# FRP Essentials \ part 1

If you have  this guide, you have probably already heard about reactive programming (or even functional reactive programming). Nowadays, a lot of developers claim to use it in their own project; it is also easy to find posts on the Internet saying how amazing it is when compared with older paradigms. But, know you are doing a great job with the tools you have in your hands.

Learning a new programming paradigm is really hard. Most of the time you don't even know you needed it before you already mastered it, and you always have to ask yourself if this is really something that is worth the hours of studying or if it is just a new buzzword that all the cool kids are talking about, without adding real value to your programming skills.

Maybe you have tried reactive programming before, even used it in a small project or, like me, you just thought it wasn't worth a try, but for some reason you decided to give the paradigm an opportunity, (you are always thirsty for knowledge, I know, I was in your shoes a couple of years ago). It took me a lot of time to understand why I needed reactive programming and how to master it. I want to make sure that you have an easier path than I had.

This guide will guide you through the reactive programming principles, and using a lot of examples, I will show you how to process and combine different sources of data or events to create astonishing live applications. In this first part, we are going to use the `bacon.js` library just to understand the basics and then we will go deeper with **Reactive Extensions** (**RxJS**).

Before we dive into programming, you need to understand what reactive programming is and the problems it is designed to solve.

This part will cover the following points:

- Understanding what reactive programming is
- Comparison between reactive programming and imperative programming
- Knowing what problems reactive programming solves
- Installation of the tools needed throughout this guide
- The first example of functional reactive programming will use a JavaScript library

# The reactive paradigm

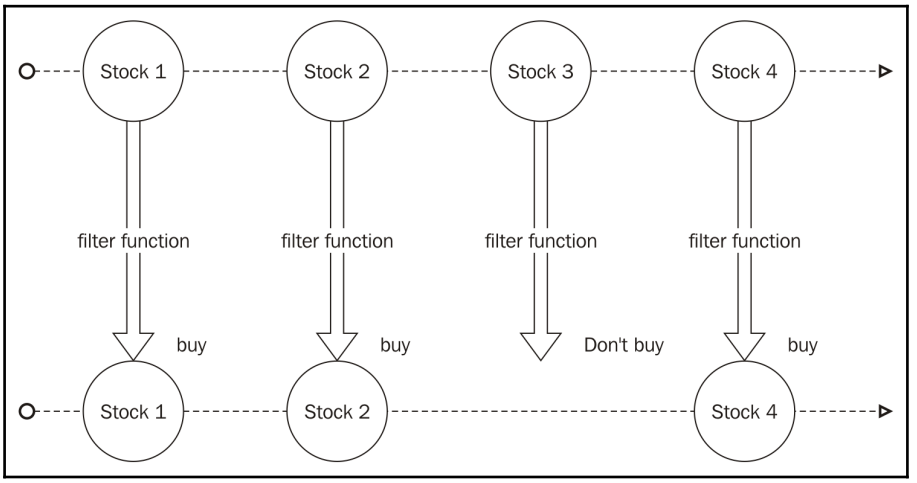
Reactive programming is a paradigm where the main focus is working with an asynchronous data flow. You may read many guides and see throughout the Internet that the reactive paradigm is about propagation of changes or some technical explanation that only makes it harder to understand.

Imperative programming makes you describe the steps a computer must do to execute a task. In comparison, functional reactive programming gives you the constructs to propagate the changes so you can focus on what to do instead of how to do it.

This can be illustrated in a simple sum of two numbers. In imperative programming $a = b + c$ is evaluated only in that line of code, so if you change the value of $b$ or $c$, it doesn't change the value of $a$. But in a reactive programming world, you can listen for the changes. Imagine the same sum in a Microsoft Excel spreadsheet, and every time you change the value of the column $b$ (or $c$), it recalculates the value of $a$, so you are always propagating the change for the ones interested in those changes.

The truth is that you probably already use an asynchronous data flow every time you add a listener to a mouse click or a keystroke in a web page you pass as an argument to a function to react to that user input. So, a mouse click can be seen as a stream of events that you can observe and execute a function on when it happens. But this is only one usage of event streams.

Reactive programming takes this to the next level–using it you can listen and react to changes in anything creating a stream of events from it, so you can react to changes in a variable or property, database, user inputs, external sources, and so on. For example, you can see the changes of value in a stock as an event stream and use it to show your user when to buy or sell in real time. Another common example for external sources streams is your Twitter feed or your Facebook timeline. Also, functional reactive programming gives you the possibility to filter, map, combine, buffer, and do a lot more with your streams of data or events. So using the stock example, you can easily listen to different stocks using a **filter function** to get the ones worth buying and show to the user a real time list of them, as shown in the following diagram:




# Why do I need it?

Functional reactive programming is especially useful when implementing one of these scenarios:

- Graphical user interface
- Animation
- Robotics
- Simulation
- Computer vision

A few years ago, all a user could do in a web app was fill a form with some data and post it to a server. Nowadays our web apps and mobile apps present to the user a richer interface, empowering them with real-time information and giving a lot more interaction possibilities. So, as the applications evolved, we needed more tools to achieve the new requirements.

Using it you can abstract the source of your data to the business logic of your application– this lets you write more concise and decoupled code, improves the reuse, and leads to a more testable code as you can easily mock your streams to test your business logic.

In this guide we will use Reactive Extensions to explain and implement an example reactive application. Reactive Extensions are widely used in the industry and they have implementations for different languages (.Net, Scala, JavaScript, Ruby, Java, and so on) so you can easily translate the things you learn in this guide to other languages.

In my personal opinion, Reactive Extensions have some concepts which are hard to understand for those unfamiliar with reactive programming. For this reason, we will learn the basics using a more simple library (`bacon.js`), and as soon as you understand the basics and the concepts, I will give you more tools using RxJS.

# Installation of tools

Before we start to use reactive programming, we need to install the tools we will be using throughout this guide.

# Node.js

We will be using node version 6.9.1, the most recent LTS version of nodes at the time of writing. You can find versions of it for Windows, Mac, and Linux at the following link:

`https://nodejs.org/en/download/releases/`.

# bacon.js

In this first part of this guide we will be using `bacon.js`, which is a library for functional reactive programming in JavaScript. This library works in both server and client.
We will use it to introduce you to some concepts of functional reactive programming as it is easier to get started. We will be using version 0.7.88.

To install it on your server, just run the following command inside a node project:

```
npm i baconjs@0.7.88 –save
```

To add it to an HTML page, just paste the following code snippet inside it:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/bacon.js/0.7.88/Bacon.min.js">
</script>
```

Don't worry with the version not being above 1.x; bacon.js is stable.

# RxJS

The last tool we need to follow in this guide is RxJS; we will use this library in later parts. This library also runs in both client and server and we will be using version 4.1.0.

To install it on your server, just run the following command inside a node project:

```
npm i rx@4.1.0 –save
```

To add it to an HTML page, just paste the following code snippet inside it:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/rxjs/4.1.0/rx.all.js">
</script>
```

For those using other package managers, you can also install `bacon.js` and RxJS from Bower and NuGet.

# Your first code using reactive programming

Now that you have installed all the tools we need to create our first program using functional reactive programming, we can start. For this first program we will be using bacon.js.

The bacon.js lets you work with events (which it calls **EventStream**) and dynamic values (which it calls **Property**). An EventStream represents a stream of events (or data), and it is an observable object where you can subscribe to be notified of new events on this stream. Bacon comes with a lot of built-in functions to create event streams from different sources such as button clicks, key strokes, interval, arrays, promises, and so on (and you can also create your own sources). A Property is an observable. Like an EventStream, the difference between both is that a Property has a current value. So every time you need to know of the changes and the current state of something, you will be using a Property; if you just want to be notified of the events and you don't need to know the current state of something, then you use an EventStream. A Property can be created from an EventStream using the `toProperty()` or `scan()` methods from the bacon.js API.

Like any other functional reactive programming, Bacon has a set of operators to let you work with your events, so you can map an event to something else, you can filter some events, you can buffer your events, you can merge different event sources, and a whole lot more.

We will be running our first example on Node.js, so let's get started:

1. Open your terminal and create a folder for this first project.
2. Now create a project:
    1. Navigate to the folder you have created.
    2. Type the following command:

```
npm init
```

3. Keep hitting *Enter* to accept the default configuration.
4. If you did it right, you should see a message like this printed in your console:

```
  {
"name": "example1",
"version": "1.0.0",
"description": "",
"main": "index.js",
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

```
      "author": "",
      "license": "ISC",
      "dependencies": {}
}
```

OK, now we have our test project. In this project we will do a timer implementation–basically, what we want is to print the current time every second. We will do it with and without the functional reactive programming library, so first let's implement it in the traditional way.

Create a file called `traditionalInterval.js` and paste the following code inside it, and save:

```
setInterval(
    ()=> console.log(new Date())
,1000);
```

This code uses the function `setInterval ()` to call our function every `1000` milliseconds (every second). Now, run this program using the following command in your terminal:

```
node ./traditionalInterval.js
```

If you did it right it will print the current date every second in your terminal, like this:

```
2016-10-30T20:28:22.778Z
2016-10-30T20:28:23.796Z
2016-10-30T20:28:24.803Z
2016-10-30T20:28:25.808Z
2016-10-30T20:28:26.809Z
2016-10-30T20:28:27.815Z
2016-10-30T20:28:28.820Z
```

To stop your program just hit *Ctrl + C*.

Now let's see how we can implement the same program using bacon.js for functional reactive programming. First, we need to install `bacon.js` on our project, as described in the *Installation of tools* section and also described here:

```
npm i baconjs@0.7.88 -save
```

With `bacon.js` installed in our project we are ready to implement our program. First create a file called `baconInterval.js` and then, as we will require the library in our project, open this file in a text editor and paste the following code:

```
var Bacon = require("baconjs");
```

Now that we have added Bacon to our code, we can use it to create a timer that prints the current time every second.

Before showing you the code that does this, it's important to understand how we model this problem using functional reactive programming. As we described before, bacon.js implements the concept of event streams, which are an observable stream of events. To create a timer, we need a special type of stream, capable of emitting events every *x* seconds, so we can listen to this stream to print the current date.

The bacon.js has a lot of built-in functions to create EventStreams from different sources. We will discuss the available functions later, but at this time we need to know the `Bacon.interval()` function that lets us create an EventStreams that emits an event every *x* seconds, where *x* is the time between the events in milliseconds. So, if we want to send an event every second, we must use it as follows:

```
Bacon.interval(1000);
```

This code creates an EventStream that emits an event every second. Now we need a way to be notified of the events in this stream so we can print the current date on the console. We can do this using the `onValue()` function–this function lets us subscribe to listen to events on this stream, and it receives the function to be executed as a parameter, so it lets us change our code to use it. The full code is as follows:

```
var Bacon = require("baconjs");
Bacon
  .interval(1000)
  .onValue(
      ()=> console.log(new Date())
);
```

If you run this code, you will see an output like this:

```
2016-10-30T20:28:22.778Z
2016-10-30T20:28:23.796Z
2016-10-30T20:28:24.803Z
2016-10-30T20:28:25.808Z
2016-10-30T20:28:26.809Z
2016-10-30T20:28:27.815Z
2016-10-30T20:28:28.820Z
```

You still have to hit *Ctrl + C* to stop your program.

Here we are creating an EventStream from an interval. bacon.js has a lot of built-in methods to create event streams; we will see these methods in more detail in part 2.

v

Using the `map ()` operator we can write a more descriptive code. This also enables us to decouple our code and test each function, instead of the whole code.

If you run this code, you will see the same kind of output as from the previous code.

In the first example (without bacon.js) we have a hard code to test, because all logic of the code is tied together. In the next example, we improved our code testability, as we detached the source of the events from the action, so we can test our `onValue()` function mocking our EventStream, but the logic of getting the current date is still tied to the action (printing on the console). The last example (using `map()`) is a lot more testable, as we can test every single piece of code separately.

Let's see the last version of our problem. Instead of printing the current date every second forever, we will print the current date only five times, every second. We can implement it without using `frp`; create a file called `traditionalInterval5times.js` and paste the following code:

```
var count = 0;

var intervalId = setInterval(()=>{
    console.log(new Date());
```

```
    count++;
    if(count===5){
        clearInterval(intervalId);
  }},1000);
```

Now our code has become a lot more complicated. To implement the new version of the proposed code we need an external counter (to make sure we run the code only five times), and we need a reference for the interval scheduler so we can stop it later. By looking at this code it is not easy to understand what it is trying to do, and as you probably already noted, it is really hard to test.

You might be wondering how we can change our last `frp` code to implement; it would be amazing if we had a way to only listen to five events on our Bacons EventStream, so I present to you the `take()` method. This method lets you listen to only a given number of events on an EventStream. So, changing our previous code to run only five times is really straightforward; create a file named `baconIntervalMap5times.js` and paste the following code:

```
var Bacon = require("baconjs");

Bacon
    .interval(1000)
    .take(5)
    .map(()=>new Date())
.onValue((currentDate)=>console.log(currentDate));
```

The `take()` operator lets us listen to only the first *x* events on the EventStream.

Our new code is that simple. We don't need to change the stream source, the mapping function, nor the subscription function; we just need to take only five events from the stream.

In this final example, we can see how functional reactive programming can improve our code readability. In the code using `setInterval()`, we had to add a counter variable to make sure we ran the log function the right number of times. We also had to remember to clear the interval so we didn't run this code forever. On the other hand, on the code using `frp` all we had to do was describe the transformations using `take()` to show how many items we wanted, `map()` to get the current date, and finally subscribed to react to the events of this stream.

If things are going too fast for you, don't worry–I just wanted to get your hands dirty with some `frp` code. In the following parts we will see these concepts in a lot more detail.

In this part, we learned what functional reactive programming is and how it compares with imperative programming. We also learned where it comes from and modeled and implemented our first functional reactive program using the `bacon.js` library. We compared it with a program with and without `frp` and saw how it can improve our code maintainability and testability by separating the source of the events from the actions caused by them.

In the next part, we will take a more in-depth look into the `bacon.js` library, and what we can do with event streams, and we will also start to model and implement more complex examples using new operators.