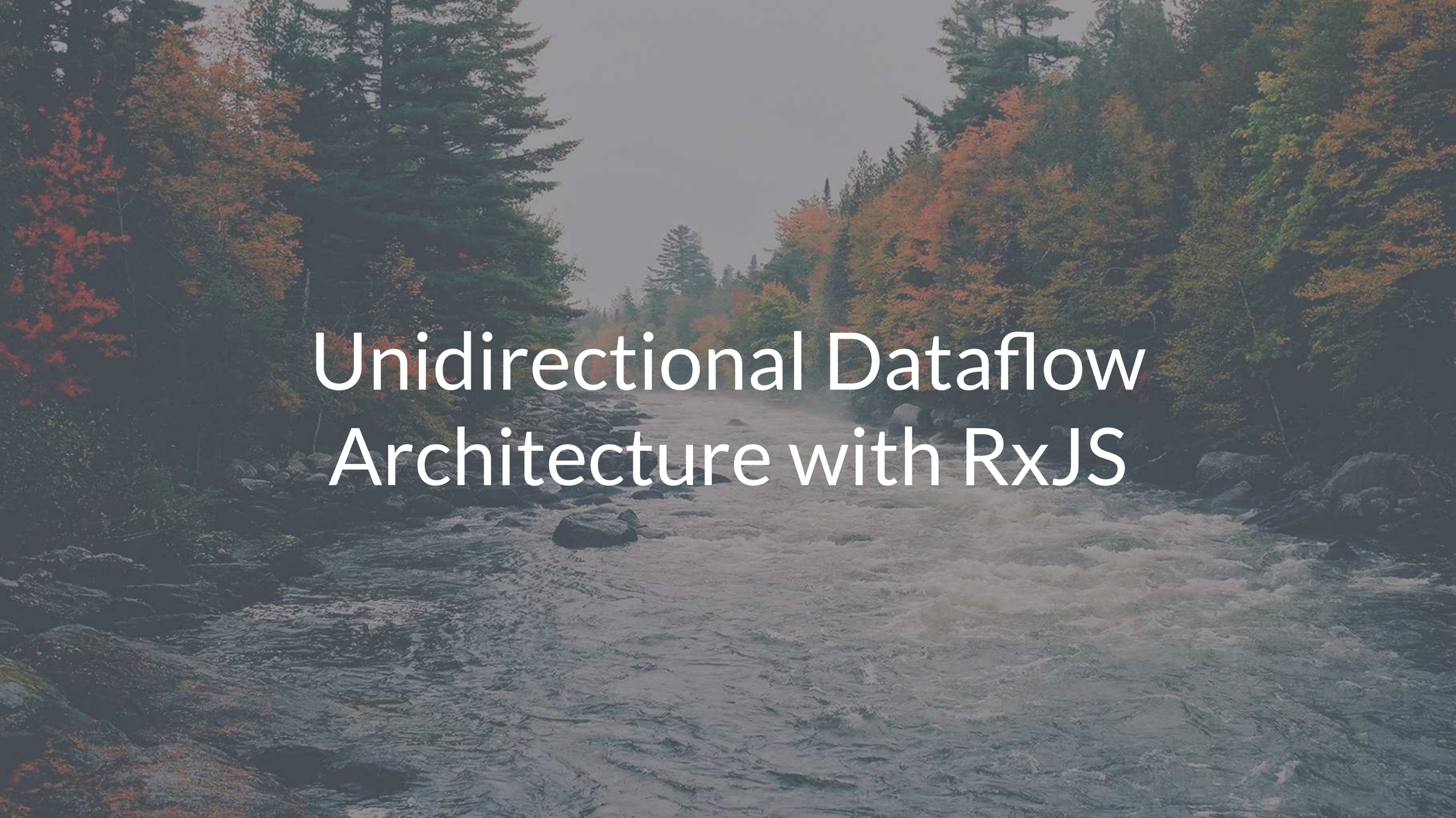

Unidirectional Dataflow
Architecture with RxJS

HI, I AM **KAHLIL LECHELT**

@kahliltweets
kahlil.info
👻 kahlilsnaps

Frontend Architect at **1&1**

Co-organizer of **KarlsruheJS** and **FrankfurtJS**

Host of podcasts: **Reactive**, **Simple**, **Descriptive**

Jake Archibald
@jaffathecake

Following

@getify @BenLesh cultish, and without a sense of irony

> **kahlil ツ** @kahliltweets
>
> Aah yeah! @jaffathecake and @getify are grumbly looking into Observables and tweeting the grumbles.
>
> Observables will be big this year. 😋

LIKES
2


10:26 AM - 14 Mar 2016

MINI-INTRO TO OBSERVABLES

WARNING

No exhaustive introduction possible during this presentation due to time-constraints.

If you need a thorough introduction into Observables read "**The introduction to Reactive Programming you've been missing**" and / or watch **"RxJS Beyond the Basics: Creating Observables from scratch"** from **André Staltz** on Egghead.io.

OH, MYSTERIOUS OBSERVABLES

Streams?

Hot? • Cold? • Unicast? • Multicast?

Lodash for Async?

Better Promises?


“
WE’RE GOING TO FIND OUT THAT A LOT OF UNICORNS ARE RHINOCEROSES.
@GARYVEE

Observable is just a function that **takes an observer** and **returns a function**.

— *Ben Lesh*

```
function myObservable(observer) {
    const datasource = new DataSource();

    datasource.ondata = (e) => observer.next(e);
    datasource.onerror = (err) => observer.error(err);
    datasource.oncomplete = () => observer.complete();

    return () => {
        datasource.destroy();
    };
}
```

ANY KIND OF DATASOURCE, SYNC OR ASYNC

Not helpful to compare with Promise.

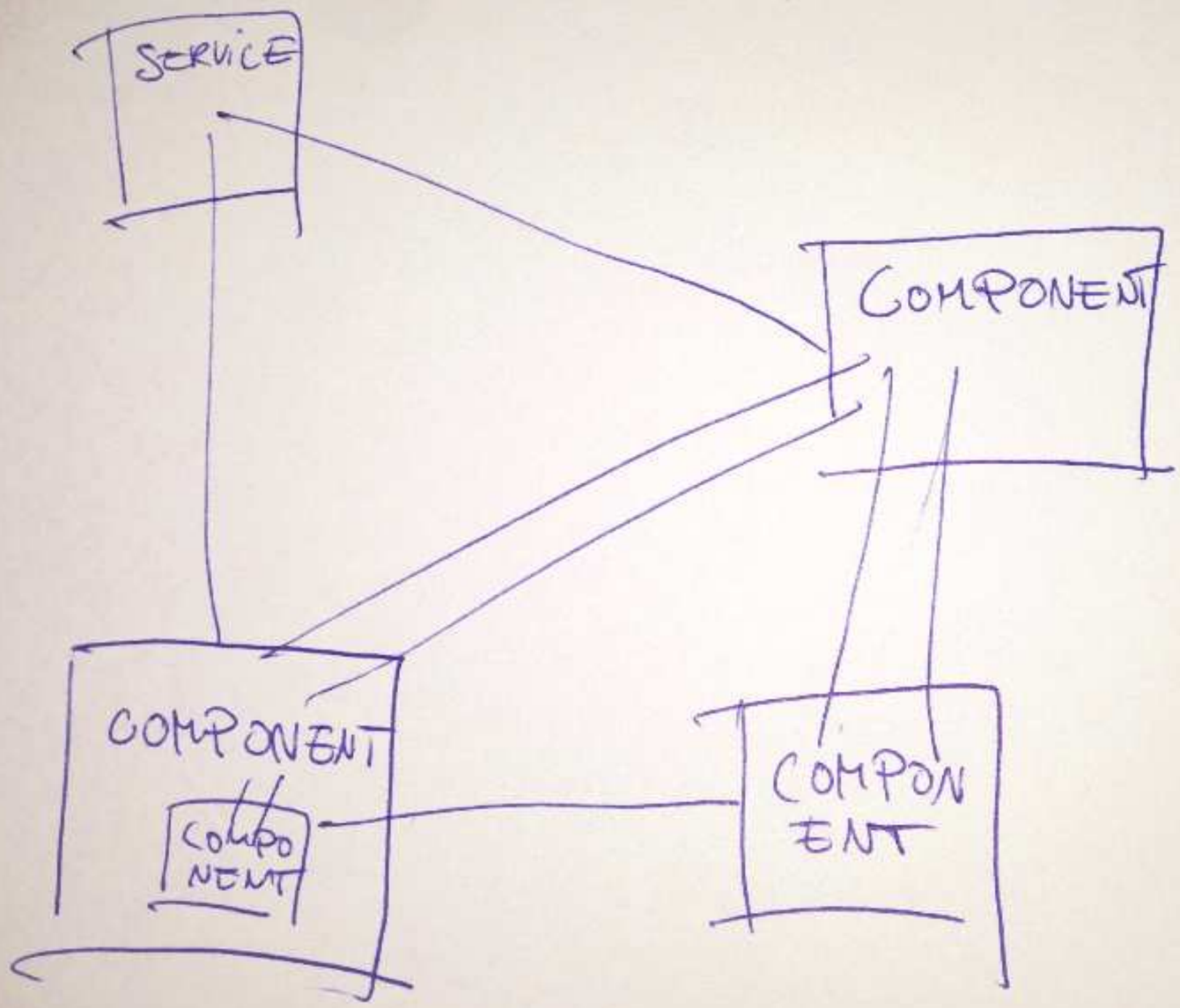
An Observable is just **a collection of items, over time**.

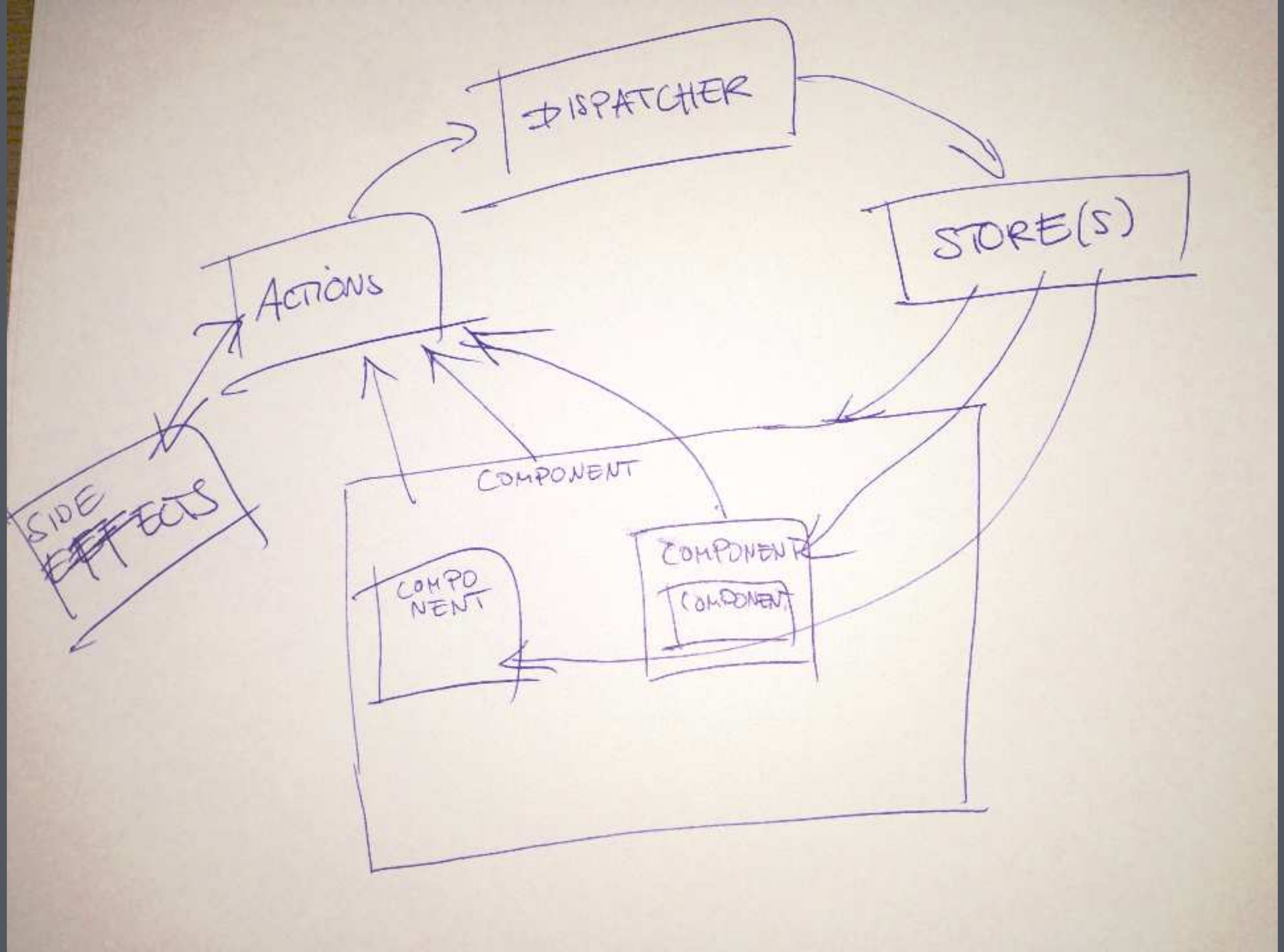
— *Jafar Husain*

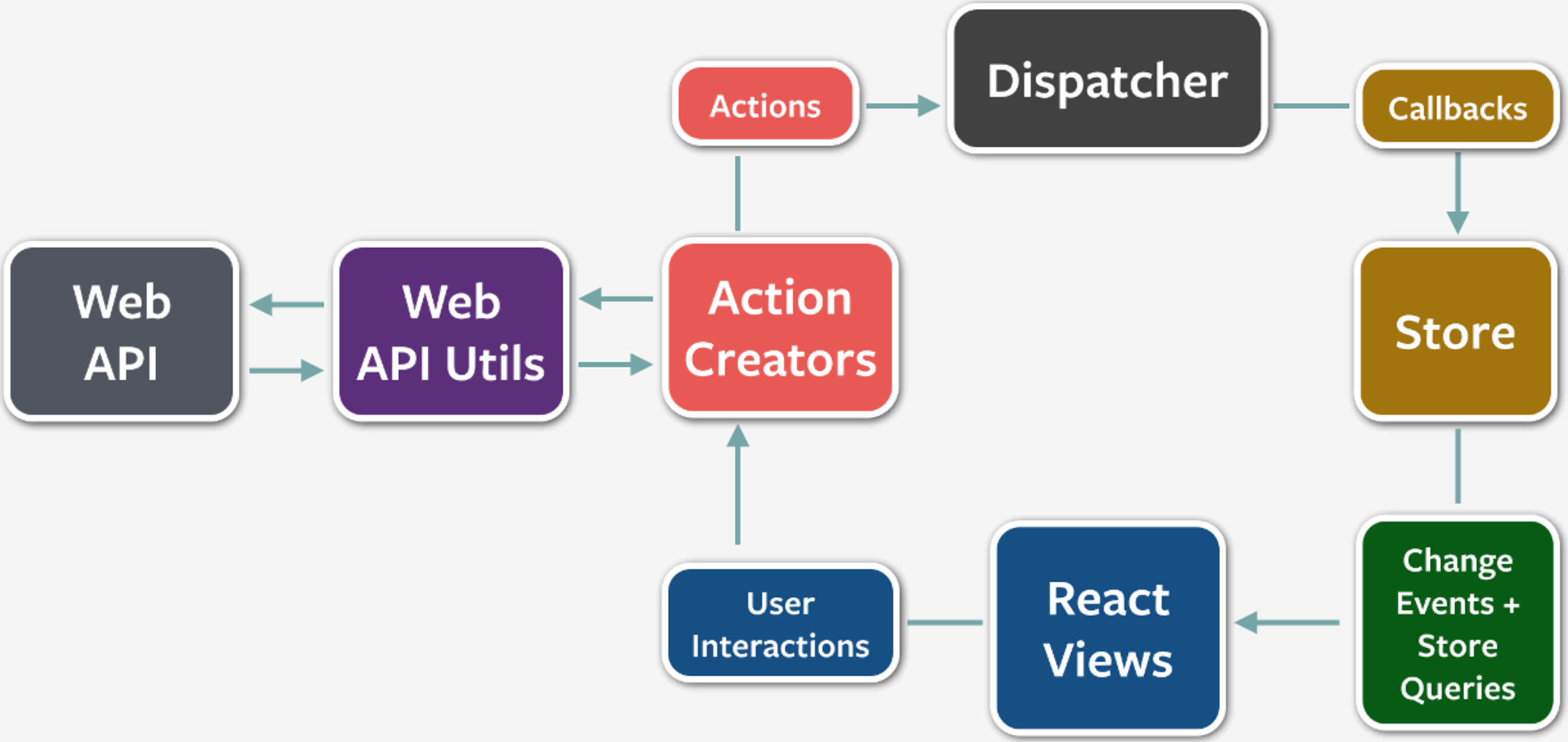
```
const stream = Rx.Observable.fromArray([1, 2, 3, 4, 5, 6]);

stream
    .map(x => x * 10)
    .filter(x => x > 30)
    .scan((acc, x) => acc + x, 0)
    .subscribe(
        x => console.log(x), // => 40, 90, 150
        err => console.error(error),
        () => console.log('completed')
    );
```

UNIDIRECTIONAL DATAFLOW

10 x BETTER THAN WHAT WE DID BEFORE


SERVICE
COMPONENT
COMPONENT
COMPO
NENT
COMPON
ENT

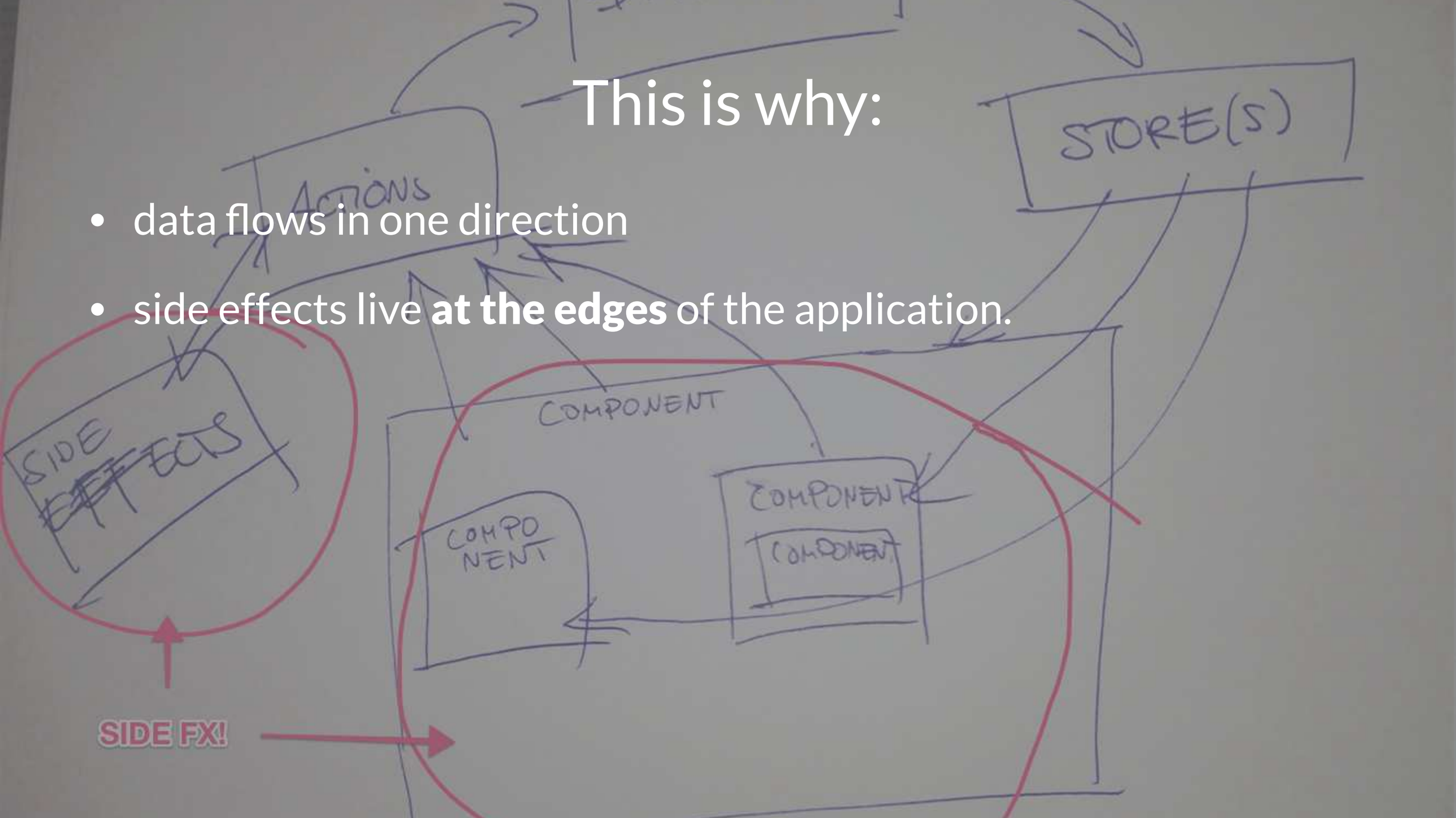

DISPATCHER
STORE(S)
ACTIONS
SIDE EFFECTS
COMPONENT
COMPO NENT
COMPONENT
COMPONENT


Actions
Dispatcher
Callbacks
Web API
Web API Utils
Action Creators
Store
User Interactions
React Views
Change Events + Store Queries

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

This is why:

- data flows in one direction
- side effects live **at the edges** of the application.



A Functional Programming Paradigm

- **Cycle.js** uses the **Cycle.js reducer pattern** and **drivers**
- **Elm** uses the **updater pattern** and **ports**
- **Redux** uses a **reducer pattern** and **middleware**

Why All This?

- keeps growing and big apps manageable
- devs stay in control of the code

DEVELOPER HAPPINESS

How it got started

Facebook **Flux** in early 2014.

and a pointer to some additional action-specific data (if applicable). That way, each widget can handle each message in the most efficient way possible. Finally, because a user action might require us to redraw a widget, we'll allow the widget to tell us if it needs to be changed. Because drawing is important, we can minimize CPU usage by only redrawing whatever it says we *have* to redraw. Finally, for programmer sanity, we'll allow these widgets to nest, and to pass custom messages to each other.

well, actually...

...it started much earlier.

Congratulations! We've just designed Windows.

Specifically, Windows 1.0. Circa 1985.

Blasts from the Past

This is, really and truly, exactly how Windows used to be programmed all the way through at least Windows 7, and many modern Windows programs still work this way under the hood.[2] Views would be drawn whenever asked via a `WM_PAINT` message. To be fast, `WM_PAINT` messages generally only used state stored locally in the view,[3] and to keep them repeatable, they were forbidden from manipulating state. Because painting was separated from state changes, Windows could redraw only the part of the screen that actually needed to be redrawn. Each view had a function associated with it, called its `WndProc`,[4] that took four parameters: the actual view getting updated; `uMsg`, which was the message type as an integer; and two parameters called `wParam` and `lParam` that contained data specific to the message.[5] The `WndProc` would update any data stores in response to that message (frequently by sending off additional application-specific messages), and then, if applicable, the data stores could mark the relevant

Now **Redux** is all the rage (which is a variety of Flux).

But! We wanna use RxJS soooo... 🤔

There is also **MVI** from Cycle.js:

> MVI is a simple pattern to refactor the main() function into three parts: **Intent** (to listen to the user), **Model** (to process information), and **View** (to output back to the user).
>
> — *Cycle.js Docs*

And **MVI** comes with the **Cycle.js reducer pattern** in order to continuously manage and hold state. It is what happens in the **`model()`** part of MVI.

Update pattern in Elm. **Reducers** in Redux.

- **central to any type of unidirectional dataflow you would model with RxJS**
- you can model Flux, Redux or MVI using this pattern
- the rest is just slightly differently organized code

1. Create action streams (intent / user actions)
   `.fromEvent(), .fromPromise() .create() ...`
2. Merge them into one big fat stream of actions (Dispatcher)
   `.merge()` or `Rx.Subject`
3. Filter for what you are interested in, in a store or the model function
4. Map action to reducer / modifier function
5. Scan over the resulting stream of functions by applying each modifier on state

Currying and Partial Application

```
function openMail(action) {
    return function(state) {
        return state.map(mail => {
            if (mail.id === action.id) { mail.open = true; }
            return mail;
        }
    }
}

const openMailModifier = openMail(action);
const newState = openMailModifier(state);
```

Make it Nice With curry( )

```
function openMail(action, state) {
    return state.map(mail => {
        if (mail.id === action.id) { mail.open = true; }
        return mail;
    }
}

const curriedOpenMail = curry(openMail);
const openMailModifier = curriedOpenMail(action);
const newState = openMailModifier(state);
```

```
const receiveMails = (action, state) => action.mails;
const openMail = (action, state) => state.map(mail => {
    if (mail.id === action.id) { mail.open = true; }
    return mail;
});
const markAsRead = (action, state) => state.map(mail => {
    if (mail.id === action.id) { mail.read = true; }
    return mail;
});

const receiveMail$ = dispatcher$
    .filter(action => action.type === 'RECEIVE_MAIL')
    .map(curry(receiveMails)(action));

const openMail$ = dispatcher$
    .filter(action => action.type === 'OPEN_MAIL')
    .map(curry(openMail)(action));

const markAsRead$ = dispatcher$
    .filter(action => action.type === 'MARK_AS_READ')
    .map(curry(markAsRead)(action));

return Rx.Observable
    .merge(openMail$, markAsRead$)
    .scan((state, modFn) => modFn(state), []);
```

Rx.Subject

```
const dispatcher = new Rx.Subject();
dispatcher.next('PAYLOAD');
dispatcher.subscribe(x => console.log(x));
```

Rx.Subject as event emitter

```
const dispatcher = new Rx.Subject();
function dispatchAction(action$) {
    action$.subscribe(data => dispatcher.next(data));
}
```

LET'S SEE SOME CODE

A simple example and naive implementation using RxJS, jQuery and ES2015: github.com/kahlil/artists

A more complicated example using Angular 2, RxJS, TypeScript and a little helper library I made: github.com/kahlil/tinydraft