Reactive Programming for Couch Potatoes

(Nothing against couch potatoes)

Hi, I'm Andrew.

Friendly neighborhood programmer at Carbon Five.

Before we begin: follow along at home!

https://github.com/andrewhao/frp-for-couch-potatoes-talk

I've been hearing lots about FRP.

I've been hearing lots about FRP.

"Reactive programming is programming with asynchronous data streams."

I've been hearing lots about FRP.

"Reactive programming is programming with asynchronous data streams."

But... what... how... so confused!

Today's talk

- Intro to FRP
- Building blocks
- RxJS
- Build an example project
- Who else uses FRP?

We will be looking at FRP through a Javascript lens.

We will be looking at FRP through a Javascript lens.

- It's popular.

We will be looking at FRP through a Javascript lens.

- It's popular.
- You get it.

We will be looking at FRP through a Javascript lens.

- It's popular.
- You get it.
- It's frontend and backend.

We will be looking at FRP through a Javascript lens.

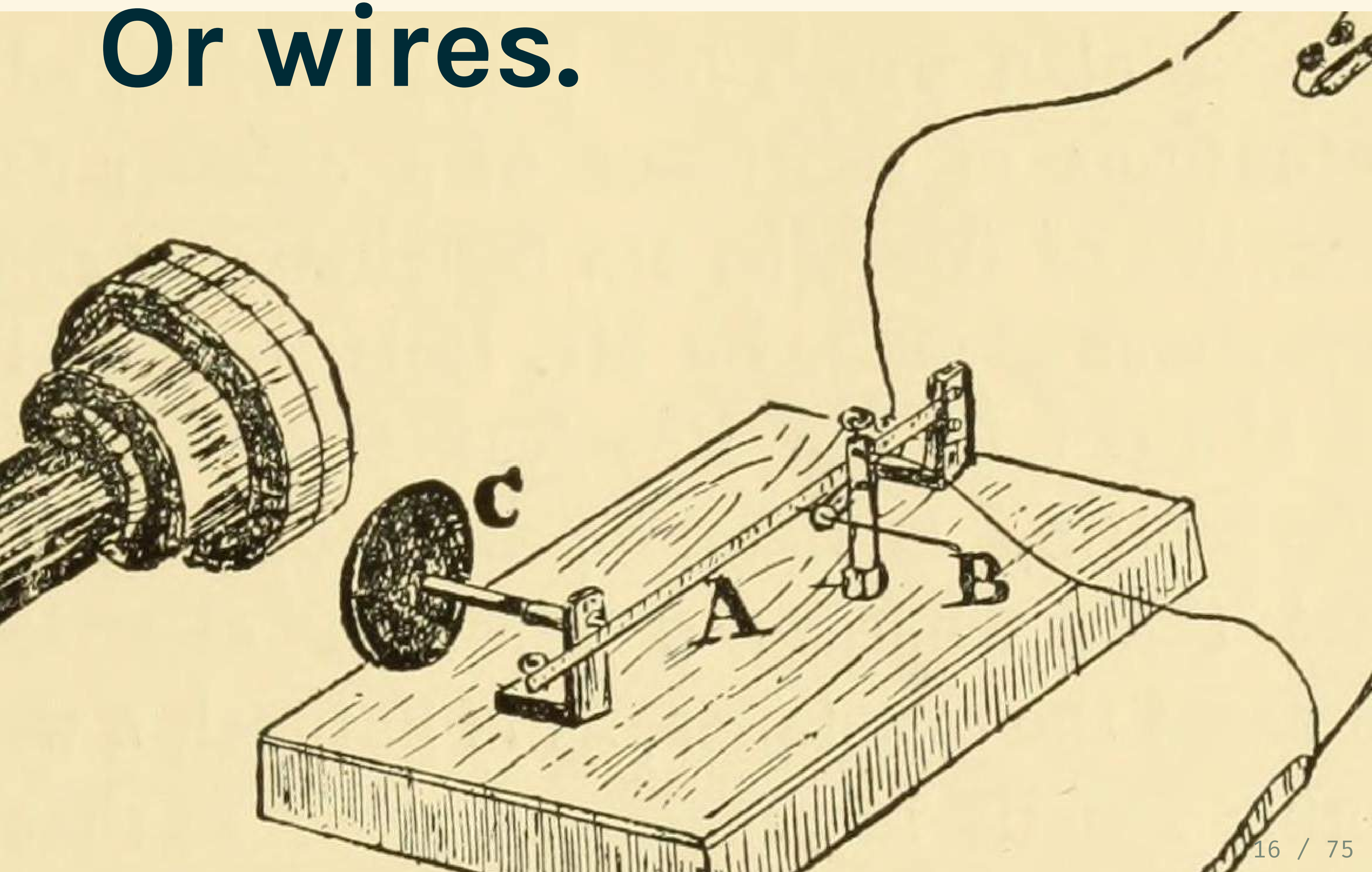
- It's popular.
- You get it.
- It's frontend and backend.
- It's not perfect.

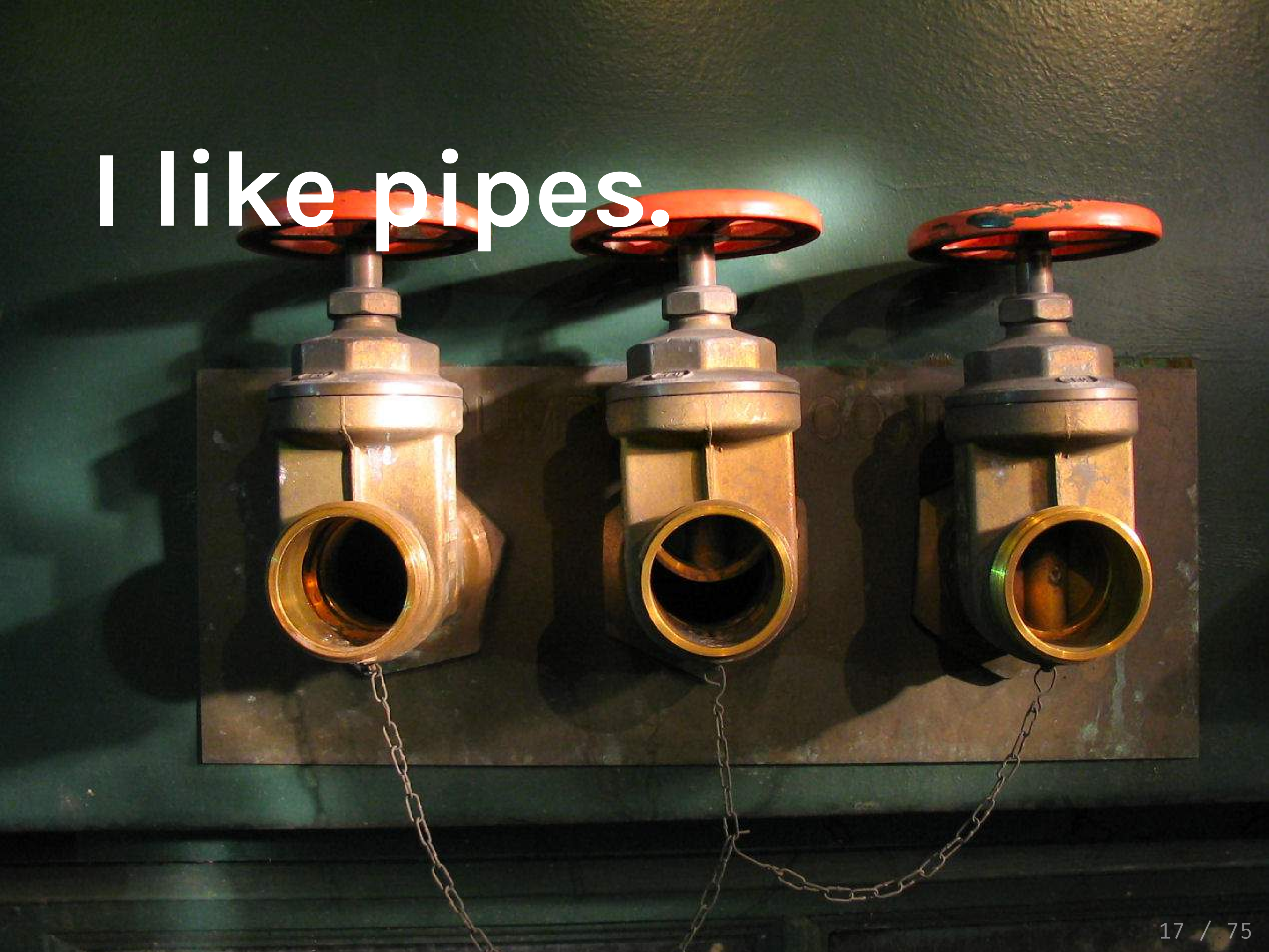
Streams are your friends.

Streams are like pipes.

Streams are like pipes.

Or wires.





I like pipes.

And you're familiar with them.

Streams are in:

- Unix pipes | > ..
- Twitter stream API
- WebSockets
- Node `streams` library.

Helpful (?) analogy

Streams are changing, asynchronous arrays.

```
let appleStockPrices = [1, 5, 10, 11, 22]
// [1, 5, 10, 11, 22, 44]
// [1, 5, 10, 11, 22, 44, 100]
```

If you can build it with Events, you can model it as a stream.

Anything you've been doing with Events can also be modeled as a stream.

```
$(window).on('mousemove', function(e) {
  console.log(e.target.x, e.target.y);
  // do something with this new movement
});
```

Voila - streams!

What if you thought about this like a pipe of mousemoves... with an unknown number of future values?

```
mouseMoveStream = [{x: 0, y: 0},
                   {x: 1, y: 122},
                   {x: 155, y: 23},
                   THE FUTURE...]
```

"Reactive programming is programming with asynchronous data streams."

Other things that are streamable

In fact, lots of other things in your software that, previously modeled as events (either explicitly, or as domain events), could be considered streams:

- `submittedForm` Event -> `formSubmitStream`
- `publishedNewsFeed` Event -> `newsFeedPublishStream`
- Any Node `EventEmitter` is easily converted to a stream.

Name change!

Streams are, more generically, `Observables`.

There are things already called Observables out there!

- RxJS has an Observable interface.
- Anything in the ReactiveX family implements an Observable interface.
- Bacon.js and Kefir.js also implement Observables.
- ES7 ships with Observables (it's the FUTURE).

F is for Functional.

Let's recap: a function is a relationship between a set of inputs and a set of outputs.

```
let f(x) = x + 4

x: 1 y: 5
x: 2 y: 6
x: 3 y: 7
```

F is for Functional.

```
let f(x) = x < 10

x: 1  y: true
x: 11 y: false
```

The F in FRP relates to the fact that we are going to be using functions to transform data and connect transformed inputs and outputs.

Your function(al) toolbelt

- `map()`
- `filter()`
- `reduce()` / `scan()`

Your function(al) toolbelt

- `map()`
- `filter()`
- `reduce()` / `scan()`

And the more advanced stuff:

- `combineLatest()`
- `zip()`

map()

```
var data = [1, 2, -100];
data.map(function(datum) { return datum > 0 });
// => [true, true, false]
```

reduce()

```
var data = [1, 2, -100];
data.reduce(function(previousValue, currentValue)
  return previousValue + currentValue;
}, 0)

// => -97
```

filter()

```
var data = [1, 2, -100];
data.filter(function(v) { return v > 0; });

// => [1, 2]
```

Now let's think of these functions in the context of streams.

Streamify all the things.

Streamify it!

```
let dataStream = getAsyncDataStream()
let outputStream = dataStream.map((datum) => datum
```

Now, when a value pops into existence on the `dataStream`, the `outputStream` will also produce a new value:

```
  data: ---[1]--[2]--[-100]-->
output: ---[t]--[t]------[f]-->
```

Whoa, what was that?

```
  data: --[1]--[2]--[-100]-->
output: --[t]--[t]------[f]-->
```

Marble diagrams are a thing: RxMarbles

Marbles, cont'd

Imagine this sort of thing could also happen with filter:

```
let outputStream = dataStream.filter((v) => v > 0)

//   data: |-- [1] -- [2] -- [-100] -->
// output: |-- [1] -- [2] ------------>
```

Magic.

scan()

Like reduce; it emits the intermediate accumulated value when a new value shows up.

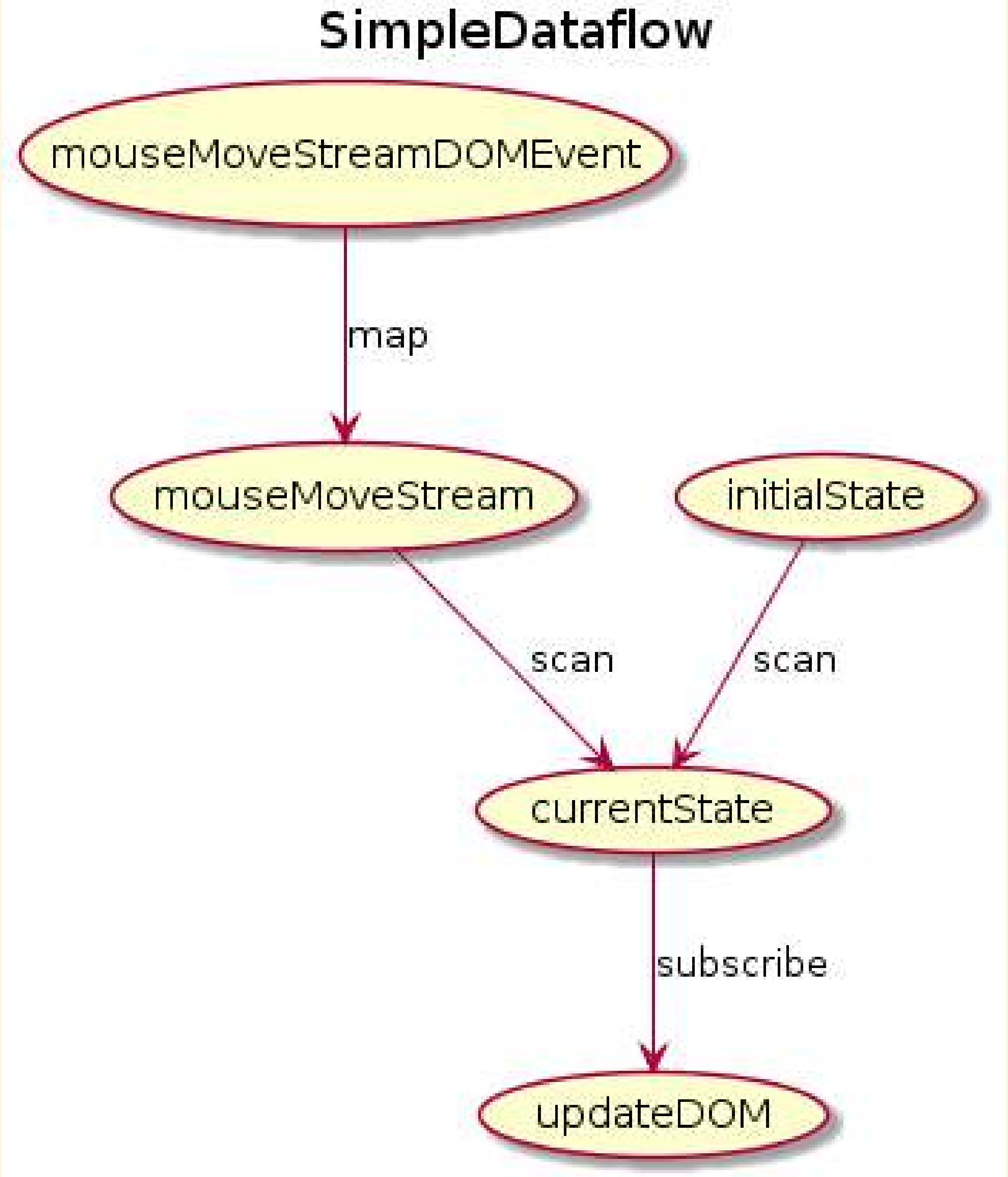
```
let outputStream = dataStream.scan((previousValue,
                                    currentValue) => {
  return previousValue + currentValue;
}, 0)

data   |------ [1] -- [2] -- [-100] -->
output |[0] -- [1] -- [3] --- [-97] -->
```

Let's enter the world of FRP

1. **Transform inputs** with `map()`
2. **Recompute state** with `scan()`
3. **Update outputs** with `map()` and `filter()`

Dataflow

FRP gives us the ability to model our system at a higher level of abstraction that is oriented around the flow of data through our system.


SimpleDataflow
mouseMoveStreamDOMEvent
map
mouseMoveStream
initialState
scan
scan
currentState
subscribe
updateDOM

Make a thing that tracks whether the mouse cursor is moving up or down.

```
var lastMovedCoordinate = {x: 0, y: 0};
$(window).on('mousemove', e => {
  let direction = (e.pageY < lastMovedCoordinate) ?
                  'up' : 'down'
  $('.output-container').html(`Moving ${direction}!`);
  lastMovedCoordinate = {x: e.pageX, y: e.pageY};
});
```

Make a thing that tracks whether the mouse cursor is moving up or down.

```
var lastMovedCoordinate = {x: 0, y: 0};
$(window).on('mousemove', e => {
  let direction = (e.pageY < lastMovedCoordinate) ?
                  'up' : 'down'
  $('.output-container').html(`Moving ${direction}!`);
  lastMovedCoordinate = {x: e.pageX, y: e.pageY};
});
```

- But state is stored in an arbitrary context.
- Callback code manages several responsibilities.

Build it again the FRP Way (tm)

1. Transform inputs with `map()`

Ask yourself: What are the inputs into the app?

```
let mouseMoveStream =
  Rx.Observable.fromEvent(window,
                          'mousemove')
// mouseMoveStream: --[e]--[e]--[e]--[e]-->
```

Transform inputs, cont'd

We should transform the data into the input format we care about. Here's where `map()` comes into play:

```
let mouseMoveStream =
  Rx.Observable.fromEvent(window, 'mousemove')
  .map((e) => { {x: e.pageX, y: e.pageY} }

// mouseMoveStream: --[{x:,y:}]--[{x:,y:}]--[{x:,y
```

Transform inputs, cont'd

We should transform the data into the input format we care about. Here's where `map()` comes into play:

```
let mouseMoveStream =
  Rx.Observable.fromEvent(window, 'mousemove')
  .map((e) => { {x: e.pageX, y: e.pageY} }

// mouseMoveStream: --[{x:,y:}]--[{x:,y:}]--[{x:,y
```

Now this stream is expressive in terms of the domain.

2. Recompute application state with `scan()`.

Ask yourself: What is the minimum amount of state that my app needs to store?

- A: Anything that the UI is dependent upon
- A: Anything that stores a value that is necessary for future events to compute from.

Application state for this app:

- To track the last event that came through, so we can compare our current coordinate and determine whether it moved up or down.
- To track the current computed directional state of the cursor.

A data structure for state

First we come up with an initial state:

```
let initialState = {lastCoordinate: {x: 0, y: 0},
                    direction: null};
```

Note how it is a simple data structure.

Next we update the application state based on the current (incoming) event.

```
let currentState = mouseMoveStream.scan(
  (oldState, newCoordinate) => {
  let newDirection =
    oldState.lastCoordinate.y < newCoordinate.y ?
'up';
  return {lastCoordinate: newCoordinate,
          direction: newDirection};
}, initialState)

// mouseMoveStream: -----[A]--[B]--[C]--[D]-->
//    currentState: [-]--[u]--[u]--[d]--[u]-->
```

3. Update outputs (UI) upon state change.

In RxJS, side effects such as modifying the DOM are performed in `subscribe` blocks.

```
currentState.subscribe(newState => {
  $('.output').text(
    `Mouse direction is: ${newState.direction}`
  );
});
```

3. Update outputs (UI) upon state change.

In RxJS, side effects such as modifying the DOM are performed in `subscribe` blocks.

```
currentState.subscribe(newState => {
  $('.output').text(
    `Mouse direction is: ${newState.direction}`
  );
});
```

Ours is a simple use case.

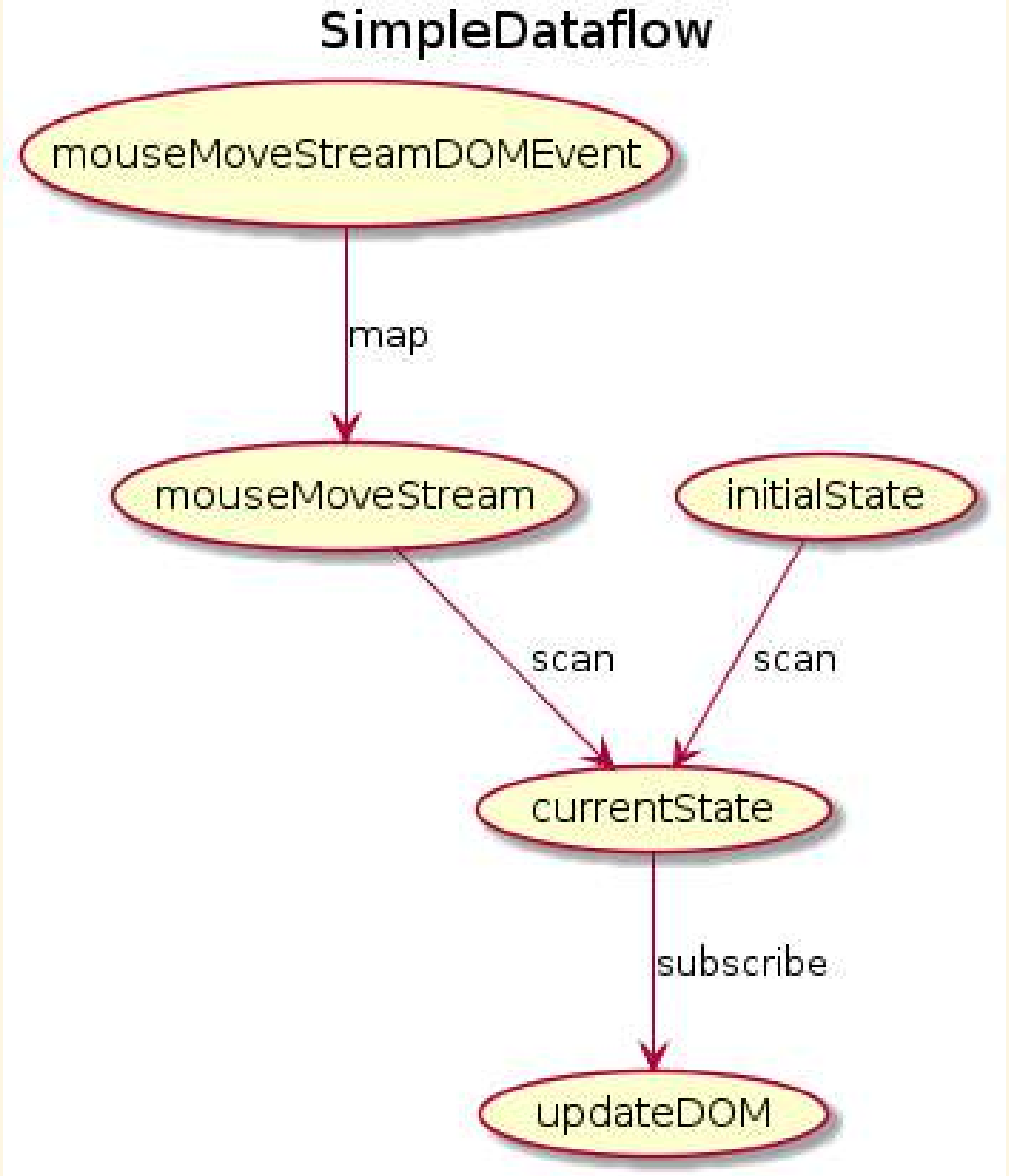
Extra RxJS bit: call **`subscribe`** to activate the stream and perform side effects.

```
currentState.subscribe(newState => {
  $('.raw-output').html(
    `<div>x: ${newState.pageX}, y:${newState.pageY
  );
});
```

Extra RxJS bit: call `subscribe` to activate the stream and perform side effects.

```
currentState.subscribe(newState => {
  $('.raw-output').html(
    `<div>x: ${newState.pageX}, y:${newState.pageY
  );
});
```

Streams are lazily evaluated.

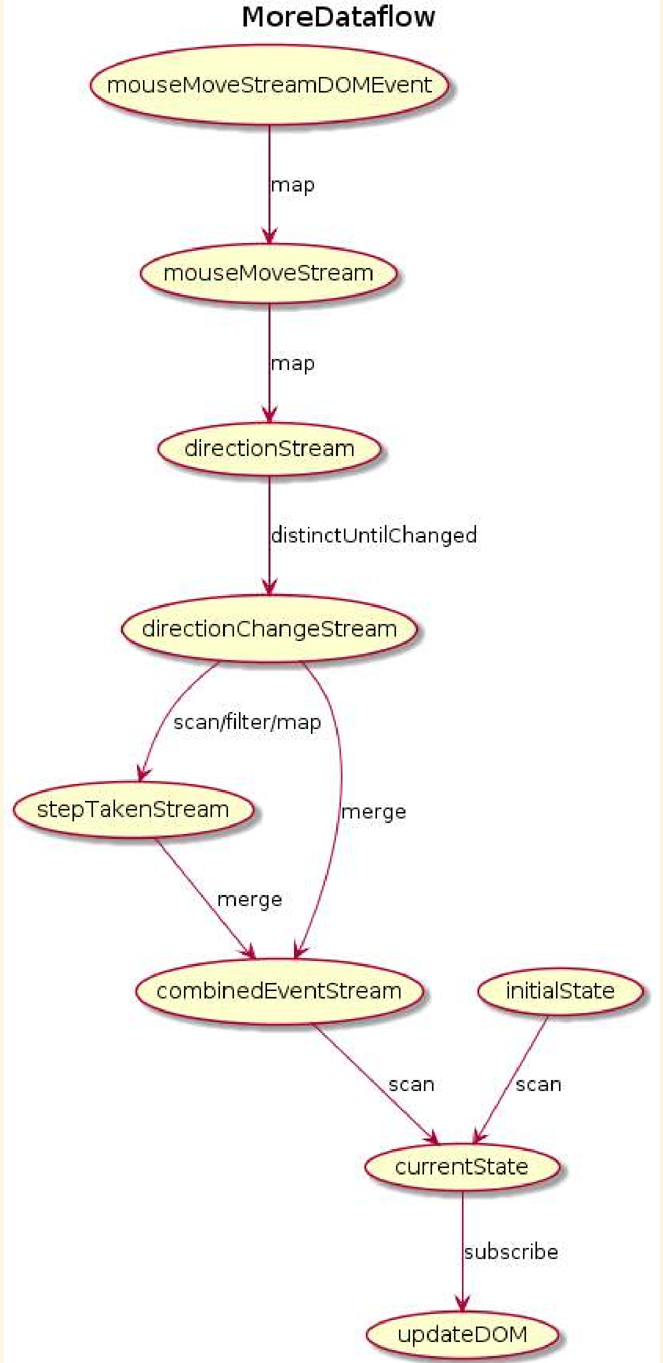

SimpleDataflow
mouseMoveStreamDOMEvent
map
mouseMoveStream
initialState
scan
scan
currentState
subscribe
updateDOM

Phew! Let's see it in action.

OK, let's make a pedometer!*

(* sorta)

Follow along in the source


MoreDataflow
mouseMoveStreamDOMEvent
map
mouseMoveStream
map
directionStream
distinctUntilChanged
directionChangeStream
scan/filter/map
stepTakenStream
merge
merge
combinedEventStream
initialState
scan
scan
currentState
subscribe
updateDOM

Error handling

FRP has special semantics around handling errors (in streams).

```
let someStream = doSomethingRx();
someStream.catch(e => {
  // handle error
})
```

This is important because we can reason about errors in an explicit manner.

FRP overall pattern

FRP Principle
Map inputs
Recompute state
Update output

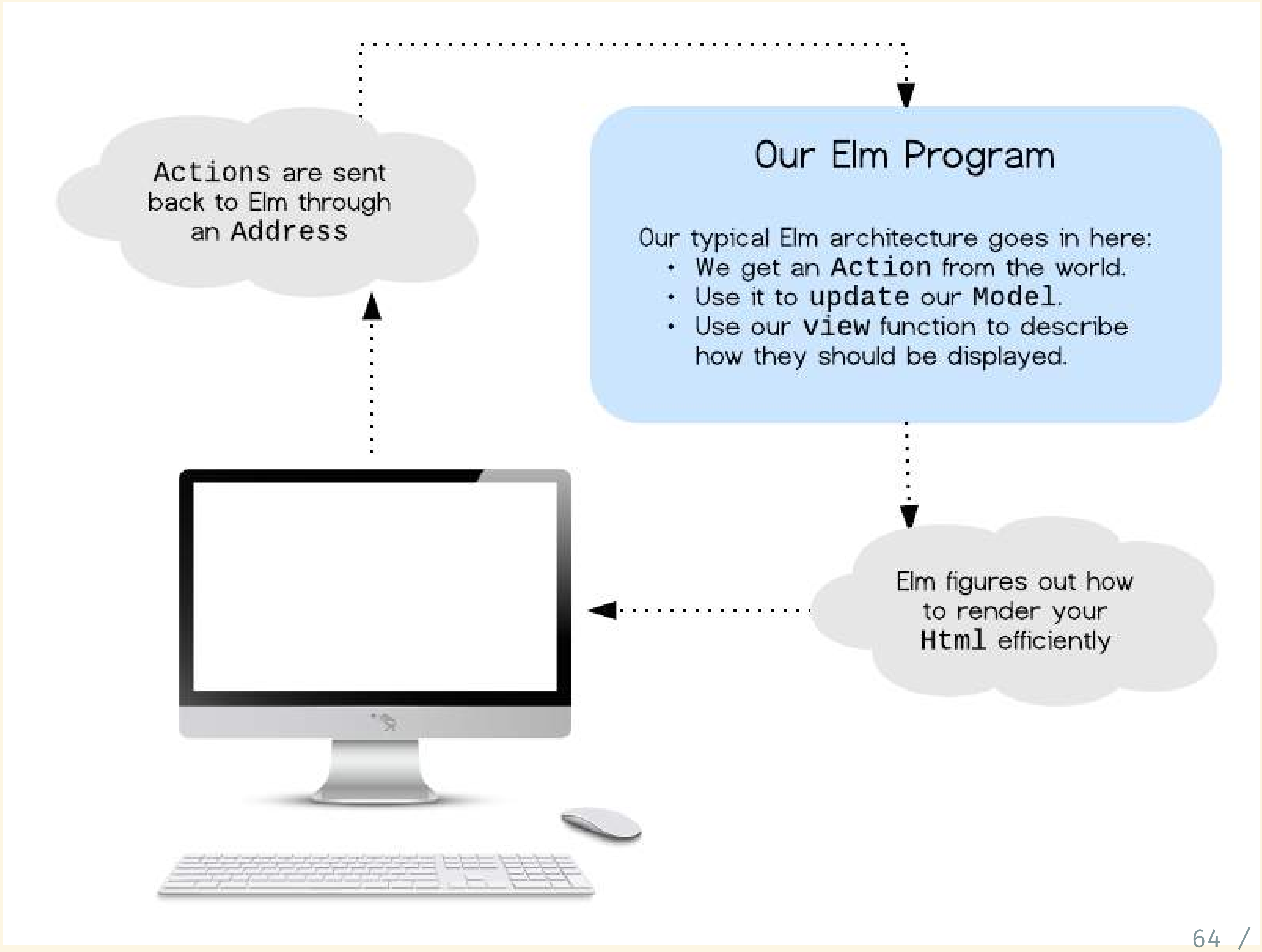
FRP overall pattern

FRP Principle
Map inputs
Recompute state
Update output

Watch as we apply these to other frameworks.

From Elm…

FRP Principle	Elm
Map inputs	`Action`
Recompute state	`Model`
Update output	`View`, `Effect`


Actions are sent back to Elm through an Address
Our Elm Program
Our typical Elm architecture goes in here:
• We get an Action from the world.
• Use it to update our Model.
• Use our view function to describe how they should be displayed.
Elm figures out how to render your Html efficiently

Example

The following chunk of code sets up a simple counter that you can increment and decrement. Notice that you focus entirely on setting up `model`, `view`, and `update`. That is it, no distractions!

```
import Html exposing (div, button, text)
import Html.Events exposing (onClick)
import StartApp.Simple as StartApp


main =
  StartApp.start { model = model, view = view, update = update }


model = 0


view address model =
  div []
    [ button [ onClick address Decrement ] [ text "-" ]
    , div [] [ text (toString model) ]
    , button [ onClick address Increment ] [ text "+" ]
    ]


type Action = Increment | Decrement


update action model =
  case action of
    Increment -> model + 1
    Decrement -> model - 1
```

Elm, cont'd

1. transform inputs to streams (`map`)
2. merge inputs into signal (`merge`)
3. update state of app architecture (`foldp`)
4. route values to appropriate service (`filter`)

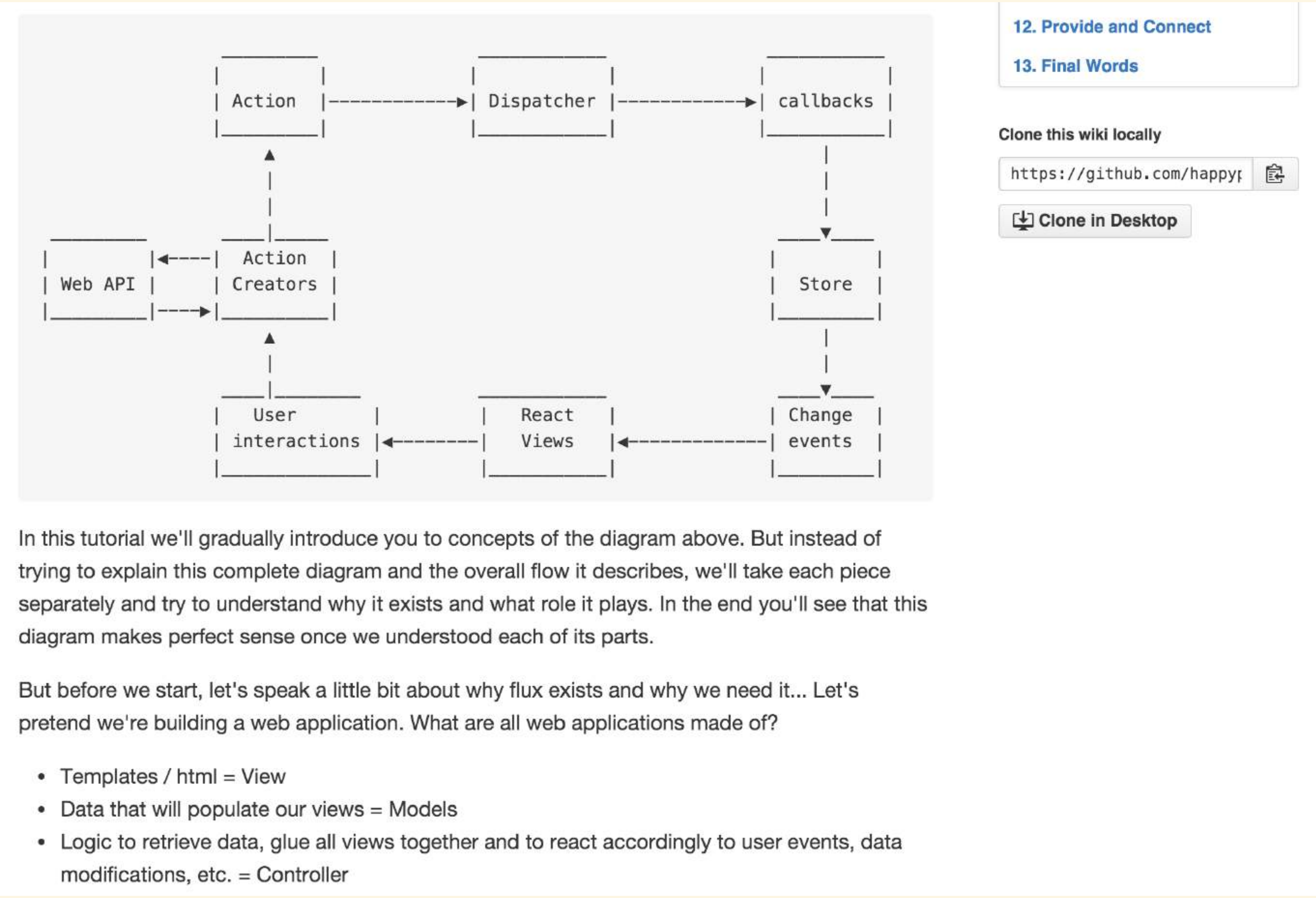
…to Redux

FRP Principle	Redux
Map inputs	`Action` is a Redux event
Recompute state	`Reducer` applied on a store
Update output	`React` to update UI

In Redux:

```
reducer = (state, action) => state
```

1. Actions are inputs
2. State is recomputed with reducers
3. Updates & side effects are created from state (React)



Clone this wiki locally

`https://github.com/happyp`

Clone in Desktop

```
                 _________            ____________            ___________
                |         |          |            |          |           |
                | Action  |--------->| Dispatcher |--------->| callbacks |
                |_________|          |____________|          |___________|
                     ^                                             |
                     |                                             |
                     |                                             |
 _________       ____|_____                                    ____v____
|         |<----|  Action  |                                  |         |
| Web API |     | Creators |                                  |  Store  |
|_________|---->|__________|                                  |_________|
                     ^                                             |
                     |                                             |
                 ____|________           ____________          ____v____
                |   User       |         |   React    |        | Change  |
                | interactions |<--------|   Views    |<-------| events  |
                |______________|         |____________|        |_________|
```

In this tutorial we'll gradually introduce you to concepts of the diagram above. But instead of trying to explain this complete diagram and the overall flow it describes, we'll take each piece separately and try to understand why it exists and what role it plays. In the end you'll see that this diagram makes perfect sense once we understood each of its parts.

But before we start, let's speak a little bit about why flux exists and why we need it... Let's pretend we're building a web application. What are all web applications made of?

- Templates / html = View
- Data that will populate our views = Models
- Logic to retrieve data, glue all views together and to react accordingly to user events, data modifications, etc. = Controller

FRP elsewhere

Netflix has some fantastic Reactive talks.

RxJS, Falcor, RxJava (Hystrix, circuit breakers)

Many other FRP frameworks/languages out there.

Cycle.js, Bacon.js, Highland.js, ClojureScript, Elm

Further reading (and many thanks!)

- "The Introduction to Reactive Programming You've Been Missing"
- "OMG Streams!"
- RxMarbles: http://rxmarbles.com/
- ReactiveX: http://reactivex.io/learnrx/

Recap

- Learn to see everything as a stream.
- Dataflow programming is powerful.
- FRP frameworks provide lots of tools. Use them!
- The pattern:
    - Map inputs
    - Recompute state
    - Update output

I made a pedometer, I swear.

https://github.com/andrewhao/quickcadence

Thanks!

https://www.github.com/andrewhao
https://www.twitter.com/andrewhao

Image attributions:

- https://www.flickr.com/photos/alphageek/21067
- https://www.flickr.com/photos/95744554@N00/
- https://www.flickr.com/photos/autowitch/42719
- https://www.flickr.com/photos/internetarchiveb