, Futures, presents futures as a viable alternative to some classes' reactive applications. It examines the limitations of Clojure futures and presents an alternative: imminent, a library of composable futures for Clojure.

Chapter 9, A Reactive API to Amazon Web Services, describes a case study taken from a real project, where a lot of the concepts introduced throughout this book have been put together to interact with a third-party service.

Appendix A, The Algebra of Library Design, introduces concepts from Category Theory that are helpful in building reusable abstractions. The appendix is optional and won't hinder learning in the previous chapters. It presents the principles used in designing the futures library seen in Chapter 8, Futures.

Appendix B, Bibliography, provides all the references used throughout the book.
What is Reactive Programming?

Reactive Programming is both an overloaded term and a broad topic. As such, this book will focus on a specific formulation of Reactive Programming called Compositional Event Systems (CES).

Before covering some history and background behind Reactive Programming and CES, I would like to open with a working and hopefully compelling example: an animation in which we draw a sine wave onto a web page.

The sine wave is simply the graph representation of the sine function. It is a smooth, repetitive oscillation, and at the end of our animation it will look like the following screenshot:

This example will highlight how CES:

- Urges us to think about what we would like to do as opposed to how
- Encourages small, specific abstractions that can be composed together
- Produces terse and maintainable code that is easy to change
What is Reactive Programming?

The core of this program boils down to four lines of ClojureScript:

```clojure
(-> sine-wave
 (.take 600)
 (.subscribe (fn [{:keys [x y]}]
 (fill-rect x y "orange"))))
```

Simply by looking at this code it is impossible to determine precisely what it does. However, do take the time to read and imagine what it could do.

First, we have a variable called `sine-wave`, which represents the 2D coordinates we will draw onto the web page. The next line gives us the intuition that `sine-wave` is some sort of collection-like abstraction: we use `.take` to retrieve 600 coordinates from it.

Finally, we `.subscribe` to this "collection" by passing it a callback. This callback will be called for each item in the `sine-wave`, finally drawing at the given `x` and `y` coordinates using the `fill-rect` function.

This is quite a bit to take in for now as we haven't seen any other code yet—but that was the point of this little exercise: even though we know nothing about the specifics of this example, we are able to develop an intuition of how it might work.

Let's see what else is necessary to make this snippet animate a sine wave on our screen.

A taste of Reactive Programming

This example is built in ClojureScript and uses HTML 5 Canvas for rendering and RxJS (see https://github.com/Reactive-Extensions/RxJS)—a framework for Reactive Programming in JavaScript.

Before we start, keep in mind that we will not go into the details of these frameworks yet—that will happen later in this book. This means I'll be asking you to take quite a few things at face value, so don't worry if you don't immediately grasp how things work. The purpose of this example is to simply get us started in the world of Reactive Programming.

For this project, we will be using Chestnut (see https://github.com/plexus/chestnut)—a leiningen template for ClojureScript that gives us a sample working application we can use as a skeleton.

To create our new project, head over to the command line and invoke leiningen as follows:

```
lein new chestnut sin-wave
cd sin-wave
```
Next, we need to modify a couple of things in the generated project. Open up `sin-wave/resources/index.html` and update it to look like the following:

```html
<!DOCTYPE html>
<html>
<head>
  <link href="css/style.css" rel="stylesheet" type="text/css">
</head>
<body>
  <div id="app"></div>
  <script src="/js/rx.all.js" type="text/javascript"></script>
  <script src="/js/app.js" type="text/javascript"></script>
  <canvas id="myCanvas" width="650" height="200" style="border:1px solid #d3d3d3;">"</canvas>
</body>
</html>
```

This simply ensures that we import both our application code and RxJS. We haven't downloaded RxJS yet so let's do this now. Browse to [https://github.com/Reactive-Extensions/RxJS/blob/master/dist/rx.all.js](https://github.com/Reactive-Extensions/RxJS/blob/master/dist/rx.all.js) and save this file to `sin-wave/resources/public`. The previous snippets also add an HTML 5 Canvas element onto which we will be drawing.

Now, open `/src/cljs/sin_wave/core.cljs`. This is where our application code will live. You can ignore what is currently there. Make sure you have a clean slate like the following one:

```cljs
(ns sin-wave.core)
(defn main [])
```

Finally, go back to the command line—under the `sin-wave` folder—and start up the following application:

```
lein run -m sin-wave.server
```

```
Starting figwheel.
Starting web server on port 10555.
Compiling ClojureScript.
Figwheel: Starting server at http://localhost:3449
Figwheel: Serving files from '(dev-resources|resources)/public'
```
What is Reactive Programming?

Once the previous command finishes, the application will be available at http://localhost:10555, where you will find a blank, rectangular canvas. We are now ready to begin.

The main reason we are using the Chestnut template for this example is that it performs hot-reloading of our application code via websockets. This means we can have the browser and the editor side by side, and as we update our code, we will see the results immediately in the browser without having to reload the page.

To validate that this is working, open your web browser's console so that you can see the output of the scripts in the page. Then add this to /src/cljs/sin_wave/core.cljs as follows:

```clojure
(.log js/console "hello clojurescript")
```

You should have seen the hello clojurescript message printed to your browser's console. Make sure you have a working environment up to this point as we will be relying on this workflow to interactively build our application.

It is also a good idea to make sure we clear the canvas every time Chestnut reloads our file. This is simple enough to do by adding the following snippet to our core namespace:

```clojure
(def canvas (.getElementById js/document "myCanvas"))
(def ctx    (.getContext canvas "2d"))

;;; Clear canvas before doing anything else
(.clearRect ctx 0 0 (.width canvas) (.height canvas))
```

Creating time

Now that we have a working environment, we can progress with our animation. It is probably a good idea to specify how often we would like to have a new animation frame.

This effectively means adding the concept of time to our application. You're free to play with different values, but let's start with a new frame every 10 milliseconds:

```clojure
(def interval   js/Rx.Observable.interval)
(def time       (interval 10))
```

As RxJS is a JavaScript library, we need to use ClojureScript's interoperability to call its functions. For convenience, we bind the interval function of RxJS to a local var. We will use this approach throughout this book when appropriate.
Next, we create an infinite stream of numbers—starting at 0—that will have a new element every 10 milliseconds. Let's make sure this is working as expected:

```clojure
(-> time
    (.take 5)
    (.subscribe (fn [n]
        (.log js/console n))))
;; 0
;; 1
;; 2
;; 3
;; 4
```

I use the term stream very loosely here. It will be defined more precisely later in this book.

Remember time is infinite, so we use .take in order to avoid indefinitely printing out numbers to the console.

Our next step is to calculate the 2D coordinate representing a segment of the sine wave we can draw. This will be given by the following functions:

```clojure
(defn deg-to-rad [n]
  (* (/ Math/PI 180) n))

(defn sine-coord [x]
  (let [sin (Math/sin (deg-to-rad x))
        y   (- 100 (* sin 90))]
    {:x   x
     :y   y
     :sin sin})
```

The `sine-coord` function takes an x point of our 2D Canvas and calculates the y point based on the sine of x. The constants 100 and 90 simply control how tall and sharp the slope should be. As an example, try calculating the sine coordinate when x is 50:

```clojure
(.log js/console (str (sine-coord 50)))
;;{:x 50, :y 31.05600011929198, :sin 0.766044443118978}
```

We will be using `time` as the source for the values of x. Creating the sine wave now is only a matter of combining both `time` and `sine-coord`:

```clojure
(def sine-wave
  (.map time sine-coord))
What is Reactive Programming?

Just like time, sine-wave is an infinite stream. The difference is that instead of just integers, we will now have the x and y coordinates of our sine wave, as demonstrated in the following:

```clojure
(-> sine-wave
    (.take 5)
    (.subscribe (fn [xysin]
                  (js/console (str xysin))))))
```

This brings us to the original code snippet which piqued our interest, alongside a function to perform the actual drawing:

```clojure
(defn fill-rect [x y colour]
  (set! (.fillStyle ctx) colour)
  (.fillRect ctx x y 2 2))
```

As this point, we can save the file again and watch as the sine wave we have just created gracefully appears on the screen.

More colors

One of the points this example sets out to illustrate is how thinking in terms of very simple abstractions and then building more complex ones on top of them make for code that is simpler to maintain and easier to modify.

As such, we will now update our animation to draw the sine wave in different colors. In this case, we would like to draw the wave in red if the sine of x is negative and blue otherwise.

We already have the sine value coming through the sine-wave stream, so all we need to do is to transform this stream into one that will give us the colors according to the preceding criteria:

```clojure
(def colour (.map sine-wave
                (fn [{:keys [sin]}]
                    (fill-rect x y "orange"))))
```

As this point, we can save the file again and watch as the sine wave we have just created gracefully appears on the screen.
The next step is to add the new stream into the main drawing loop—remember to comment the previous one so that we don't end up with multiple waves being drawn at the same time:

```
(-> (.zip sine-wave colour #(vector % %2))
  (.take 600)
  (.subscribe (fn [[{:keys [x y]} colour]]
                (fill-rect x y colour))))
```

Once we save the file, we should see a new sine wave alternating between red and blue as the sine of \(x\) oscillates from \(-1\) to \(1\).

### Making it reactive

As fun as this has been so far, the animation we have created isn't really reactive. Sure, it does react to time itself, but that is the very nature of animation. As we will later see, Reactive Programming is so called because programs react to external inputs such as mouse or network events.

We will, therefore, update the animation so that the user is in control of when the color switch occurs: the wave will start red and switch to blue when the user clicks anywhere within the canvas area. Further clicks will simply alternate between red and blue.

We start by creating infinite—as per the definition of time—streams for our color primitives as follows:

```
(def red  (.map time (fn [_] "red")))
(def blue (.map time (fn [_] "blue")))
```

On their own, `red` and `blue` aren't that interesting as their values don't change. We can think of them as constant streams. They become a lot more interesting when combined with another infinite stream that cycles between them based on user input:

```
(def concat     js/Rx.Observable.concat)
(def defer      js/Rx.Observable.defer)
(def from-event js/Rx.Observable.fromEvent)
```

```
(def mouse-click (from-event canvas "click"))
```
What is Reactive Programming?

(def cycle-colour
  (concat (.takeUntil red mouse-click)
          (defer #(concat (.takeUntil blue mouse-click)
                        cycle-colour))))

This is our most complex update so far. If you look closely, you will also notice that cycle-colour is a recursive stream; that is, it is defined in terms of itself.

When we first saw code of this nature, we took a leap of faith in trying to understand what it does. After a quick read, however, we realized that cycle-colour follows closely how we might have talked about the problem: we will use red until a mouse click occurs, after which we will use blue until another mouse click occurs. Then, we start the recursion.

The change to our animation loop is minimal:

```
  (-> (.zip sine-wave cycle-colour #(vector % %2))
      .take 600
      .subscribe (fn [[{:keys [x y]} colour]]
                   (fill-rect x y colour)))
```

The purpose of this book is to help you develop the instinct required to model problems in the way demonstrated here. After each chapter, more and more of this example will make sense. Additionally, a number of frameworks will be used both in ClojureScript and Clojure to give you a wide range of tools to choose from.

Before we move on to that, we must take a little detour and understand how we got here.

Exercise 1.1

Modify the previous example in such a way that the sine wave is drawn using all rainbow colors. The drawing loop should look like the following:

```
  (-> (.zip sine-wave rainbow-colours #(vector % %2))
      .take 600
      .subscribe (fn [[{:keys [x y]} colour]]
                   (fill-rect x y colour)))
```

Your task is to implement the rainbow-colours stream. As everything up until now has been very light on explanations, you might choose to come back to this exercise later, once we have covered more about CES.

The repeat, scan, and flatMap functions may be useful in solving this exercise. Be sure to consult RxJS' API at https://github.com/Reactive-Extensions/RxJS/blob/master/doc/libraries/rx.complete.md.
A bit of history

Before we talk about what Reactive Programming is, it is important to understand how other relevant programming paradigms influenced how we develop software. This will also help us understand the motivations behind reactive programming.

With few exceptions most of us have been taught—either self-taught or at school/university—imperative programming languages such as C and Pascal or object-oriented languages such as Java and C++.

In both cases, the imperative programming paradigm—of which object-oriented languages are part—dictates we write programs as a series of statements that modify program state.

In order to understand what this means, let's look at a short program written in pseudo-code that calculates the sum and the mean value of a list of numbers:

\[
\begin{align*}
\text{numbers} & := [1, 2, 3, 4, 5, 6] \\
\text{sum} & := 0 \\
\text{for each number in numbers} & \\
\quad \text{sum} & := \text{sum} + \text{number} \\
\text{end} \\
\text{mean} & := \text{sum} / \text{count(numbers)}
\end{align*}
\]

The mean value is the average of the numbers in the list, obtained by dividing the sum by the number of elements.

First, we create a new array of integers, called \text{numbers}, with numbers from 1 to 6, inclusive. Then, we initialize \text{sum} to zero. Next, we iterate over the array of integers, one at a time, adding to \text{sum} the value of each number.

Lastly, we calculate and assign the average of the numbers in the list to the \text{mean} local variable. This concludes the program logic.

This program would print 21 for the sum and 3 for the mean, if executed.

Though a simple example, it highlights its imperative style: we set up an application state—\text{sum}—and then explicitly tell the computer how to modify that state in order to calculate the result.
What is Reactive Programming?

Dataflow programming

The previous example has an interesting property: the value of mean clearly has a dependency on the contents of sum.

Dataflow programming makes this relationship explicit. It models applications as a dependency graph through which data flows—from operation to operation—and as values change, these changes are propagated to its dependencies.

Historically, dataflow programming has been supported by custom-built languages such as Lucid and BLODI, as such, leaving other general purpose programming languages out.

Let's see how this new insight would impact our previous example. We know that once the last line gets executed, the value of mean is assigned and won't change unless we explicitly reassign the variable.

However, let's imagine for a second that the pseudo-language we used earlier does support dataflow programming. In that case, assigning mean to an expression that refers to both sum and count, such as sum / count(numbers), would be enough to create the directed dependency graph in the following diagram:

![Dependency Graph](image)

Note that a direct side effect of this relationship is that an implicit dependency from sum to numbers is also created. This means that if numbers change, the change is propagated through the graph, first updating sum and then finally updating mean.

This is where Reactive Programming comes in. This paradigm builds on dataflow programming and change propagation to bring this style of programming to languages that don't have native support for it.
For imperative programming languages, Reactive Programming can be made available via libraries or language extensions. We don't cover this approach in this book, but should the reader want more information on the subject, please refer to dc-lib (see https://code.google.com/p/dc-lib/) for an example. It is a framework that adds Reactive Programming support to C++ via dataflow constraints.

**Object-oriented Reactive Programming**

When designing interactive applications such as desktop **Graphical User Interfaces** (GUIs), we are essentially using an object-oriented approach to Reactive Programming. We will build a simple calculator application to demonstrate this style.

Clojure isn't an object-oriented language, but we will be interacting with parts of the Java API to build user interfaces that were developed in an OO paradigm, hence the title of this section.

Let's start by creating a new leiningen project from the command line:

```
lein new calculator
```

This will create a directory called calculator in the current folder. Next, open the `project.clj` file in your favorite text editor and add a dependency on Seesaw, a Clojure library for working with Java Swing:

```clj
(defproject calculator "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
  :dependencies [[org.clojure/clojure "1.5.1"]
                  [seesaw "1.4.4"]])
```

At the time of this writing, the latest Seesaw version available is 1.4.4.

Next, in the `src/calculator/core.clj` file, we'll start by requiring the Seesaw library and creating the visual components we'll be using:

```clj
(ns calculator.core
  (:require [seesaw.core :refer :all]))

(native!)

(def main-frame (frame :title "Calculator" :on-close :exit))
```
What is Reactive Programming?

(def field-x (text "1"))
(def field-y (text "2"))

(def result-label (label "Type numbers in the boxes to add them up!"))

The preceding snippet creates a window with the title Calculator that ends the program when closed. We also create two text input fields, field-x and field-y, as well as a label that will be used to display the results, aptly named result-label.

We would like the label to be updated automatically as soon as a user types a new number in any of the input fields. The following code does exactly that:

(defn update-sum [e]
  (try
    (text! result-label
      (str "Sum is " (+ (Integer/parseInt (text field-x))
         (Integer/parseInt (text field-y))))
    (catch Exception e
      (println "Error parsing input."))
  )
)
(listen field-x :key-released update-sum)
(listen field-y :key-released update-sum)

The first function, update-sum, is our event handler. It sets the text of result-label to the sum of the values in field-x and field-y. We use try/catch here as a really basic way to handle errors since the key pressed might not have been a number.

We then add the event handler to the :key-released event of both input fields.

In real applications, we never want a catch block such as the previous one. This is considered bad style, and the catch block should do something more useful such as logging the exception, firing a notification, or resuming the application if possible.

We are almost done. All we need to do now is add the components we have created so far to our main-frame and finally display it as follows:

(config! main-frame :content
  (border-panel
    :north (horizontal-panel :items [field-x field-y])
    :center result-label
    :border 5))

(defn -main [& args]
  (-> main-frame pack! show!))
Now we can save the file and run the program from the command line in the project's root directory:

`lein run -m calculator.core`

You should see something like the following screenshot:

![Calculator screenshot]  
<table>
<thead>
<tr>
<th>Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="images/input_fields.png" alt="Input fields" /></td>
</tr>
<tr>
<td><img src="images/result_label.png" alt="Result label" /></td>
</tr>
<tr>
<td><img src="images/text.png" alt="Text" /></td>
</tr>
<tr>
<td><img src="images/numbers.png" alt="Numbers" /></td>
</tr>
<tr>
<td><img src="images/add_them_up.png" alt="Add them up!" /></td>
</tr>
</tbody>
</table>

Experiment by typing some numbers in either or both text input fields and watch how the value of the label changes automatically, displaying the sum of both numbers.

Congratulations! You have just created your first reactive application!

As alluded to previously, this application is reactive because the value of the result label reacts to user input and is updated automatically. However, this isn't the whole story—it lacks in composability and requires us to specify the how, not the what of what we're trying to achieve.

As familiar as this style of programming may be, making applications reactive this way isn't always ideal.

Given previous discussions, we notice we still had to be fairly explicit in setting up the relationships between the various components as evidenced by having to write a custom handler and bind it to both input fields.

As we will see throughout the rest of this book, there is a much better way to handle similar scenarios.

**The most widely used reactive program**

Both examples in the previous section will feel familiar to some readers. If we call the input text fields "cells" and the result label's handler a "formula", we now have the nomenclature used in modern spreadsheet applications such as Microsoft Excel.

The term Reactive Programming has only been in use in recent years, but the idea of a reactive application isn't new. The first electronic spreadsheet dates back to 1969 when Rene Pardo and Remy Landau, then recent graduates from Harvard University, created LANPAR (LANguage for Programming Arrays at Random) [1].
What is Reactive Programming?

It was invented to solve a problem that Bell Canada and AT&T had at the time: their budgeting forms had 2000 cells that, when modified, forced a software re-write taking anywhere from six months to two years.

To this day, electronic spreadsheets remain a powerful and useful tool for professionals of various fields.

The Observer design pattern

Another similarity the keen reader may have noticed is with the Observer design pattern. It is mainly used in object-oriented applications as a way for objects to communicate with each other without having any knowledge of who depends on its changes.

In Clojure, a simple version of the Observer pattern can be implemented using *watches*:

```clojure
(def numbers (atom []))

(defn adder [key ref old-state new-state]
  (print "Current sum is " (reduce + new-state)))

(add-watch numbers :adder adder)
```

We start by creating our program state, in this case an atom holding an empty vector. Next, we create a watch function that knows how to sum all numbers in `numbers`. Finally, we add our watch function to the numbers atom under the `:adder` key (useful for removing watches).

The `adder` key conforms with the API contract required by `add-watch` and receives four arguments. In this example, we only care about `new-state`.

Now, whenever we update the value of `numbers`, its watch will be executed, as demonstrated in the following:

```clojure
(swap! numbers conj 1)
;; Current sum is 1

(swap! numbers conj 2)
;; Current sum is 3

(swap! numbers conj 7)
;; Current sum is 10
```

The highlighted lines above indicate the result that is printed on the screen each time we update the atom.
Though useful, the Observer pattern still requires some amount of work in setting up the dependencies and the required program state in addition to being hard to compose.

That being said, this pattern has been extended and is at the core of one of the Reactive Programming frameworks we will look at later in this book, Microsoft’s Reactive Extensions (Rx).

**Functional Reactive Programming**

Just like Reactive Programming, **Functional Reactive Programming — FRP** for short—has unfortunately become an overloaded term.

Frameworks such as RxJava (see [https://github.com/ReactiveX/RxJava](https://github.com/ReactiveX/RxJava)), ReactiveCocoa (see [https://github.com/ReactiveCocoa/ReactiveCocoa](https://github.com/ReactiveCocoa/ReactiveCocoa)), and Bacon.js (see [https://baconjs.github.io/](https://baconjs.github.io/)) became extremely popular in recent years and had positioned themselves incorrectly as FRP libraries. This led to the confusion surrounding the terminology.

As we will see, these frameworks do not implement FRP but rather are inspired by it.

In the interest of using the correct terminology as well as understanding what "inspired by FRP" means, we will have a brief look at the different formulations of FRP.

**Higher-order FRP**

Higher-order FRP refers to the original research on FRP developed by Conal Elliott and Paul Hudak in their paper *Functional Reactive Animation* [2] from 1997. This paper presents *Fran*, a domain-specific language embedded in Haskell for creating reactive animations. It has since been implemented in several languages as a library as well as purpose built reactive languages.

If you recall the calculator example we created a few pages ago, we can see how that style of Reactive Programming requires us to manage state explicitly by directly reading and writing from/to the input fields. As Clojure developers, we know that avoiding state and mutable data is a good principle to keep in mind when building software. This principle is at the core of Functional Programming:

```
(->> [1 2 3 4 5 6]
    (map inc)
    (filter even?)
    (reduce +))
;; 12
```
**What is Reactive Programming?**

This short program increments by one all elements in the original list, filters all even numbers, and adds them up using `reduce`.

Note how we didn't have to explicitly manage local state through at each step of the computation.

Differently from imperative programming, we focus on what we want to do, for example iteration, and not how we want it to be done, for example using a `for` loop. This is why the implementation matches our description of the program closely. This is known as declarative programming.

FRP brings the same philosophy to Reactive Programming. As the Haskell programming language wiki on the subject has wisely put it:

> FRP is about handling time-varying values like they were regular values.

Put another way, FRP is a declarative way of modeling systems that respond to input over time.

Both statements touch on the concept of time. We'll be exploring that in the next section, where we introduce the key abstractions provided by FRP: signals (or behaviors) and events.

## Signals and events

So far we have been dealing with the idea of programs that react to user input. This is of course only a small subset of reactive systems but is enough for the purposes of this discussion.

User input happens several times through the execution of a program: key presses, mouse drags, and clicks are but a few examples of how a user might interact with our system. All these interactions happen over a period of time. FRP recognizes that time is an important aspect of reactive programs and makes it a first-class citizen through its abstractions.

Both signals (also called behaviors) and events are related to time. Signals represent continuous, time-varying values. Events, on the other hand, represent discrete occurrences at a given point in time.

For example, time is itself a signal. It varies continuously and indefinitely. On the other hand, a key press by a user is an event, a discrete occurrence.

It is important to note, however, that the semantics of how a signal changes need not be continuous. Imagine a signal that represents the current (x,y) coordinates of your mouse pointer.
This signal is said to change discretely as it depends on the user moving the mouse pointer—an event—which isn't a continuous action.

**Implementation challenges**

Perhaps the most defining characteristic of classical FRP is the use of continuous time.

This means FRP assumes that signals are changing all the time, even if their value is still the same, leading to needless recomputation. For example, the mouse position signal will trigger updates to the application dependency graph—like the one we saw previously for the mean program—even when the mouse is stationary.

Another problem is that classical FRP is synchronous by default: events are processed in order, one at a time. Harmless at first, this can cause delays, which would render an application unresponsive should an event take substantially longer to process.

Paul Hudak and others furthered research on higher-order FRP [7] [8] to address these issues, but that came at the cost of expressivity.

The other formulations of FRP aim to overcome these implementation challenges.

Throughout the rest of the chapter, I’ll be using signals and behaviors interchangeably.

**First-order FRP**

The most well-known reactive language in this category is Elm (see [http://elm-lang.org/](http://elm-lang.org/)), an FRP language that compiles to JavaScript. It was created by Evan Czaplicki and presented in his paper *Elm: Concurrent FRP for Functional GUIs* [3].

Elm makes some significant changes to higher-order FRP.

It abandons the idea of continuous time and is entirely event-driven. As a result, it solves the problem of needless recomputation highlighted earlier. First-order FRP combines both behaviors and events into signals which, in contrast to higher-order FRP, are discrete.

Additionally, first-order FRP allows the programmer to specify when synchronous processing of events isn't necessary, preventing unnecessary processing delays.

Finally, Elm is a strict programming language—meaning arguments to functions are evaluated eagerly—and that is a conscious decision as it prevents space and time leaks, which are possible in a lazy language such as Haskell.
What is Reactive Programming?

In an FRP library such as Fran, implemented in a lazy language, memory usage can grow unwieldy as computations are deferred to the absolutely last possible moment, therefore causing a space leak. These larger computations, accumulated over time due to laziness, can then cause unexpected delays when finally executed, causing time leaks.

Asynchronous data flow

Asynchronous Data Flow generally refers to frameworks such as Reactive Extensions (Rx), ReactiveCocoa, and Bacon.js. It is called as such as it completely eliminates synchronous updates.

These frameworks introduce the concept of Observable Sequences [4], sometimes called Event Streams.

This formulation of FRP has the advantage of not being confined to functional languages. Therefore, even imperative languages like Java can take advantage of this style of programming.

Arguably, these frameworks were responsible for the confusion around FRP terminology. Conal Elliott at some point suggested the term CES (see https://twitter.com/conal/status/468875014461468677).

I have since adopted this terminology (see http://vimeo.com/100688924) as I believe it highlights two important factors:

- A fundamental difference between CES and FRP: CES is entirely event-driven
- CES is highly composable via combinators, taking inspiration from FRP

CES is the main focus of this book.

Arrowized FRP

This is the last formulation we will look at. Arrowized FRP [5] introduces two main differences over higher-order FRP: it uses signal functions instead of signals and is built on top of John Hughes' Arrow combinators [6].

It is mostly about a different way of structuring code and can be implemented as a library. As an example, Elm supports Arrowized FRP via its Automaton (see https://github.com/evancz/automaton) library.
Chapter 1

The first draft of this chapter grouped the different formulations of FRP under the broad categories of Continuous and Discrete FRP. Thanks to Evan Czaplicki’s excellent talk Controlling Time and Space: understanding the many formulations of FRP (see https://www.youtube.com/watch?v=Agu6jipRfYw), I was able to borrow the more specific categories used here. These come in handy when discussing the different approaches to FRP.

Applications of FRP

The different FRP formulations are being used today in several problem spaces by professionals and big organizations alike. Throughout this book, we’ll look at several examples of how CES can be applied. Some of these are interrelated as most modern programs have several cross-cutting concerns, but we will highlight two main areas.

Asynchronous programming and networking

GUIs are a great example of asynchronous programming. Once you open a web or a desktop application, it simply sits there, idle, waiting for user input.

This state is often called the event or main event loop. It is simply waiting for external stimuli, such as a key press, a mouse button click, new data from the network, or even a simple timer.

Each of these stimuli is associated with an event handler that gets called when one of these events happen, hence the asynchronous nature of GUI systems.

This is a style of programming we have been used to for many years, but as business and user needs grow, these applications grow in complexity as well, and better abstractions are needed to handle the dependencies between all the components of an application.

Another great example that deals with managing complexity around network traffic is Netflix, which uses CES to provide a reactive API to their backend services.

Complex GUIs and animations

Games are, perhaps, the best example of complex user interfaces as they have intricate requirements around user input and animations.

The Elm language we mentioned before is one of the most exciting efforts in building complex GUIs. Another example is Flapjax, also targeted at web applications, but is provided as a JavaScript library that can be integrated with existing JavaScript code bases.
Summary
Reactive Programming is all about building responsive applications. There are several ways in which we can make our applications reactive. Some are old ideas: dataflow programming, electronic spreadsheets, and the Observer pattern are all examples. But CES in particular has become popular in recent years.

CES aims to bring to Reactive Programming the declarative way of modeling problems that is at the core of Functional Programming. We should worry about what and not about how.

In next chapters, we will learn how we can apply CES to our own programs.
Where to buy this book

You can buy Clojure Reactive Programming from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.