APP ARCHITECTURE WITH RXJS

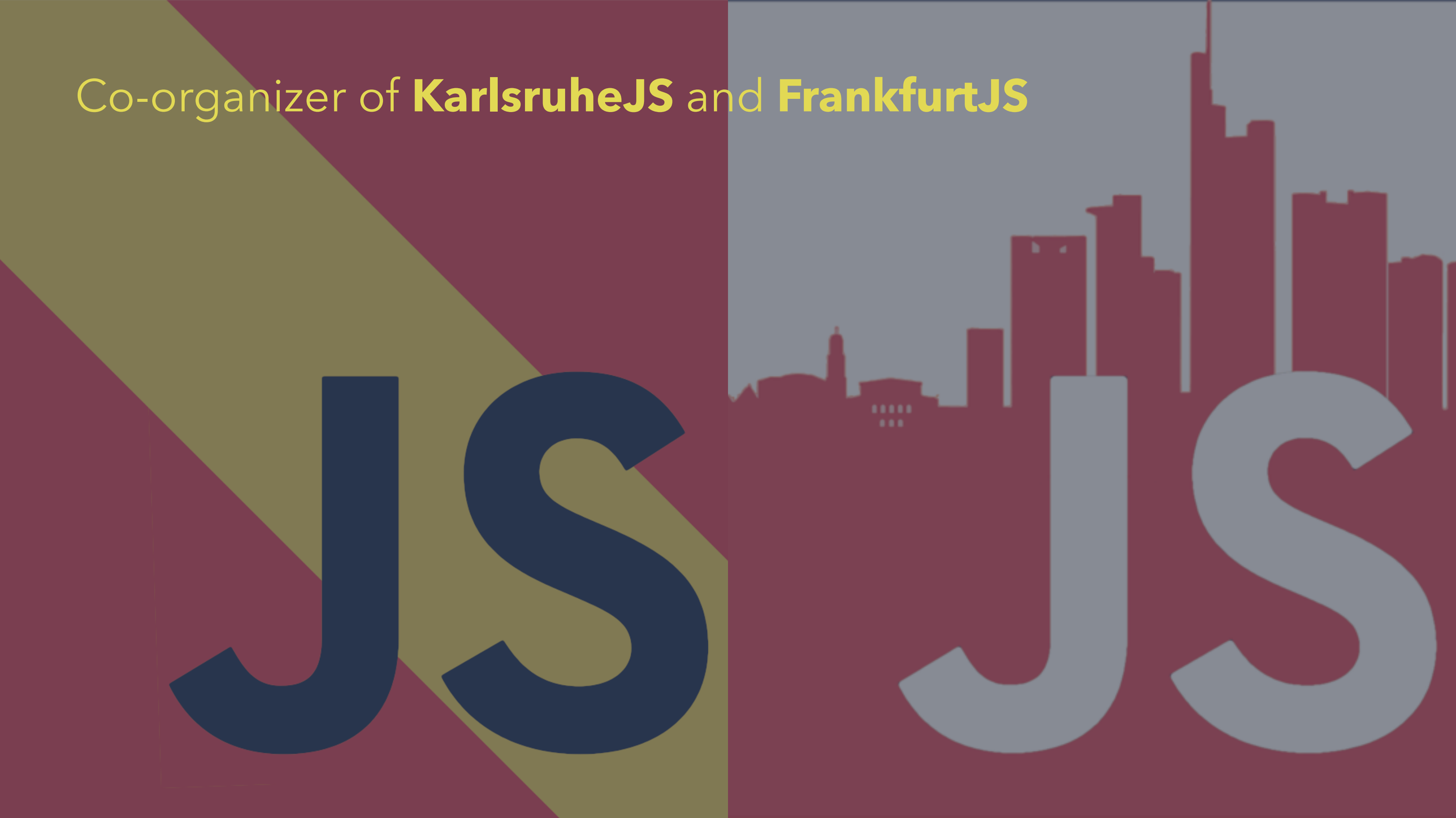
Join The Cult

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

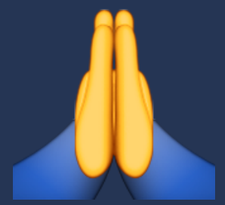
Not helpful to compare with Promise.

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

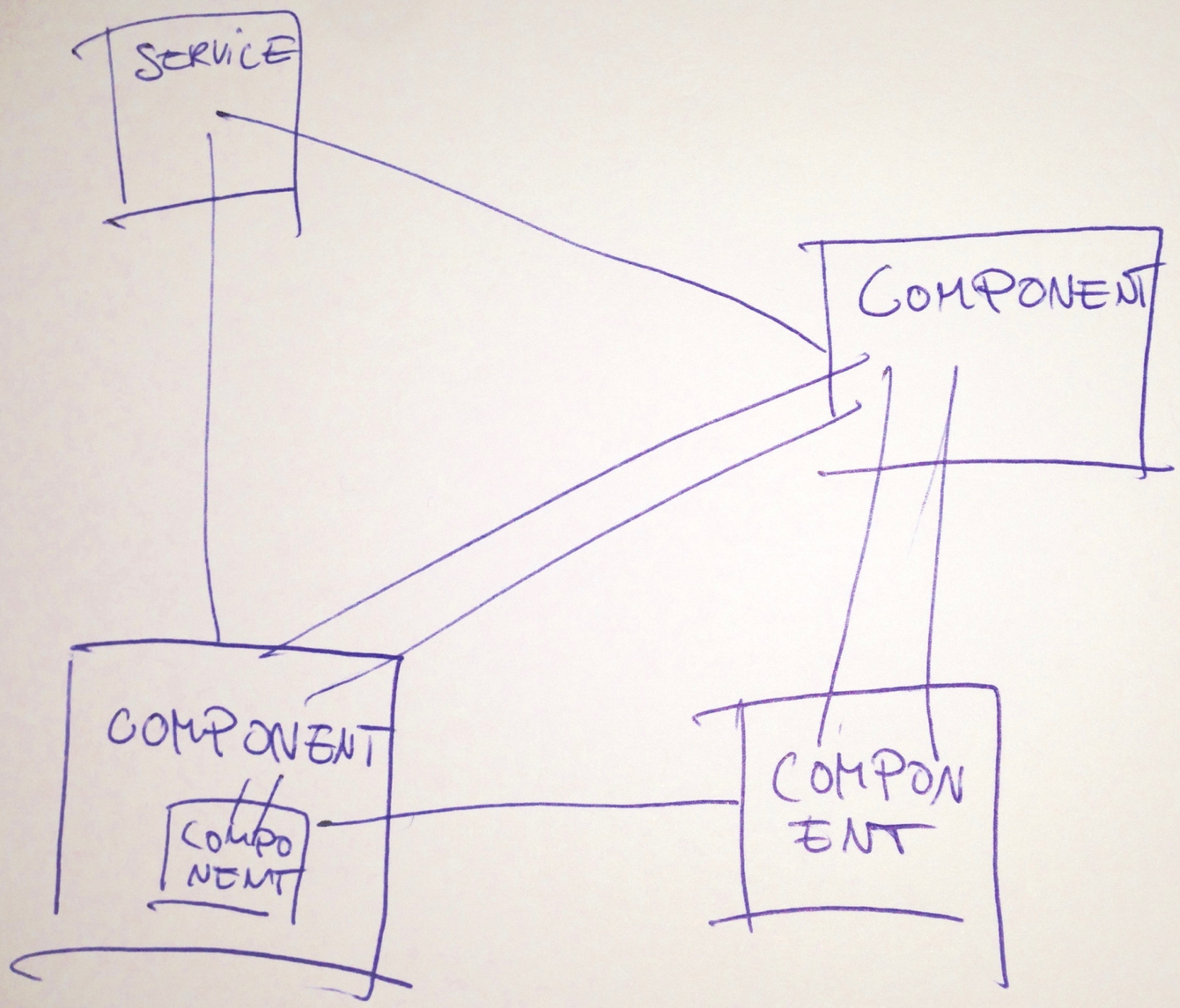
– Jafar Husain

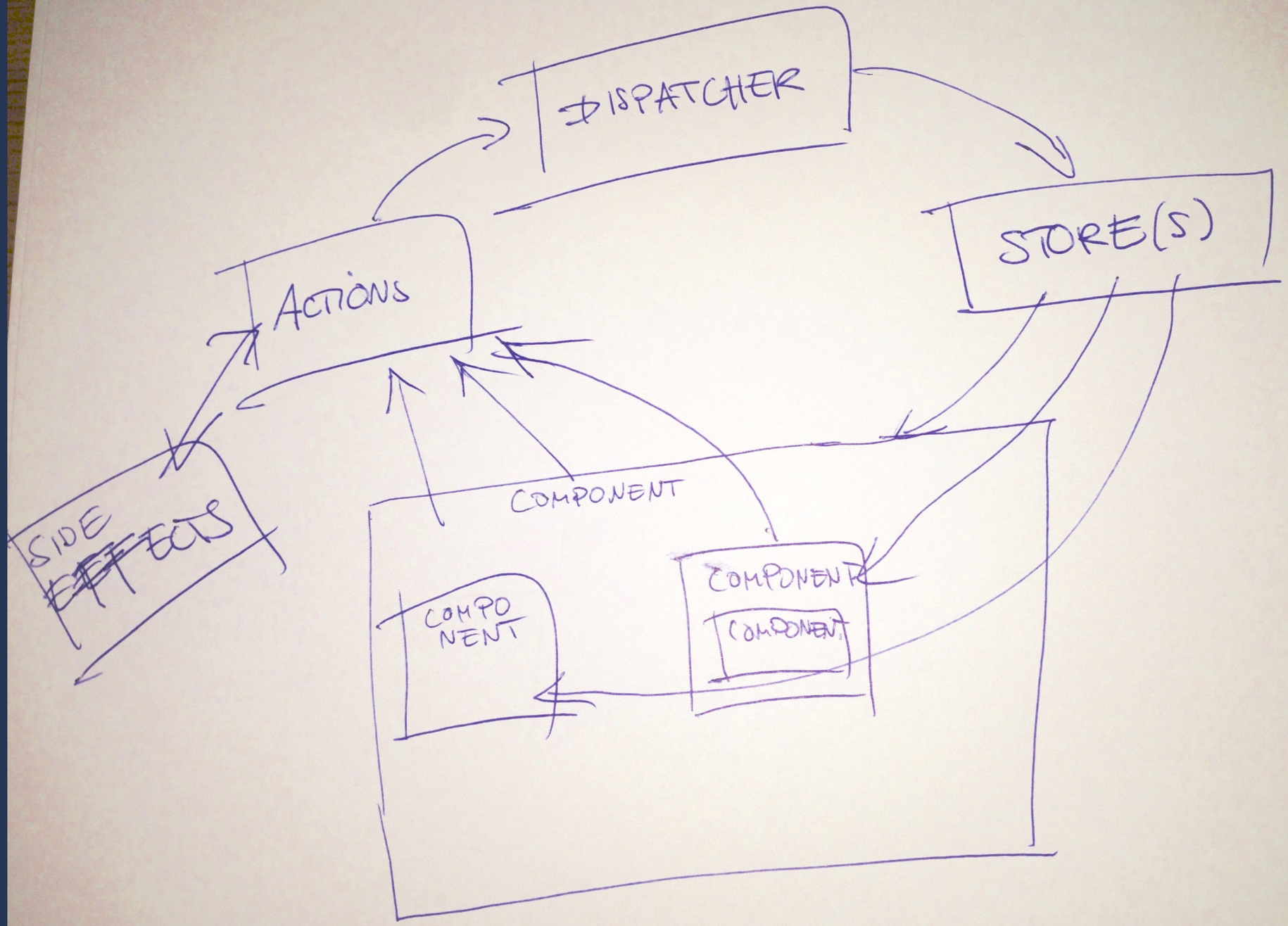
```
const stream = Rx.Observable.fromArray([1, 2, 3, 4, 5, 6]);

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



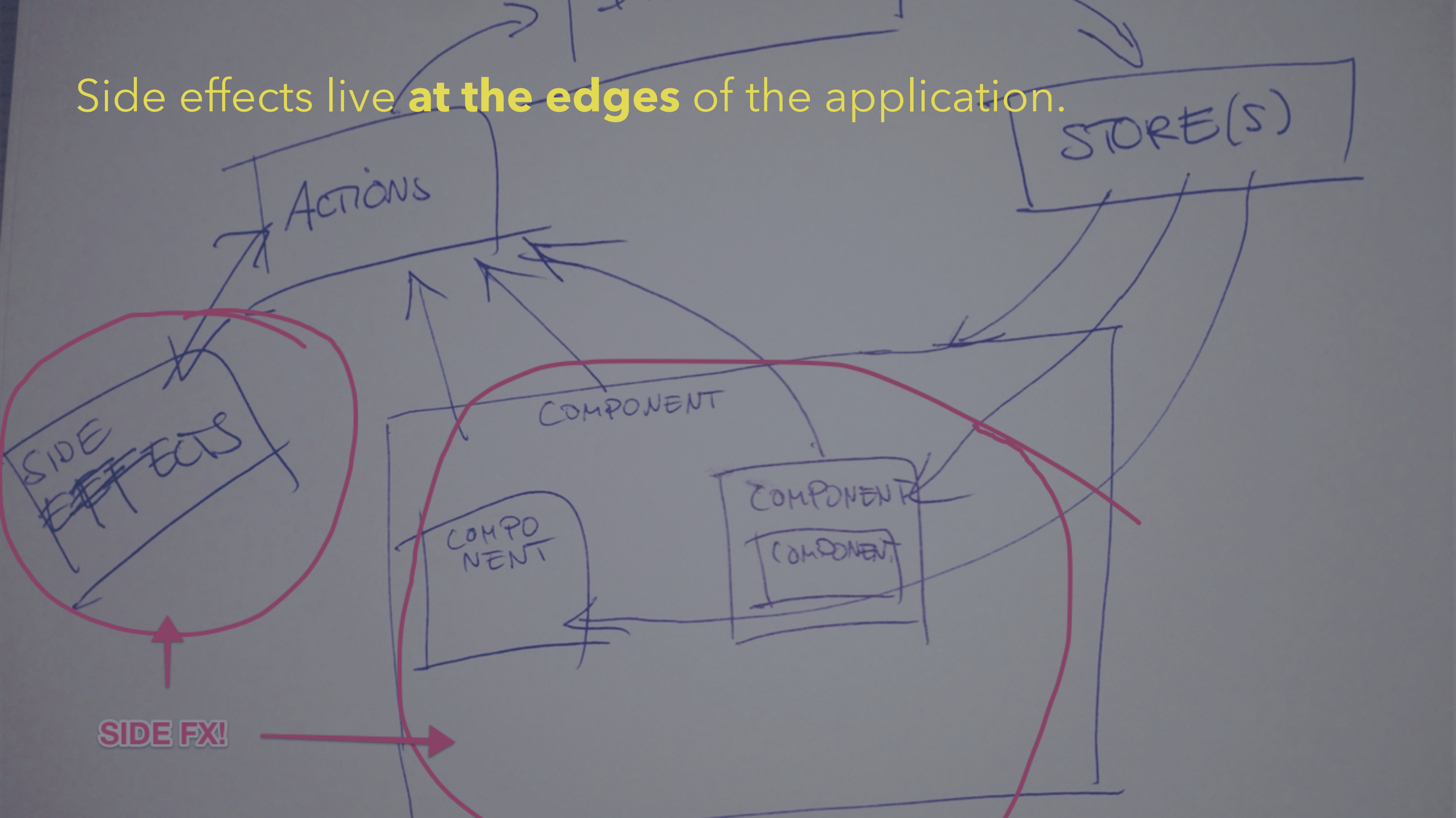
10 x BETTER THAN WHAT WE DID BEFORE


SERVICE
COMPONENT
COMPONENT
COMPO
NENT
COMPON
ENT


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

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

Side effects live **at the edges** of the application.



Cycle.js and Elm which are functional reactive frameworks actually explicitly isolate side effects in things they call **ports** (Elm) and **drivers** (Cycle.js) and in the Redux you isolate them in so-called Thunks in Redux middleware.

Currently **the best known way** to keep **a big or continuously growing app** maintainable and modifyable.

The **developer controls the code**. Not the other way round.

DEVELOPER HAPPINESS

It started with **Flux** early 2014.

(Actually it started much earlier.)

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

But 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.

1. Create action streams (intent / user actions)
   `.fromEvent()`, `.fromPromise()` ...
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

```
const openMail$ = dispatcher$
    .filter(action => action.type === 'OPEN_MAIL')
    .map(action => state => state.map(mail => {
        if (mail.id === action.id) { mail.open = true; }
        return mail;
    }));
const markAsRead$ = dispatcher$
    .filter(action => action.type === 'MARK_AS_READ')
    .map(action => state => state.map(mail => {
        if (mail.id === action.id) { mail.read = true; }
        return mail;
    }));
return Rx.Observable
    .merge(openMail$, markAsRead$)
    .scan((state, modFn) => modFn(state));
```

This reducer pattern is **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.

LET'S SEE SOME CODE

github.com/kahlil/tinydraft