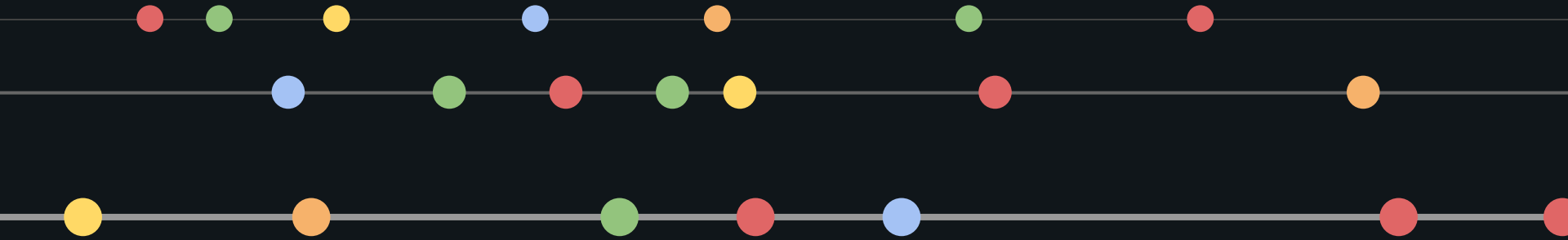
Reactive Programming



Marius Lundgård
Frontendforum, 2016
NRK Medieutvikling

Virkeligheten er asynkron

Virkeligheten er asynkron

clicks

Virkeligheten er asynkron

chat

clicks

lyd

scroll

Virkeligheten er asynkron

chat

clicks

lyd

video

scroll

meldingstrøm

pageviews

Virkeligheten er asynkron

chat

dashboard

clicks

lyd

video

scroll

chatbots

meldingstrøm

pageviews

Virkeligheten er asynkron

second screen

chat

dashboard

clicks

websockets

lyd

live-resultater

video

scroll

chatbots

meldingstrøm

pageviews

Virkeligheten er asynkron

http

second screen

chat

requests

dashboard

clicks

websockets

lyd

live-resultater

video

scroll

realtime

collaboration

chatbots

meldingstrøm

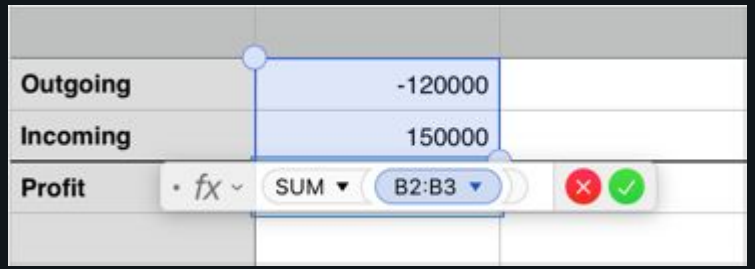
pageviews

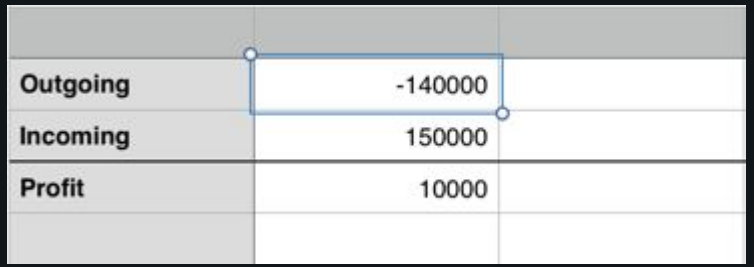
Virkeligheten er asynkron

- Front end applikasjoner øker i kompleksitet – tusenvis av states!
- FRP modellerer forandring over tid (mutable state)
- Spreadsheets
- Sjekk ut draw-cycle

Virkeligheten er asynkron

- Front end applikasjoner øker i kompleksitet – tusenvis av states!
- FRP modellerer forandring over tid (mutable state)
- Spreadsheet-metaforen





Side effects

- Operasjoner som endrer *state* i systemet (state mutation)
- Ingenting er galt med hverken *state* eller *side effects*, men det fører til «accidental complexity» dersom man det brukes overalt

```
function right (event) {
  event.preventDefault() // side effect
}

function right (html) {
  document.body.innerHTML = html // side effect
}
```

```
function wrong (arr) {
  arr.push('foo') // side effect (oops!)
  return arr.length
}
```

Pure functions

- Gitt samme *input*, skal pure functions gi samme *output*
- Påvirker ikke andre deler av systemet
- Endrer ingen argumenter – returnerer en verdi basert på argumentene

```
function a (x, y) {
  return x + y * 10 // pure
}
```

```
function b (x, y) {
  return x + y * Math.random() // impure
}
```

Mutation og Immutability

- Referanser vs. instanser: objekter i JS kan muteres
- Fremfor å endre data – lage en ny verdi
- Teori om persistente datastrukturer (sjekk ut Rich Hickey og/eller Lee Byron)

```
const a = [1, 2, 3]

console.log(a)
// 1, 2, 3

a.push(4) // mutation

console.log(a)
// 1, 2, 3, 4
```

```
const b = [1, 2, 3]
const c = b.concat([4])

console.log(b)
// 1, 2, 3

console.log(c)
// 1, 2, 3, 4
```

“FRP integrates time flow and compositional events into functional programming.”

https://wiki.haskell.org/FRP

FRP

- En variant av Reactor pattern
- Dukket opp i 1997 og popularisert gjennon Rx.NET av Erik Meijer (Microsoft)
- **Observer** og **Observable** – de viktigste datatypene i RP

	single items	multiple items
synchronous	`T`	`Array<T>`
asynchronous	`Promise<T>`	`Observable<T>`

Observer

- Den som mottar data
- Behandler 3 situasjoner: next, error og complete

```
const observer = {
  next: (value) => {},
  error: (error) => {},
  complete: () => {}
}

observable.subscribe(observer)
```

Observable

- Bruk av Observables likner på Array

  ```
  .map()
  .filter()
  .reduce()
  +++
  ```

- I motsetning til Arrays kommer verdier i sekvens – til ulik tid, en om gangen
- Kalles også ‘Stream’ og ‘Signal’
- Å opprette en Observable kan sammenliknes med å definere en funksjon – ingen kode kjøres før man «kaller» den (via `.subscribe()`)

```
/* Array */
const values = [1, 2, 3]
const doubles = values.map((x) => x * 2)

doubles.forEach(console.log)
// 2, 4, 6



/* Observable */
const value$ = Observable.of(1, 2, 3)
const double$ = value$.map((x) => x * 2)

double$.forEach(console.log)
// 2
// 4
// 6
```

```
const value$ = Observable.of(1, 2, 3)

// .forEach() is the same as .subscribe()
value$.forEach(console.log)
// 1
// 2
// 3

const double$ = value$.map((x) => x * 2)

double$.subscribe(console.log)
// 2
// 4
// 6
```

```
const value$ = Observable.of('2', '4', '6')

const int$ = value$.map((x) => parseInt(x, 10))

const moreThan3$ = int$.filter((x) => x > 3)

moreThan3$.subscribe(console.log)
// 4
// 6


// Marble diagram of `moreThan3$`
//     of -----'2'--'4'--'6'|
//    map ------2----4----6-|
// filter ------------4----6-|
```

```
const location$ = Observable.fromPromise(fetch('/api/v0/location/oslo'))
  .map(res => res.body)

const name$ = location$.map((d) => d.name)

locationStream.subscribe(console.log)
// {...}
```

```
const refreshButtonClick$ = Observable.fromEvent(refreshButtonEl, 'click')

const location$ = refreshButtonClick$
  .flatMap(() => Observable.fromPromise(
    fetch('/api/v0/location/oslo')
      .then(res => res.json()))
  )

location$.subscribe(console.log)
// {...}
// {...}
// {...}
```

```
const tick$ = Observable.timer(0, 1000)

const location$ = tick$
  .flatMap(() => Observable.fromPromise(
    fetch('/api/v0/location/oslo')
      .then(res => res.json()))
  )

location$.subscribe(console.log)
// {...}
// {...}
// {...}
```

https://github.com/mariuslundgard/ff-reactive

MVC → IMV
(filosofi)

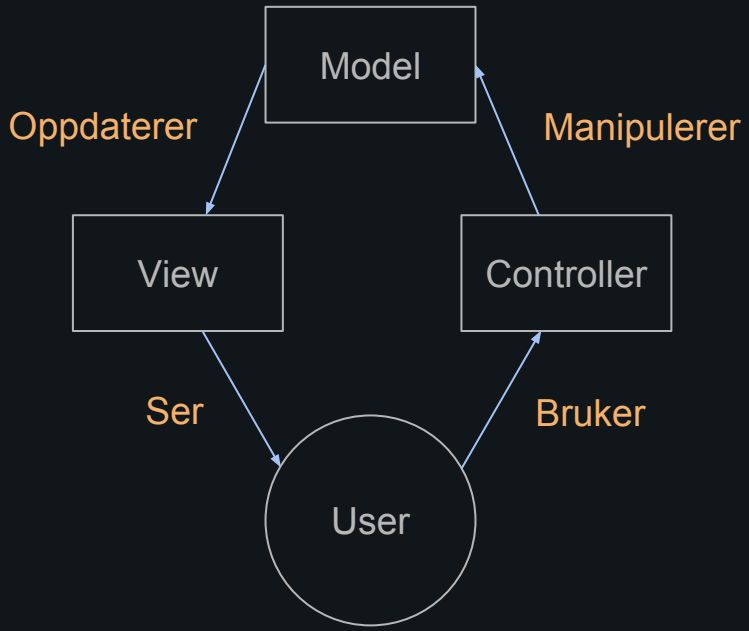
MVC

- Divide and conquer:
  avgrense problemer og *state* til små medgjørlige deler
- Skiller uttegning og endring fra data-logikk
- «Imperative» stil (mutere state direkte) `a.id = 23`
- Observer pattern

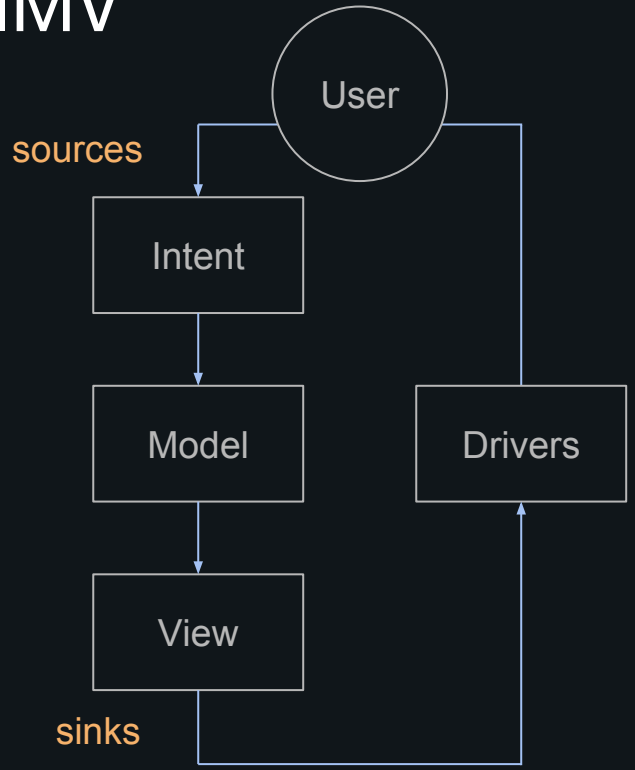
IMV

- En abstraksjon for FRP-komponenter
- Tar utgangspunkt i *input* fra brukeren (upstream)
- Behandler events (asynkron) og data (synkron) på samme måte
- Alt er lister (signals/streams)
- View er en «konsekvens» av data (downstream)
- Deklarerer ønsket resultat
- Observable pattern

MVC


Model
Oppdaterer
Manipulerer
View
Controller
Ser
Bruker
User

IMV


User
sources
Intent
Model
Drivers
View
sinks

Så hva kan dette brukes til?

Lenkeliste

- Sted å starte for funksjonell programmering i JS: Introducing Ramda
- Egghead.io: Introduction to Reactive Programming (Andre Staltz)
- Artikkel: Model-View-Intent
- Dokumentasjon: rxmarbles.com
- Biblioteker:
  - RxJS
  - most
  - xstream
  - +++
- Spesifikasjon: EcmaScript proposal
- Rammeverk: Cycle.js
- Runtime/programmeringsspråk: Elm