Homework 4: Reactive Library

[Dates](#h.9qxbstc65oxl)

[Overview](#h.p72iq1fm3xop)

[Before Starting](#h.ivlegzkzds80)

[Supplied Skeleton](#h.9d5bmh8apdc)

[Part 1: Reactive library (30%)](#h.16dik693to52)

[Question 1: Creating streams](#h.6mqrn3t3gjpa)

[Question 2: Transforming streams](#h.9picuaw17d3k)

[Question 3: Combining Streams](#h.1cr81cudv07w)

[Part 2: Using the library (35%)](#h.gm6w32ruc2dg)

[Question 1: A timer stream](#h.6rimj5e5fpcm)

[Question 2: An event stream](#h.wienyql12fkr)

[Question 3: Throttling](#h.5rc6tkcrmbq4)

[Question 4: Remote requests](#h.yvummt49yu5v)

[Question 5: Search suggestions](#h.6uqks74rj8pi)

[Part 3: Fire 911 call filterer (30%)](#h.15qiynbjp3p5)

[Question 1: Re-retrieving calls](#h.34zh2zyr1mu4)

[Question 2: Removing duplicate entries](#h.6q4t8ftpusx1)

[Question 3: Adding a search input](#h.p7cbcp2exg1k)

[Food for thought (5%)](#h.8kfjp2q5c6r6)

[Deliverables](#h.najhft3gxma4)

# Dates

Assigned: 11 May.

Due: 25 May, 11:59pm.

# Overview

This assignment will implement a reactive library much like Microsoft’s RX and then use that library to build some simple web-page interactions. The assignment proceeds in two parts: first, you implement the library; next, you develop your RX-based web page interactions. For the first part, you will be guided by tests. Since the reactive interactions involved in the second part are harder to test automatically, you will have to manually test your code.

**Yes, you can add comments to this document!** Like this. This is the best way to ask clarification questions and pose suggestions on how to improve the homework.

# Before Starting

We will be testing your homework projects using Chrome 49 (not Chromium). Since different versions of Chrome behave differently, make sure you are using the right version to avoid any compatibility issues.

# Supplied Skeleton

The webpage rx.html executes all provided test cases from test.js. Initially, most of them will fail; as you implement the reactive library, those tests will start to pass. We will also test your code or several hidden additional tests, so please make your implementation general-purpose (as opposed to working only on the provided tests).

Unlike in previous assignments, rx.html contains additional buttons and text after the tests. We recommend moving on to these sections once you have passed all the tests. These buttons and text are scaffolding for Part 2 of the assignment.

The supplied skeleton is minimal, containing just a definition of the Stream class for Part 1 and a run-on-load block for Part 2, plus some helper code for one of the questions in Part 2.

# Part 1: Reactive library (30%)

In this part you implement a library for manipulating Streams. Streams produce values of some type, and can be subscribed to in order to be notified when those values are produced. Since the only thing users can do to streams is to subscribe to them, a stream contains only a list of callbacks, which it calls every time it produces a new value.

## Question 1: Creating streams

Internally, a stream remembers the callbacks that it will invoke for every new element it produces. The stream produces values when its \_push method is called; callbacks can be added with subscribe. Implement subscribe and \_push. You should be able to pass the test named “Subscribe and \_push”, which runs this code:

var elts = [];  
var out = new Stream();  
out.subscribe(function(x) { elts.push(x) });  
out.\_push(1);  
out.\_push(2);  
out.\_push(3);  
chai.expect(elts).to.deep.equal([1, 2, 3]);

The callback function that we subscribe to out adds every element produced to the array elts. When the values 1, 2, and 3 are pushed, we expect them to be in elts. (The line beginning with “chai” is using the Chai framework, itself a small DSL for describing test results.)

Also implement a \_push\_many function, like \_push but taking an array of values and pushing each of them onto the stream.

## Question 2: Transforming streams

Implement four functions to transform streams. Each of these functions is a method of an arbitrary stream and returns a new stream.

* stream.first() — this stream should contain just one value, the first produced by stream
* stream.map(f) — this stream should contain the result of f applied to the values produced by stream
* stream.filter(f) — this stream should contain the values produced by stream for which f returns true
* stream.flatten() — assuming stream produces arrays, this stream should produce the elements of each array produced by stream

Each of these functions has its own test in the test suite.

You’ve by now implemented map and filter quite a few times! But maps and filters are near-universally-useful ways of transforming some data, so it’s no surprise that they keep coming up.

## Question 3: Combining Streams

The next three functions will combine streams. Each of these is a method of streams A and B and returns a new stream.

* A.join(B) — this stream should contain both values produced by A and those produced by B, interleaved as they arrive. If A produces x, and then B produces y, and then A produces z, then in this case A.join(B) should produce x, and then y, and then z.
* A.combine() — assuming A produces streams, this stream should produce any value produced by a stream produced by A. A.combine(), in other words is the join of all streams produced by A.
* A.zip(B, f) — this stream should contain the results of calling f on the latest values of A and B, whenever either of them produces a value; that is, if the last value A has produced is a and the last value B has produced is b, this stream should produce f(a, b)

Each of these functions has its own test in the test suite. combine will be especially useful for handling HTTP requests.

# Part 2: Using the library (35%)

Unfortunately, streams are not too useful when they must be manually pushed to! Instead, streams are best when used as the API to access the browser. In this part you’ll implement hooks to interact with the browser using your streams library.

## Question 1: A timer stream

Create a function Stream.timer(N) which produces the current time every N milliseconds. You can create a JavaScript Date object for the current time with new Date() and set a task that occurs every N milliseconds with setInterval(*callback*, N). (*Note*: Most browsers have problems with intervals that are smaller than approximately 16 milliseconds. You won’t need anything that low for this assignment, though.)

Inside the block in the assignment skeleton that reads “PART 2 INTERACTIONS HERE”, write code that causes the text “[make this a timer]” on the test page be replaced with the current time. You can change that text with this invocation:

$("#time").text(*text*);

(This line uses the jQuery library, itself a small DSL for manipulating web pages.)

## Question 2: An event stream

Create a function Stream.dom(element, eventname) which produces a JavaScript *event object* whenever an event of the named type is triggered on that element. We’re using jQuery throughout this assignment, so you can assume that element is a jQuery object and so you can find out when a certain event happens with the jQuery invocation:

element.on(eventname, *callback*);

Test your implementation by making the text “[count clicks]” count the number of times you click on the “Click me!” button.

Mozilla has [a list of the events](https://developer.mozilla.org/en-US/docs/Web/Events) available in browsers.

## Question 3: Throttling

Implement the function stream.throttle(N), which produces values from stream but never more often than once every N milliseconds. There are several ways to do this. You may want to use the setTimeout(N, *callback*) function, which calls the callback in N milliseconds. The same comment as above on milliseconds applies.

Test your implementation by changing the text “[mouse position here]” to give the coordinates of the mouse within the white square right below that text, without updating that text more often than once a second. (You can use any of the available coordinate systems; we used pageX/Y.)

## Question 4: Remote requests

Implement the function Stream.url(url), which makes a remote request to the given URL and produces the parsed JSON response when the remote server responds. To make a remote request in JavaScript, you have to use the following incantation:

$.get(*url*, *callback*, “json”);

The callback is called with the parsed JSON response.

To test your implementation, use the city of Seattle’s [near-real-time feed](https://data.seattle.gov/views/kzjm-xkqj/rows.json?accessType=WEBSITE&method=getByIds&asHashes=true&start=0&length=10&meta=false&$order=:id) of Fire 911 calls, whose URL is also defined in the FIRE911URL variable in the assignment skeleton. That URL returns a JSON response that is an array where each element is an object describing a single call. That object has several fields, among others id with a unique per-call identifier, updated\_at with the time of the event, 3479077 with the address of the event, and 3479078 with a description.

Typically, the feed receives a new event about every ten minutes during normal hours.

You should download the list, extract the addresses, and create a list containing the locations of the most recent events. You can add an event to this list with this line:

$("#fireevents").append($("<li></li>").text(*where*));

## Question 5: Search suggestions

Implement the function stream.latest(), which acts as a “ratchet” mechanism. Suppose stream produces values which are themselves streams. stream.latest() should only produce values from the latest stream produced by stream that has produced values.

We’ll use this to implement search suggestions. Wikipedia has [an API](https://en.wikipedia.org/w/api.php?action=opensearch&search=opensearch&namespace=0&limit=10) for getting search suggestions. This API is a bit complex to use, so we’ve wrapped it with the function WIKIPEDIAGET(*term*, *callback*). The callback receives an array of Wikipedia page titles suggested for the search term.

You should make it so that whenever the user types in the element $(“wikipediasearch”), they get search suggestions as list items in the list $(“wikipediasuggestions”).

Here’s why this is tricky: each of these calls is asynchronous, so you might get responses for previous search terms. You also want to avoid sending too many requests to the server. I wouldn’t send a request more often than once every 100ms.

# Part 3: Fire 911 call filterer (30%)

In this part you extend the 911 event log from the last assignment in a series of steps so that you have a filterable, real-time log of 911 events.

## Question 1: Re-retrieving calls

Modify your response to Part 2 Question 4 to retrieve the 911 events data every minute and add the results to the same list $(“fireevents”).

## Question 2: Removing duplicate entries

Implement a stream.unique(f) method that produces all values produced by stream for which no previous value has had the same output from f. That is, produces those values whose f-value is unique. You’ll have to store past values produced by stream to do this.

Use this method to only output 911 calls with unique IDs. This prevents the same entries from appearing every minute.

## Question 3: Adding a search input

Use the search box (refer to it by $(“#firesearch”)) to allow users to enter search terms to narrow the addresses they see. For example, if you live in SE seattle, you could enter “Se”.

HTML input boxes fire the input event when they change, and event.target.value can be called to extract the current value of the text entry box.

The search interface should work like so: new Fire 911 events should only appear if they contain the search term as a substring. When the search term is changed, all events that have occurred since the page was loaded and which match the new search term should be displayed.

If you need to clear out the HTML list of fire events, you can do that with $(“fireevents”).empty().

# Food for thought (5%)

Some of the methods you implemented for streams—map and filter, for example—are not that different from the sorts of queries you implemented in HW2 (“A Query Language”). There, your library built an Abstract Syntax Tree and directly manipulated it to make queries faster. Yet in this assignment, no explicit abstract syntax tree is built. What optimizations could this reactive library do if it had access to your reactive program as an AST?

Give an optimized variant for the query you used for Part 2 Question 5, and describe (as a comment in your code) a pattern you could apply to transform your original program to this optimized form.

(If you already wrote that query in what you think is the most optimized possible form, give a more readable version and describe transformations that could turn it into the optimized code.)

# Deliverables

Please create a new Bitbucket git repository, share it with the [csep590c\_sp16\_overlord](https://bitbucket.org/csep590c_sp16_overlord/) account (give us *write* permissions), and tell us its URL in the survey to be posted on Piazza. Your repository should be a clone of the [official starter kit](https://bitbucket.org/csep590c_sp16_overlord/csep590c-sp16/), with your additions.

**For all problems:** your code should extend rx.js, in hw4/.