Asynchronous JavaScript

Eman Abdel Ghaffar


Key Words
Events Steams Promises Functional
Observables Async/Await Generators
Rx Callbacks Pure Functions Composition

The problem

Challenges with **error-first-callbacks**

- Callback hells
- Spaghetti code
- Error handling is easy to miss
- Can't return values with the return statement, nor can use the throw keyword

node95.js

```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function () {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function () {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function () {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function () {
                                floppy.load('disk5', function (data5) {
                                    // if node.js would have existed in 1995
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});
```

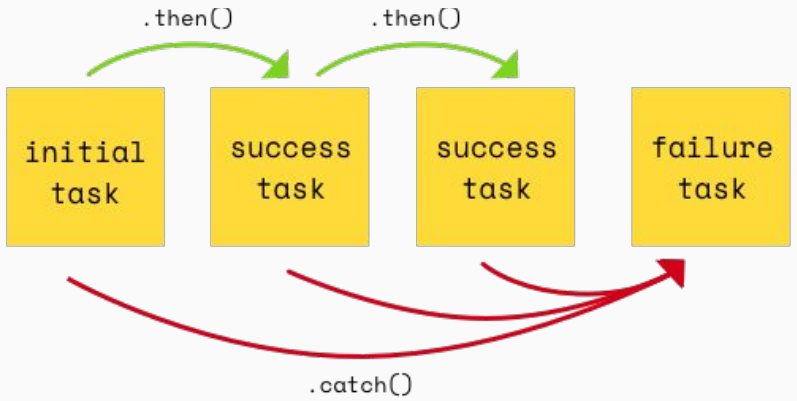
A small **callback hell** example

The problem

Promises act on data, and then return. They're sugar over the callback pattern

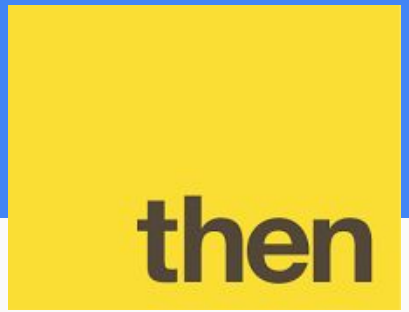
Challenges with **promise**

- Can't be cancelled
- Can't be retried
- Resolves to a single value asynchronously



Promises data flow example

Trying to Solve Async DataFlow



Promises

- Introduced in ES6
- A representation of a future value

Async/Await

- introduced in ES7
- gives a synchronous feel to asynchronous code.
- syntactical sugar.

Generators

could pause at any point, calculate something else, do other things, and then return to it

Reactive Programming

Rx offers a natural paradigm for dealing with sequences of events.

Functional Reactive Programming

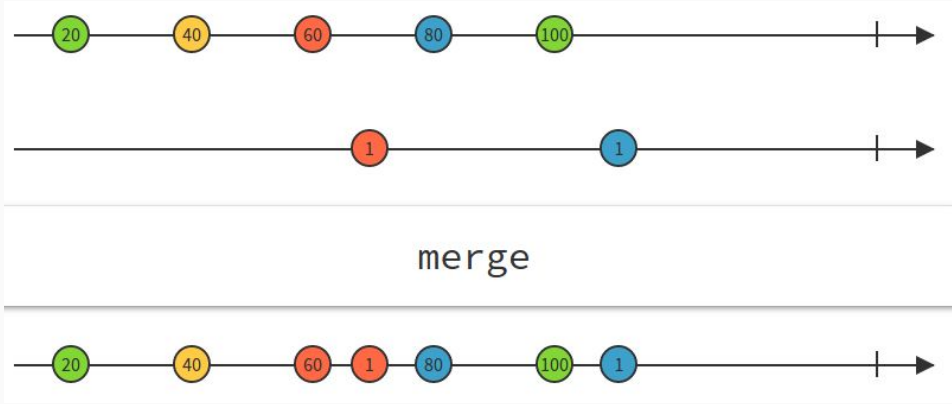
Functional Reactive Programming

FRP uses functional utilities like **map**, **filter**, and **reduce** to create and process **data flows** which propagate changes through the system: hence, reactive. When **input x changes, output y updates automatically in response**.

Functional Reactive Programming

Observable streams Vs Promise

A promise resolves to a single value asynchronously, an observable resolves to (or emits) multiple values asynchronously (over time)



Merging Two Observables

Cycle Js

- **Functional and Reactive**

Functional means “clean”, and Reactive means “separated”

- **Simple and Concise**

Functional reactive streams are able to build complex dataflows with a few operations. Apps in Cycle.js are small and readable

- **Extensible and Testable**

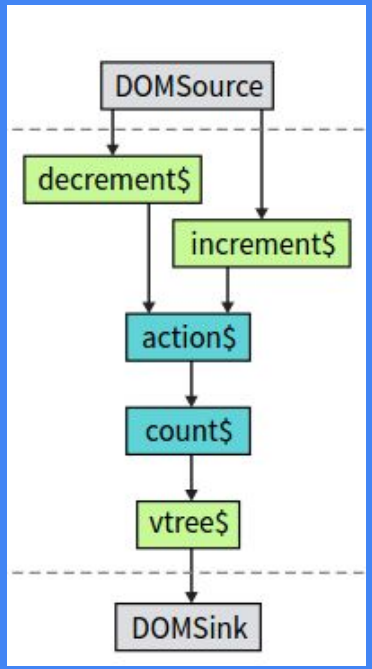
All side effects are contained in drivers. This means your application is just a pure function.

Cycle Js

- **Explicit dataflow**

In many frameworks the flow of data is implicit: you need to build a mental model of how data moves around in your app.

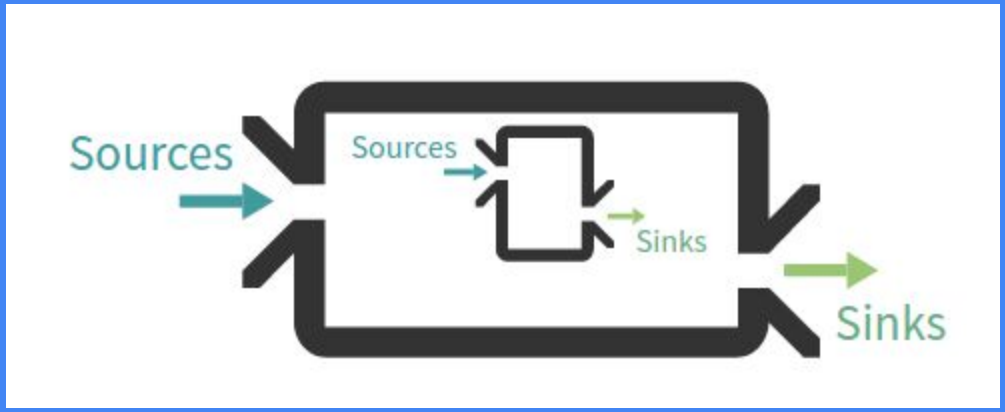
```
function main(sources) {
  const decrement$ = sources.DOM
    .select('.decrement').events('click').mapTo(-1);

  const increment$ = sources.DOM
    .select('.increment').events('click').mapTo(+1);

  const action$ = xs.merge(decrement$, increment$);
  const count$ = action$.fold((x, y) => x + y, 0);

  const vtree$ = count$.map(count =>
    div([
      button('.decrement', 'Decrement'),
      button('.increment', 'Increment'),
      p('Counter: ' + count)
    ])
  );
  return { DOM: vtree$ };
}
```



Cycle Js

- **Composable**

Every single Cycle.js app, no matter how complex, is a function that can be reused in a larger Cycle.js app



References

- The-two-pillars-of-javascript-pt-2-functional-programming
- https://cycle.js.org/
- Plug-and-play-all-your-observable-streams-with-cycle-js
- http://moduscreate.com/observables-and-promises/
- http://rxmarbles.com/
- Node-hero-async-programming-in-node-js
- rxjs-rxjs-observables-vs-promises