

# FUNCTIONAL PROGRAMMING IN VIEWS

@mikaelbrevik

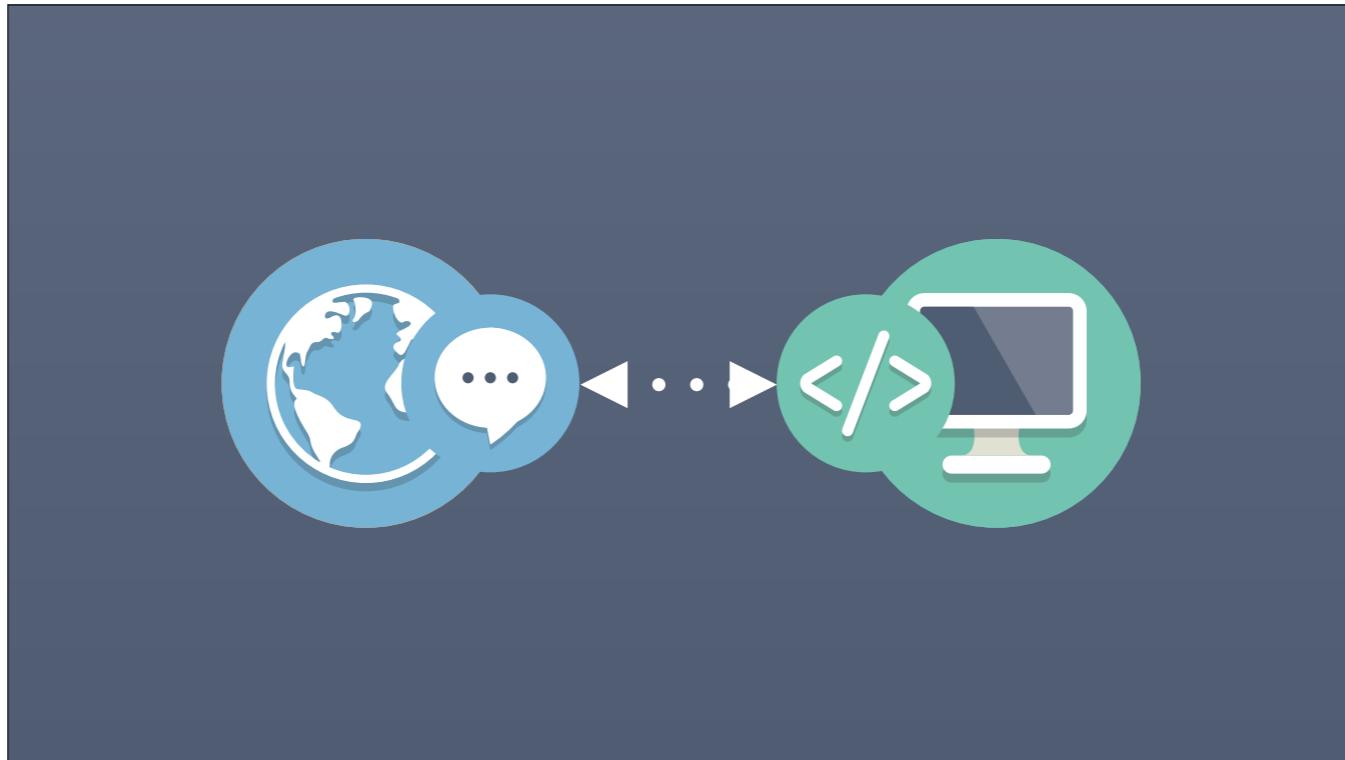


- Hi!
- Today I'll be talking about basic functional programming applied to UIs, unidirectional data flow, and how to make our views simpler.
- Først litt om meg: Jeg er Mikael Brevik og har jobbet som utvikler i BEKK Trondheim i 2 1/2 år, hvor jeg også er fagleder for Frontend gruppa.
- I frontend-gruppa tar vi for oss alt som har med frontend og gjøre og har regelmessige til dels høylytte diskusjoner om hva som er beste fremgangsmåte.



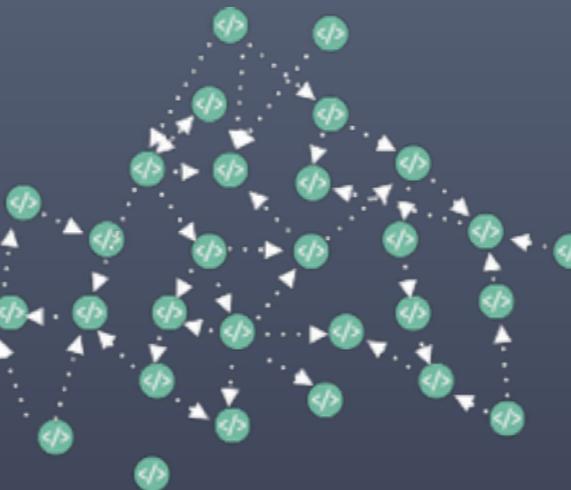
So, how did we get here? Why do we need to re-evaluate how we make UI?

- “In the beginning” we had the traditional server - client relationship. Simple request and response.
- **(next)** That was it. Simple flow, and what was rendered on the client was our system output.
- If we were to update our view we had to get more data from the server and re-render the entire page. **A complete refresh.**
- This obviously had its drawbacks, but we had complete static models of our page. The HTML was the finished product. We could easily read from top to bottom and know exactly our structure. It was simple.
- **It was far from easy, fast or flexible. But it was simple and predictable in output.**



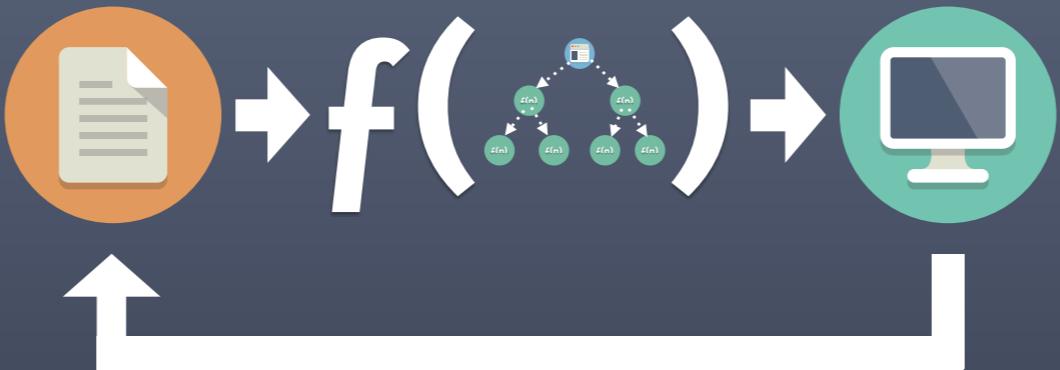
- As the web matured we found the need for dynamic content - and not doing complete browser reload every time we wanted to change a tiny bit of information on the page.
- **(next)** We started moving more code **from the server to the client**.
- Instead of sending all structure and content on the initial load, we do AJAX requests and dynamically create the content on the client-side.
- We build up a **massive time-dependent state representation in the DOM** and we **no longer have the complete static models** we had before.
- The advantage is more flexibility and potential speed, but we often update the DOM, bit-by-bit and
- **Tightly couple our application to the DOM integration-point.**
- **We build a ever changing blob of state that is very hard to predict the output of.**

# VIEWS OUT OF CONTROL



- We can think of our pages and applications of multiple pieces of code which is in some way connected and communicating with each other.
- The problem we often have with our applications, is that we have no clear direction of communication.
- (**next**) Our pieces of code can send messages in multiple directions and the pieces can be tightly coupled.
- In many cases, it's impossible to see at a single piece of code and imagine what the actual output might be. **What if two pieces share data and that data changes in one of the pieces?**
- If all pieces are tightly coupled, we can't simply change the behaviour of one individual piece and expect it not to have repercussions for the rest of the system.
- With shared state and multi-directional communication we essentially make our entire application the source for potential bugs, instead of isolating parts of the program which is much less braided.
- We need to have better control of our code. **This needs to be simpler.**

## WHAT WE WANT



- Men det vi egentlig vil ha er en mye enklere flyt i applikasjonen vår. Vi vil bare tenke på det som en funksjon av tilstand eller av tilstand på et gitt tidspunkt.
- Vi ønsker å redusere kompleksitet, og det vi må ha i hodet til en gitt tid. Med et system som ikke har en tydelig flyt, vil vi måtte holde alt for mye informasjon i hodet dersom vi skal skjønne hva noe produserer.
- Vi skal prøve å oppnå en flyt som ser slik ut. Og underveis besøker vi kanskje noen programteoretiske prinsipper, noen paradigmer og et par fancye ord.

$$f(x) \rightarrow 2x$$

$$f(2) = 4$$

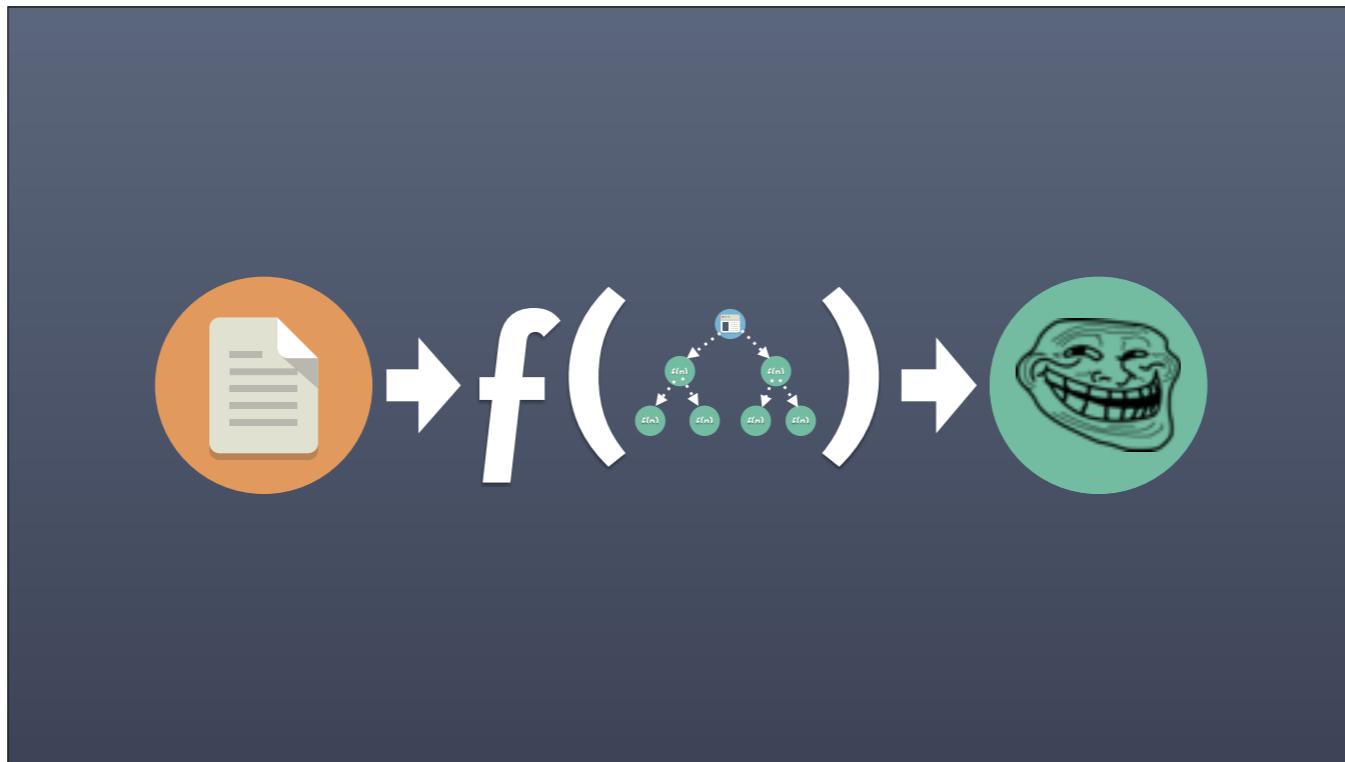
$$f(2) = 4$$

- Så hvordan kan det ofte være nå? Om vi prøver å se på det som et mattestykke: Vi har en matematisk funksjon som vi prøver å kalle.
- De to første gangene gir det oss forventet resultat: 4



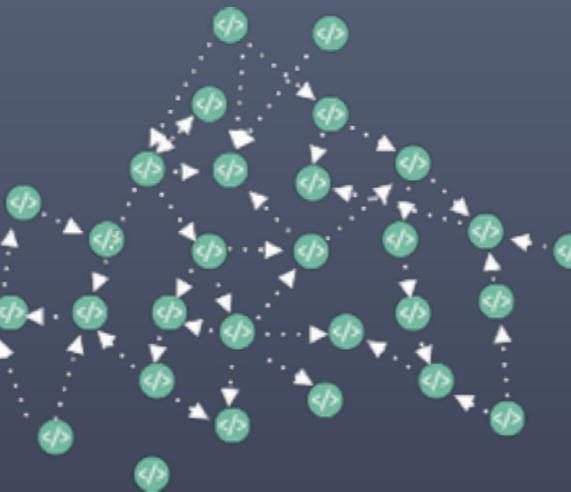
quote @torgeir

- Men neste gang blir det noe helt uventet? 5??
- (**next**) WTF?



- Akkurat samme blir det ofte i grensesnitt ofte. Det fungerer noen ganger, men så helt uprovosert
- (**next**) kommer det et troll av en uventet data fra ingenting. Uten at vi enkelt kan finne ut hvorfor det endres eller hvordan.

## VIEWS OUT OF CONTROL



- Det skyldes i mange tilfeller for at vi har views som er helt ute av kontroll. De kommuniserer diagonalt, vertikalt, fremover, bakover, osv. De er alt for tett koblet og det er umulig å følge en flyt av informasjon uten å konsultere Skynet.

# LOOSELY COUPLED & UNIDIRECTIONAL MESSAGES



- Så hvordan burde dette være? Vi må forenkle flyten!
- Hvordan kan det bli gjort? Vet å forenkle hver puslebit til å være enklere.
- De burde bare kommunisere i én retning! Ta noe data, transformere det og gi ut noe data.
- Dette er ofte det programmering og særlig funksjonell programmering handler om.

# REFERENTIAL TRANSPARENCY

```
> 4 + 4  
> 2 * 4  
> 8
```

- Da kommer vi inn på dagens første store ord. Dersom man skal være en funksjonell programmerer, må man kunne briljere med store ord.
- Referential Transparency sier bare at våre programmeringsfunksjoner burde være slik som matematiske funksjoner og operasjoner.
- Helt spesifikt sier det at vi skal kunne erstatte et funksjonskall med resultatet av det funksjonskallet uten at det påvirker systemet vårt på noen måte.

# REFERENTIAL TRANSPARENCY

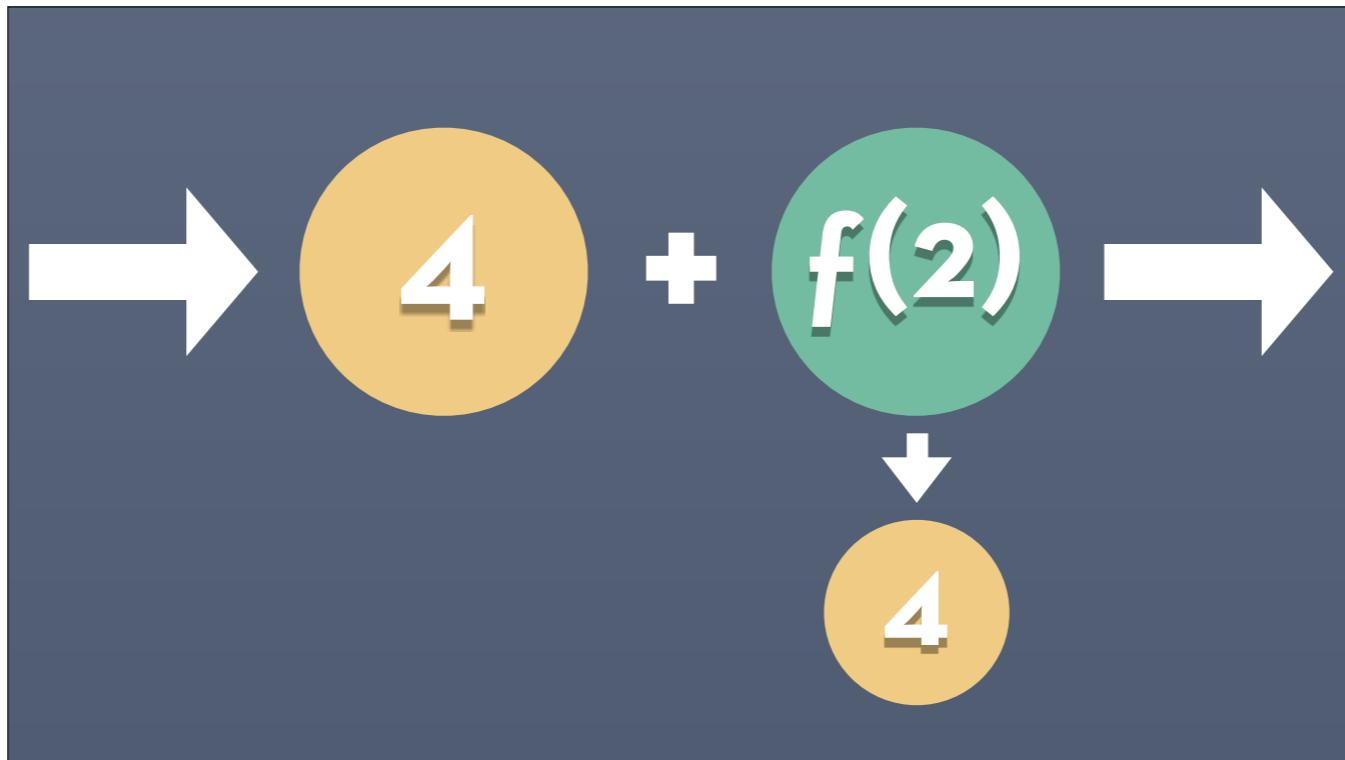
```
const square = (n) => n * n;  
> square(2) + square(2);  
> 2 * square(2);  
> 2 * 4  
> 8
```

- Vi kan se det enda tydeligere med kode.
- Alle disse linjene vil produsere det samme resultatet.

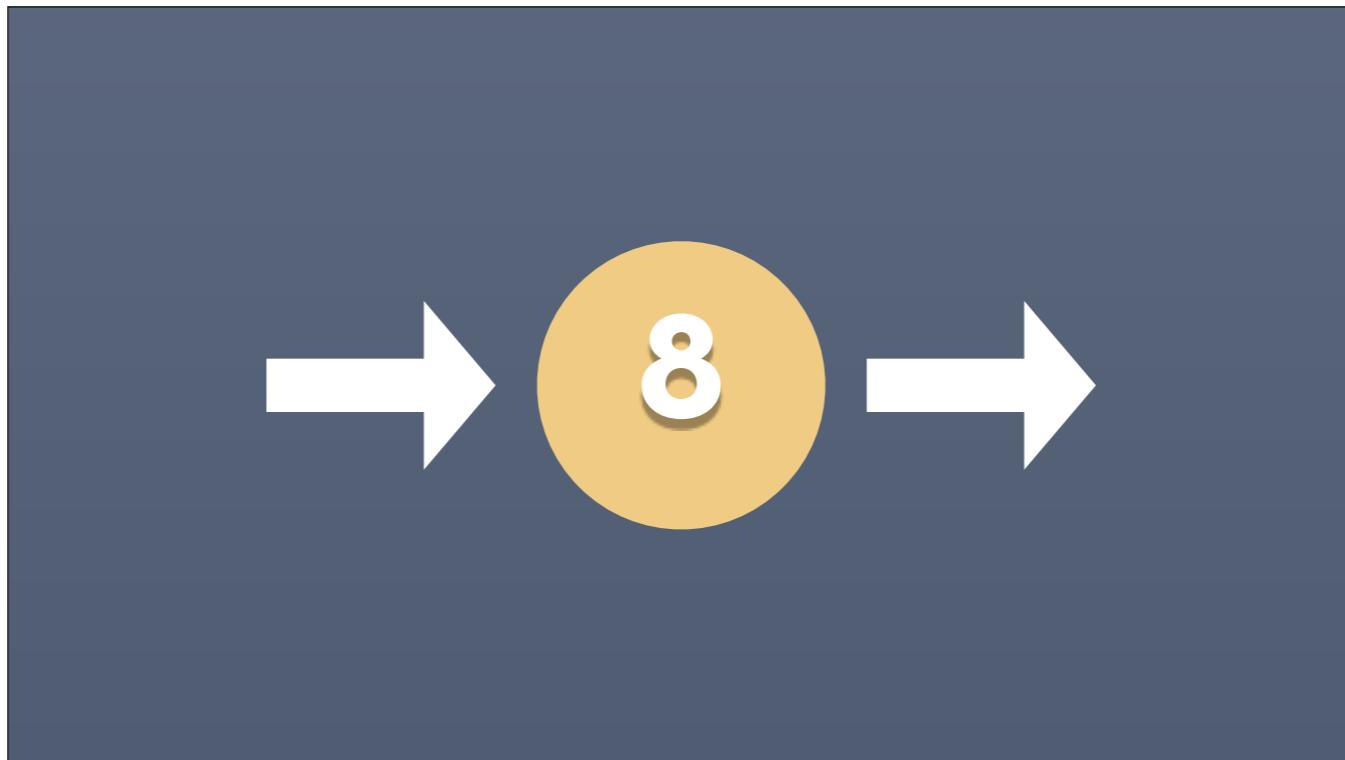
# PREDICTABILITY

```
const square = (n) => n * n;  
> square(3); // 9  
> square(3); // 9  
> square(3); // 9  
> square(3); // 9
```

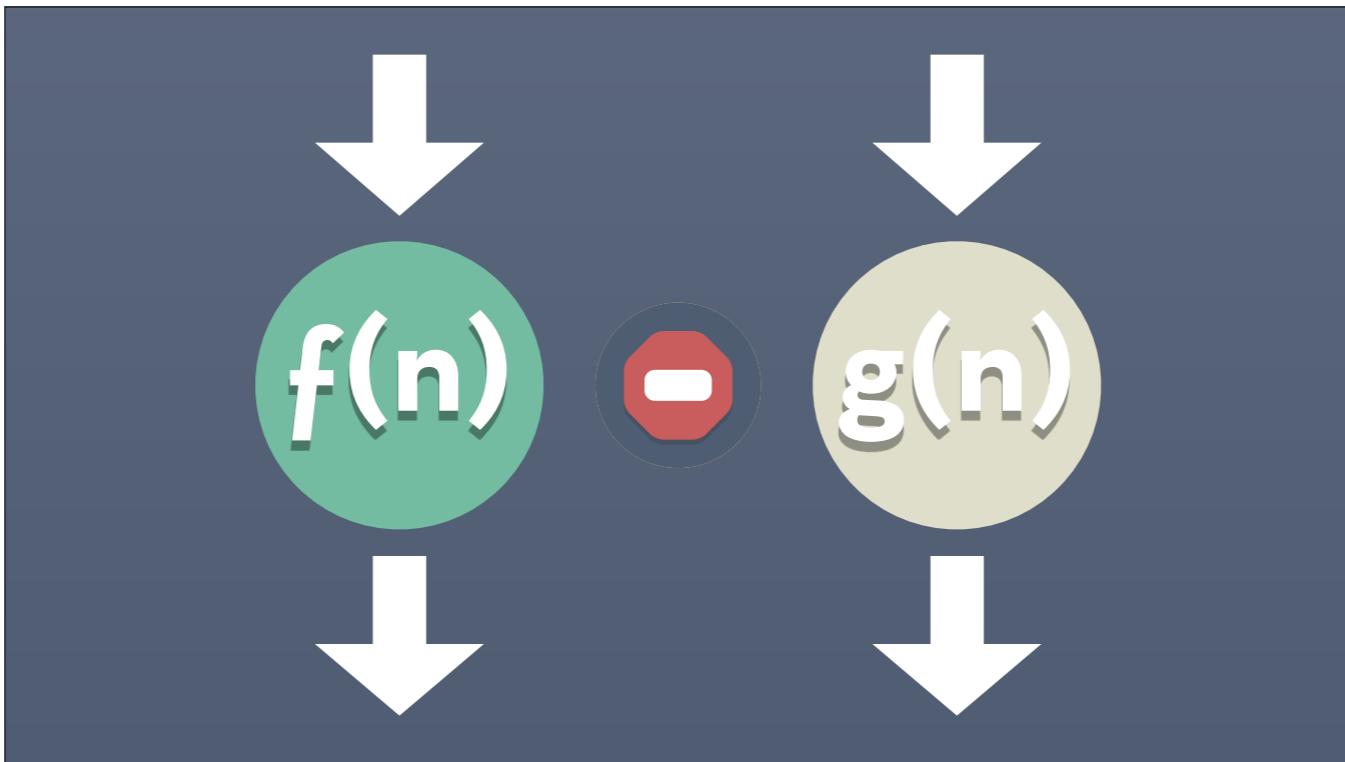
- This means our referentially transparent functions are very predictable.
- Uansett hvor og når slike funksjoner blir kalt, dersom input er det samme vil output og være nøyaktig det samme.
- Dette er verdt å tenke litt over og vi kommer tilbake til det senere: om input er det samme, er output det samme.
- Dette vil være testbart. Per definisjon, dersom byggeblokkene våre er forutsigbare vil de være testbare.



- Simply put: A function is referentially transparent if, at invocation level, we can swap out the function call with the result of the invocation.
- **(next)** For instance, if we have a square function and we pass it the argument of 2, we can swap out the invocation with the number 4 **without it changing the behaviour of the system at all**.
- Expressions can be referentially transparent as well. In JavaScript, we have built in operations like `add` that is, by nature, a referentially transparent operation given that we just add two numbers.
- **(next)**



- So if we have an expression that say 4 plus 4, we can replace that with the resulting 8, and our system would still **behave the exact same way**.



- One important feature for functions that is referentially transparent is *purity*.
- **(next)** We cannot have any side-effects between functions.
- **Not one code inside a function can alter the behaviour of a different section of your application except in its output.**

## RESULTS OF SHARED MUTABLE STATE



- This is the effect of having shared mutable state.

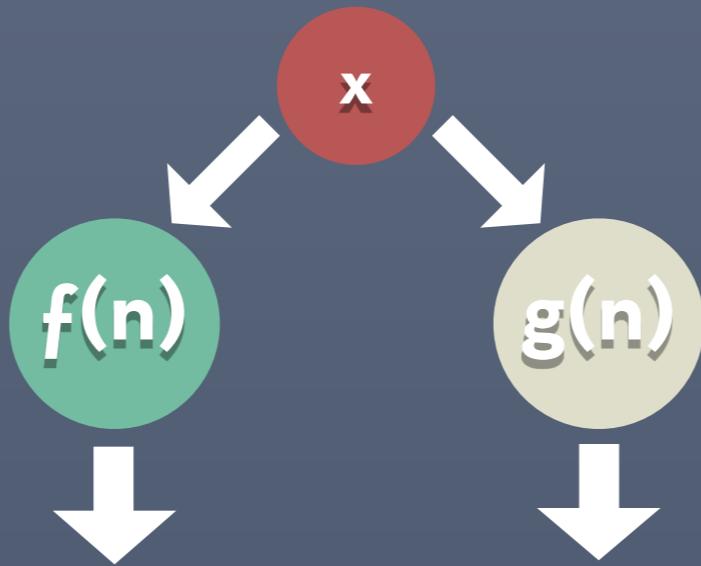
# ACCIDENTAL SHARED STATE

```
const setName = (obj, val) => obj.name = val;
const logInTime = (obj) =>
  setTimeout(() => log(obj.name));

const myObj = { name: 'Janet van Dyne' };
logInTime(myObj); // Logs 'Oops'
setName(myObj, 'Oops');
```

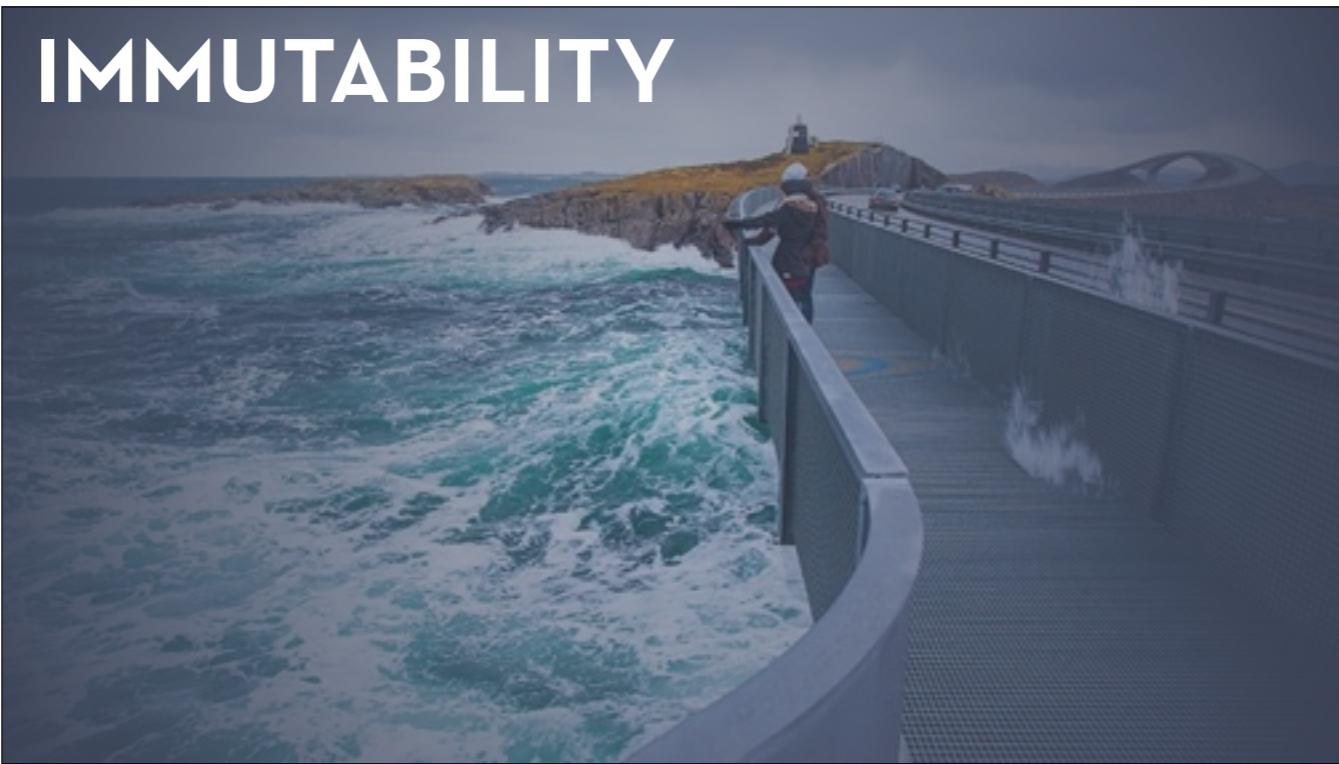
- Often we can have side-effects without even knowing it.
- Objects in Javascript is per default mutable.
- (**next**) This means, if two functions get passed the same object and that object is changed in some way inside one of the functions, the **other function has to live with the consequences of that change**.
- This is a fairly common side-effect, and often a source of bugs in larger systems.

# CAN'T PASS MUTABLE OBJECT



- Passing other values than objects by reference could also lead to these types of side-effects.
- This is one way our functions can get tightly coupled.

# IMMUTABILITY



- There might be a better way to avoid the messiness that is shared mutable objects leaking through our system.
- Instead of passing in mutable objects, we can use immutable objects.

# REFERENCE CHECKS

```
const obj = { name: "T'Challa" };

> obj === obj;
> obj !== { name: "T'Challa" };
> deepEqual(obj, { name: "T'Challa" });
```

- As we always create new objects when we change any of its values, we can easily check if two objects are the same.
- **(next)** We won't have to do deep value checks, but simply see if two variables points to the same memory slot.

# REFERENCE CHECKS

```
const obj = { name: "T'Challa" };

> obj === obj;
> obj !== { name: "T'Challa" };
> deepEqual(obj, { name: "T'Challa" });
```

- Reference checks a very cheap operation.

# REFERENCE CHECKS

```
const obj = { name: "T'Challa" };

> obj === obj;
> obj !== { name: "T'Challa" };
> deepEqual(obj, { name: "T'Challa" });
```

- In contrast to deep value checks, which can be costly with large structures.

# IMMUTABLE MAPS

```
const obj = Immutable.Map({ name: "T'Challa" });
const obj2 = obj.set('name', 'Black Panther');
obj.get('name'); //=> T'Challa
obj2.get('name'); //=> Black Panther
```

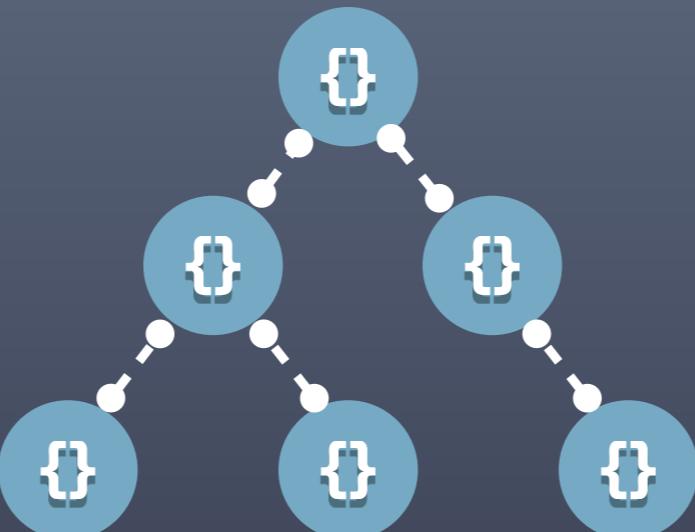
- With immutability, instead of altering the existing object, you would create a new object when it is changed and you would have to store the new reference created for the new object.
- **This means, the original object is unchanged and we can still have a reference to it.**
- Not only will we avoid accidental shared state, but also allow us to re-use values and structures in different ways.
-

# REFERENCES IN IMMUTABLE

```
const obj = Immutable.Map({ name: "T'Challa" });
const obj2 = obj.set('name', 'Black Panther');
obj !== obj2;
obj === obj;
```

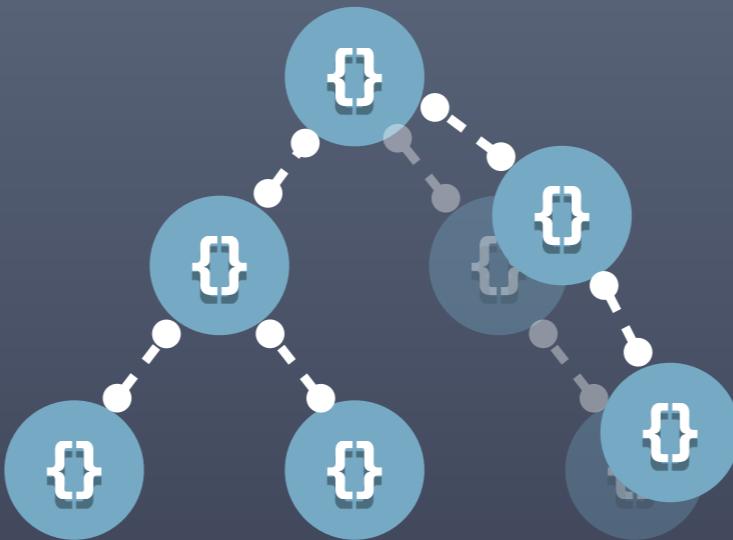
- Så om vi har en referanse til den forrige tilstanden vil ikke den være lik den nye.
- Objektene de peker til er forskjellige.

# STRUCTURAL SHARING



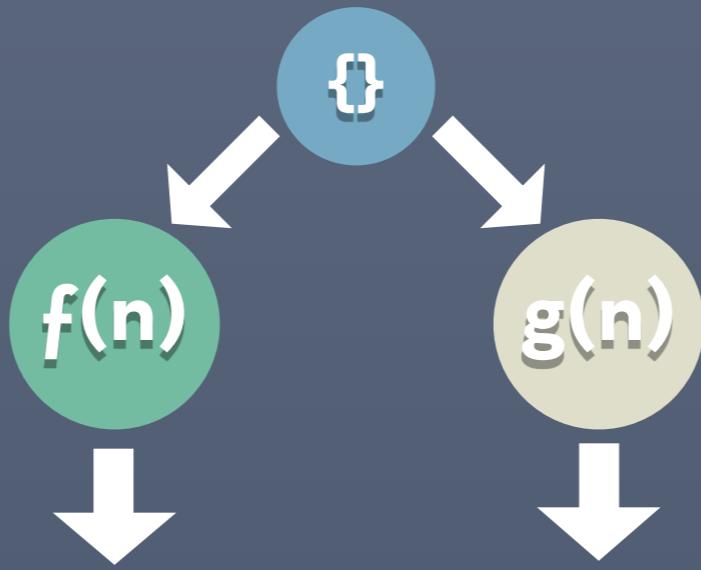
- It might sound like immutable structures is not memory efficient, but that's not really the case.
- We can use something called persistent data structures.

# STRUCTURAL SHARING



- In which, if we change a small part of a larger structure, we can re-use most of the original structure, only swapping out the part that has changed.

# CAN PASS IMMUTABLE OBJECT



- By having immutable values we can pass the same object to different functions and not worry about whether or not the object has changed.
- The only part we are concerned about is what the functions outputs!

# PURITY IN A STATEFUL ENVIRONMENT (DOM)



- Så da er vi vel reddet? Nå har vi jo immutabel data som vi sender inn i funksjonene våre, så vi kan ikke ha noe shared mutable state?
- Vell. Siden vi lager web apps, så må vi jo kommunisere med DOM-en.
- DOM-en er egentlig bare en massiv bolle av en mutabel tilstand som alle kan dele en referanse til. Og selv om vi bruker immutabel datastruktur som flyter igjennom programmet vårt, så kan vi måtte bli påvirket av DOM-en.

# ACCIDENTAL SHARED STATE

```
const setName = (obj, val) => obj.name = val;
const logInTime = (obj) =>
  setTimeout(() => log(obj.name));

const myObj = { name: 'Janet van Dyne' };
logInTime(myObj); // Logs 'Oops'
setName(myObj, 'Oops');
```

- Akkurat på samme måte som vi hadde det med vanlige objekter.

## ACCIDENTAL SHARED STATE IN THE DOM

```
function updateH1 (h1, val) {  
    h1.textContent = val;  
    return h1;  
}  
  
// <h1>Janet van Dyne</h1>  
var el = document.querySelector('h1');  
logInTime(el); // Logs 'Oops'  
updateH1(el, 'Oops');
```

- If all of our building blocks can communicate with the DOM, our functions aren't pure and referentially transparent, even if it may look that way initially.
- Tommelfingerregel på dette er å aldri bruke DOM-en som en kilde for sannhet, men noen ganger holder ikke det for at man er avhengig av dataen som er der

## THE DOM AS A SIDE-EFFECT



- Vi er egentlig litt tilbake til all bi-effektene og den delte tilstanden vi hadde tidligere.

$$\begin{aligned}f(x) &\rightarrow 2x \\f(2) &= 4\end{aligned}$$

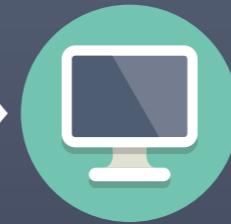
- Vi har det i alle fall ikke på denne måten. Der vi har forutsigbar, bi-effektfri kode som er konsekvent og lett og tenke på!

# SYSTEM AS A FUNCTION OF STATE

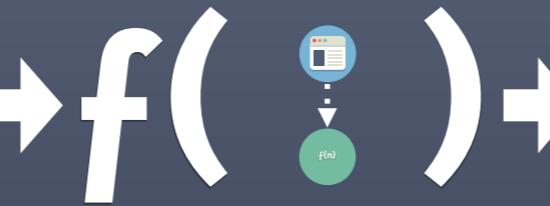
System Input



System Output



All of our UI code



- For det vi egentlig vil ha er en kommunikasjon fra start til slutt og aldri bruke slutt-tilstanden til noe, og ikke tenke på hvordan den blir håndtert. Visningen i nettleseren er sluttmålet vårt, det er bare resultatet av systemet, burde ikke være en del av logikken vi gjør.
- Men dersom vi har gjort hele systemet vårt til å være slik som matematiske funksjoner: enkle og bare tar input og gir output, kan vi tenke på hele systemet vårt som en stor funksjon.

## SYSTEM AS A FUNCTION OF STATE

```
var systemOutput = allOfUICode(systemInput);  
React.render(systemOutput, outputDomElement);
```

- I kode vil det se noe slik som det her ut: Vi henter ut en output ved å sende inn noe tilstand i alle vår UI logikk.
- Most of you have probably seen React. React or other Virtual DOM frameworks or libraries, can allow us to write, not **HTML templates**, but **HTML representations, either using JSX or just plain JavaScript**, and have smart ways of updating our DOM.

# SYSTEM AS A FUNCTION OF STATE

A lot of View Components

```
var systemOutput = allOfUICode(systemInput);
```

```
React.render(systemOutput, outputDomElement);
```



- What these sort of libraries do is **build an internal abstract DOM tree**, and **only when any of the internal representations changes**, the **appropriate update steps get taken to output to the actual DOM**.
- The HTML representations are often called components.

# COMPONENTS

```
const MyHeader = (name) => <h1>Hello, {name}!</h1>;
```

- A component, is a small bit of view code. Like a HTML element that contains one or more elements them self - **but in code**.
- By having components as a part of our programming we **move views into a more powerful environment** instead of **moving javascript into a less powerful one**.
- HTML elements are very restrictive. They can only communicate between them self by text or numbers through attributes.

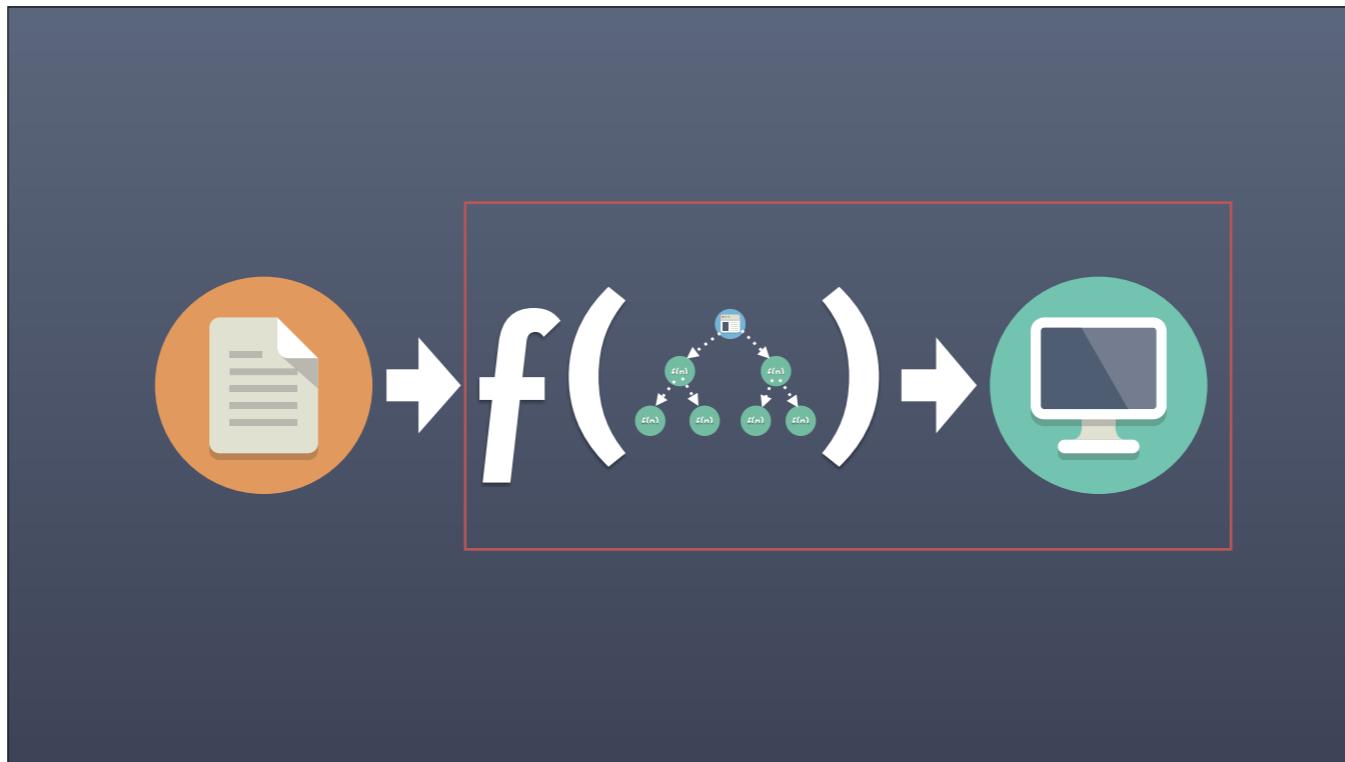
```
const MyApp =({children}) => (
  <article>
    <Header>Hello, {children}!</Header>
  </article>
);
```

- With components we can communicate with much clearer intent and advanced structures: We can pass objects.
- With the way javascript works, this means we can pass functions and lists as well. **Which is very powerful.**
- Vi kan ha mange komponenter nedover. Og siden de alle kommuniserer som enkle funksjoner, vil det ettehvert kunne bli et stort tre av komponenter.

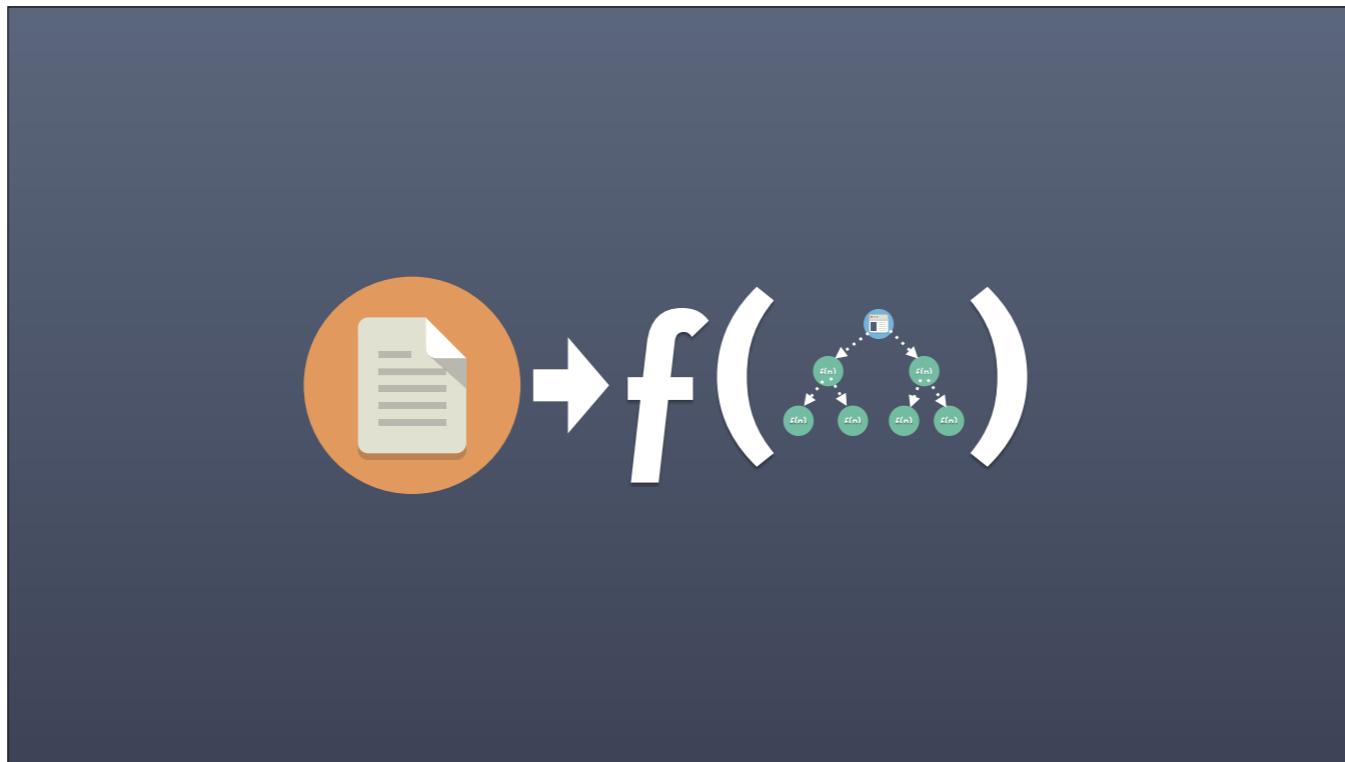
# PREDICTABLE

```
render(<MyApp>Robert Bruce Banner</MyApp>, el);
```

- And of course, since our system is pure and referentially transparent, the output would be predictable and testable.
- **We can do the rendering multiple times with the same input and the system wouldn't actually do any DOM operations.**
- Og React tar seg av all den oppdateringen til DOM-en, så vi trenger ikke tenke på det. Vi bare vet hva systemet vårt skal ha som output, så fikser React at det blir output på en optimal måte!

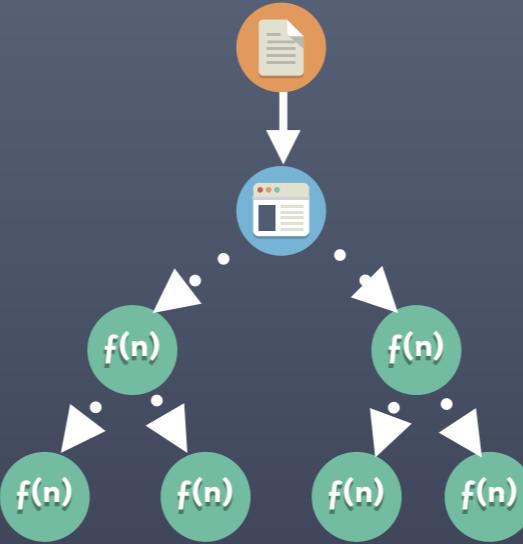


- Nå har vi sett på output delen av systemet vårt. Hvordan vi bare kan returnere et resultat og deletere bort ansvaret for rendring.

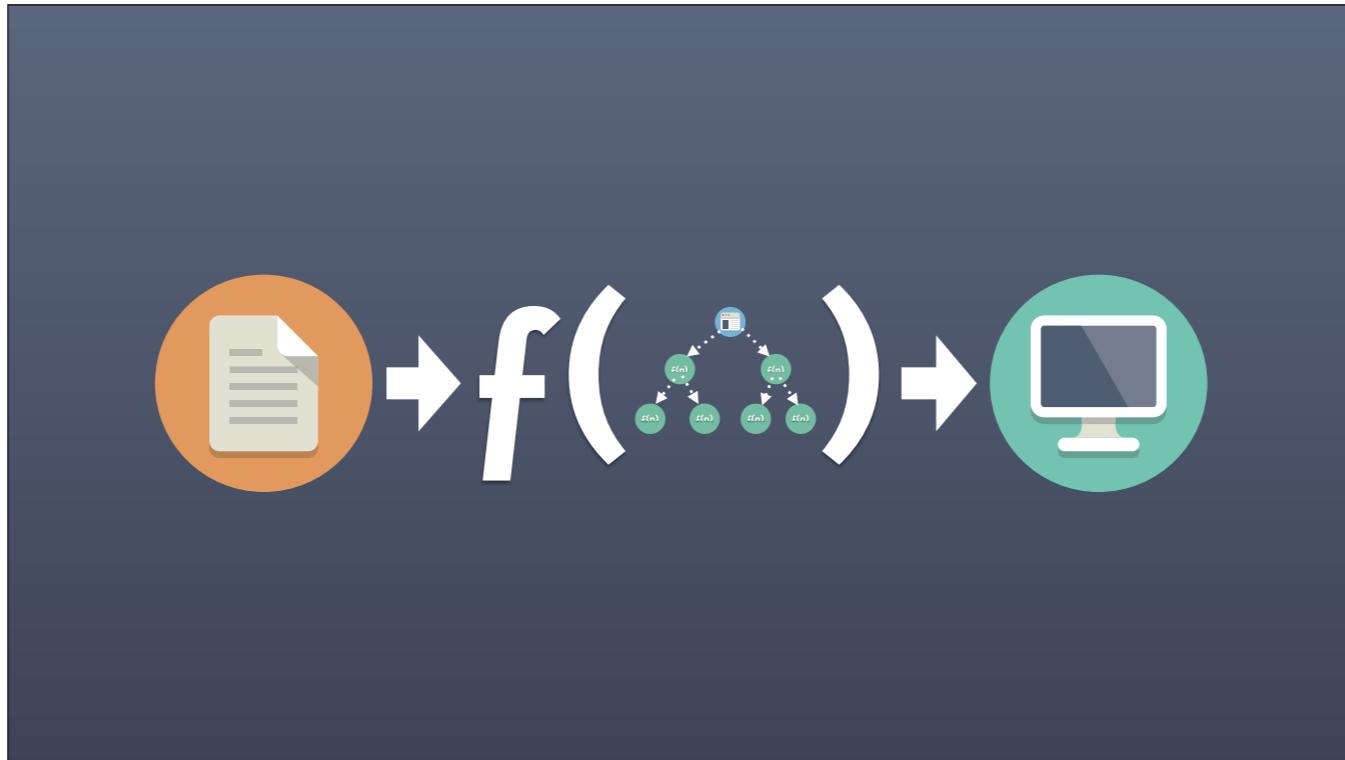


- Men hva betyr egentlig det at vi sender en eller annen tilstand inn til systemet vårt?

# GLOBAL STATE

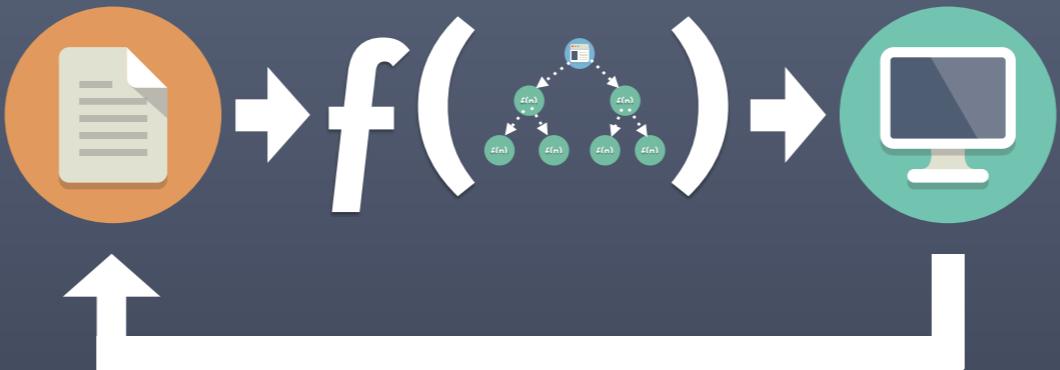


- Jo, vi har all mulig data som hele systemet vårt trenger utenfor selve systemet. Via en såkalt global tilstand.
- Vi sender den inn til entry-pointet av applikasjonen vår, så får den videre distribuere den nedover til forskjellige deler som trenger informasjonen.
- Ofte vil vi få en datastruktur som ikke er så veldig ulik i form som det strukturen av komponenter er.



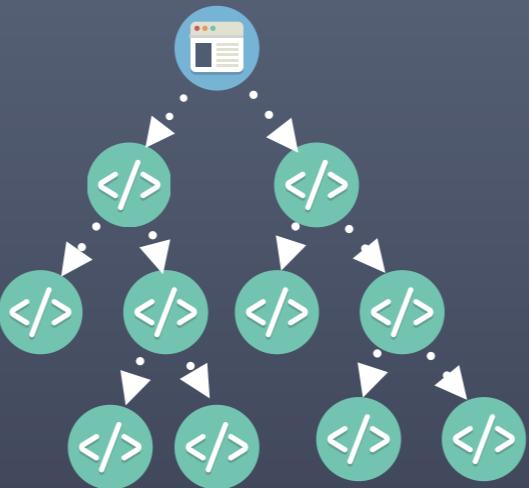
- Så slik kan vi se for oss at flyten til programmet er
- Sender inn HELE tilstanden til toppnivået av komponentene våre som egentlig bare er en funksjon.
- Den kaller alle sine komponenter som funksjoner nedover, helt til den siste returneres og resultatet propagerer hele veien opp og er ferdig med et resultat.
- Resultatet, hele DOM-representasjonen vi har laget kan vi bare sende videre til output og vi er egentlig ferdig med det.
- Kommunikasjonen går en vei, som betyr at det så og si alltid vil være enkelt å følge flyten til applikasjonen.
- Men hva med å oppdatere ting? hva skjer dersom vi klikker på en knapp, endrer på et inputfelt, får et resultat fra et AJAX-kall?

# ALWAYS RERENDER ENTIRE APP



- Hver gang vi gjør en endring så kan vi bare kalle hele funksjonen på nytt!
- Så vi bare sender en funksjon opp til på toppen som kjører hele greia på nytt igjen.
- Så det er bare en vei kommunikasjonen går.
- Noe å oppdatere? Kjør systemet på nytt med den nye tilstanden!
- Dette er stort sett det samme som en full refresh i nettleseren!

# ISN'T THIS SLOW?



- Men er ikke dette litt langsomt, kan du tenke nå?
- Selv om React tar en diff på outputen så er det fortsatt mye arbeid som kan skje i selve view komponentene vår til ingen nytte.

# REMEMBER: IMMUTABLE MAPS

```
const obj = Immutable.Map({ name: "T'Challa" });
const obj2 = obj.set('name', 'Black Panther');
obj !== obj2;
obj === obj;
```

- Om vi husker fra tidligere: Om INPUT var det samme, blir output det samme!
- Og om vi da husker at vi bruker immutabel data der vi kan sjekke på enkle referanser av objekter, kan vi prøve å være litt smarte.

```
function fibonacci(n) {  
    if (n === 0 || n === 1)  
        return n;  
    else  
        return fibonacci(n - 1) + fibonacci(n - 2);  
}
```

- Feks. Hva ville vi ha gjort dersom vi ønsket å ha bedre ytelse på en fibonacci-funksjon?

# MEMOIZATION

```
function fibonacci = memoize(  
  function(n) {  
    if (n === 0 || n === 1)  
      return n;  
    else  
      return fibonacci(n - 1) + fibonacci(n - 2);  
  }  
);
```

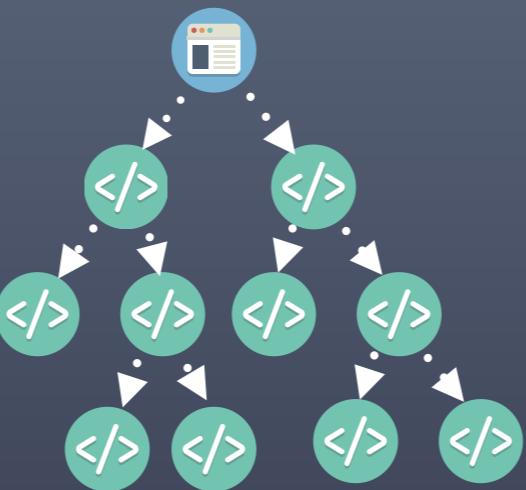
- Jo, vi ville ha lagt på memoization.
- Hva er det? Den sjekker bare på input og ser om det input er blitt sendt tidligere, om den har det så kan vi bare sende resultatet av forrige gang den bli kalt.

# MEMOIZATION

```
const MemoizedComponent = memoize(SomeComponent);
```

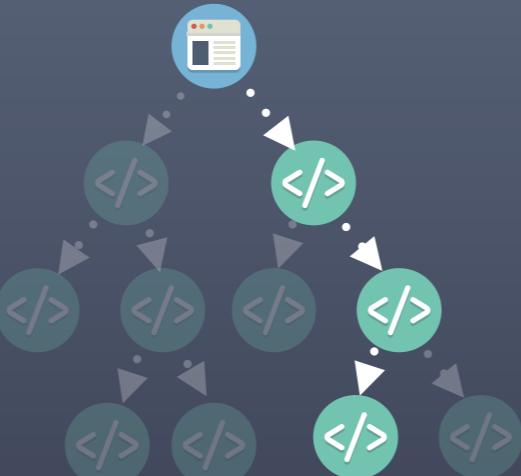
- Akkurat det samme kan vi gjøre med komponentene våre!
- Dersom den har blitt kalt med samme input før, så ikke prøv å kjør den igjen.
- Vi vet jo at samme input betyr samme output. Ingenting burde ha endret seg.

# SMART TREE UPDATE



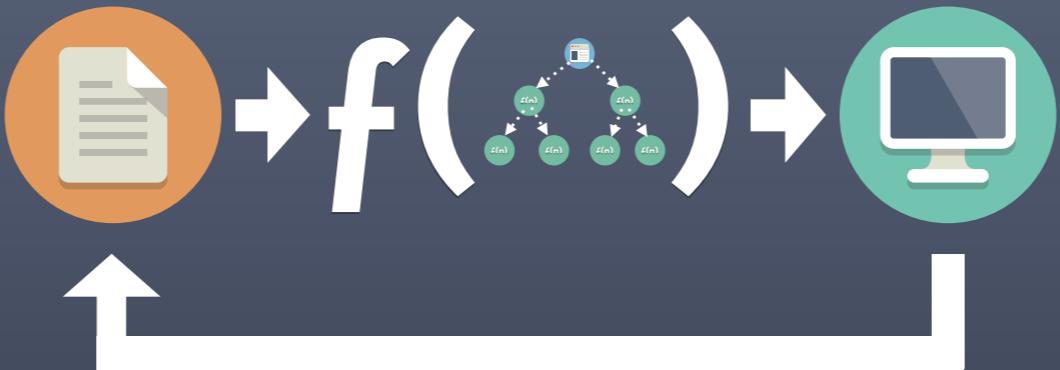
- Om vi ser på det som et oppdatering av et tre som er applikasjonen vår.

# SMART TREE UPDATE



- Vil vi se at det er kun pathen til den komponenten som har endret seg som faktisk blir prøvd å endret.
- Da sparar vi veldig mye arbeid og vi kan gjøre mange, mange operasjoner i sekundet.
- I tillegg har vi jo output diffinga til React som et ekstra lag av sikring mot ekstra arbeid. Den gjør at vi ikke gjør for mange DOM-operasjoner, mens dette gjør at vi unngår en del ekstra arbeid generelt.

## GAME LOOP OF THE APP



- Så da sitter vi igjen med noe som ligner mye på en game loop. Det er bare en render loop som kjører over og over igjen per oppdaterte endring.
- Det gjør at vi kan lese systemet fra toppen til bunnen og skjønne hva output blir uten egentlig å sette oss inn i hele systemet. Vi begrenser den mentale kapasiteten vi har behov for adskillig.

# STATE OVER TIME



- By sending messages to our global state, we get kind of revisions between changes.
- Every time something in our system changes, we get a new iteration of the top structure.
- **(next)** This means we can step through our system state or react to a error by undoing an update.

# STATE OVER TIME

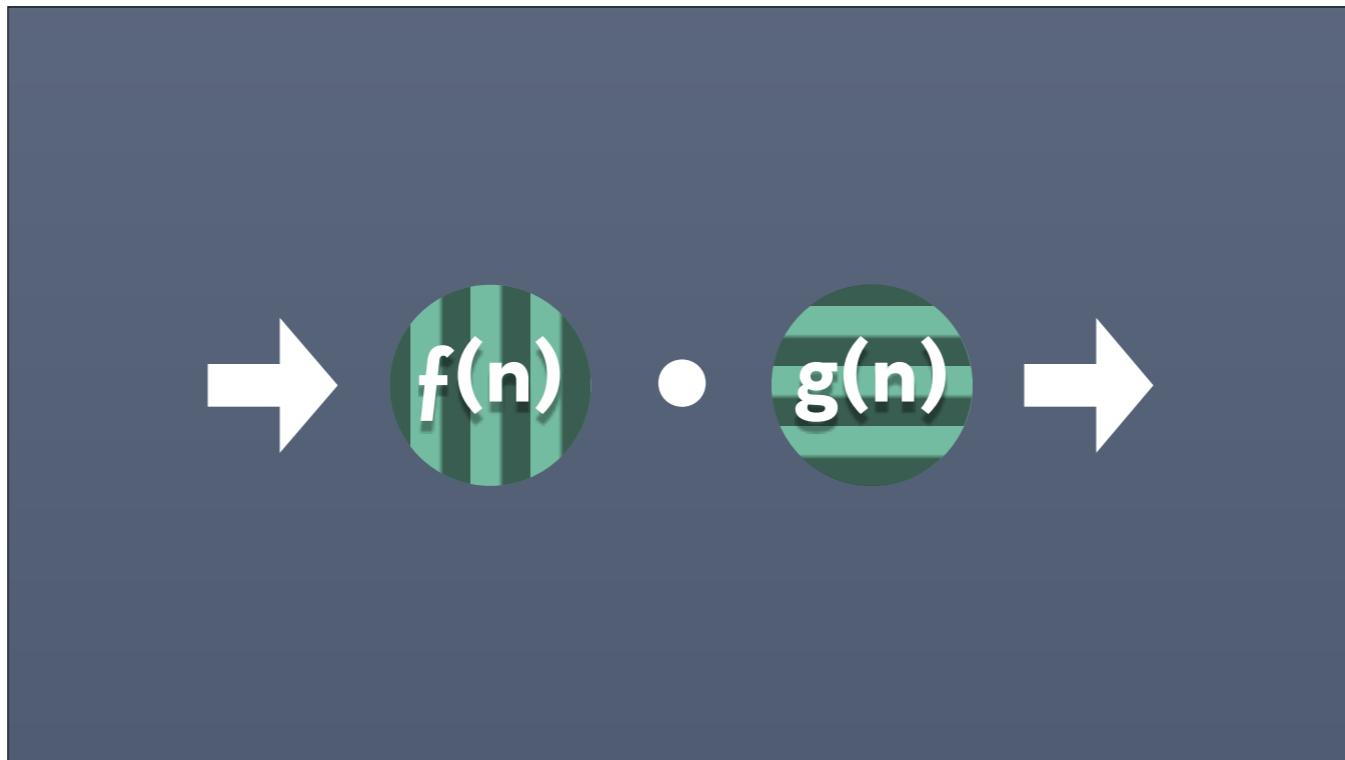


- We could even build test tools that **instead of having to click through our application**, or use tools like selenium, we can just **iterate over a history and check the complete output**.
- Another use might be to do something like serialisation on error, and logging the application state. **Making it very easy to reproduce any error and fixing bugs.**

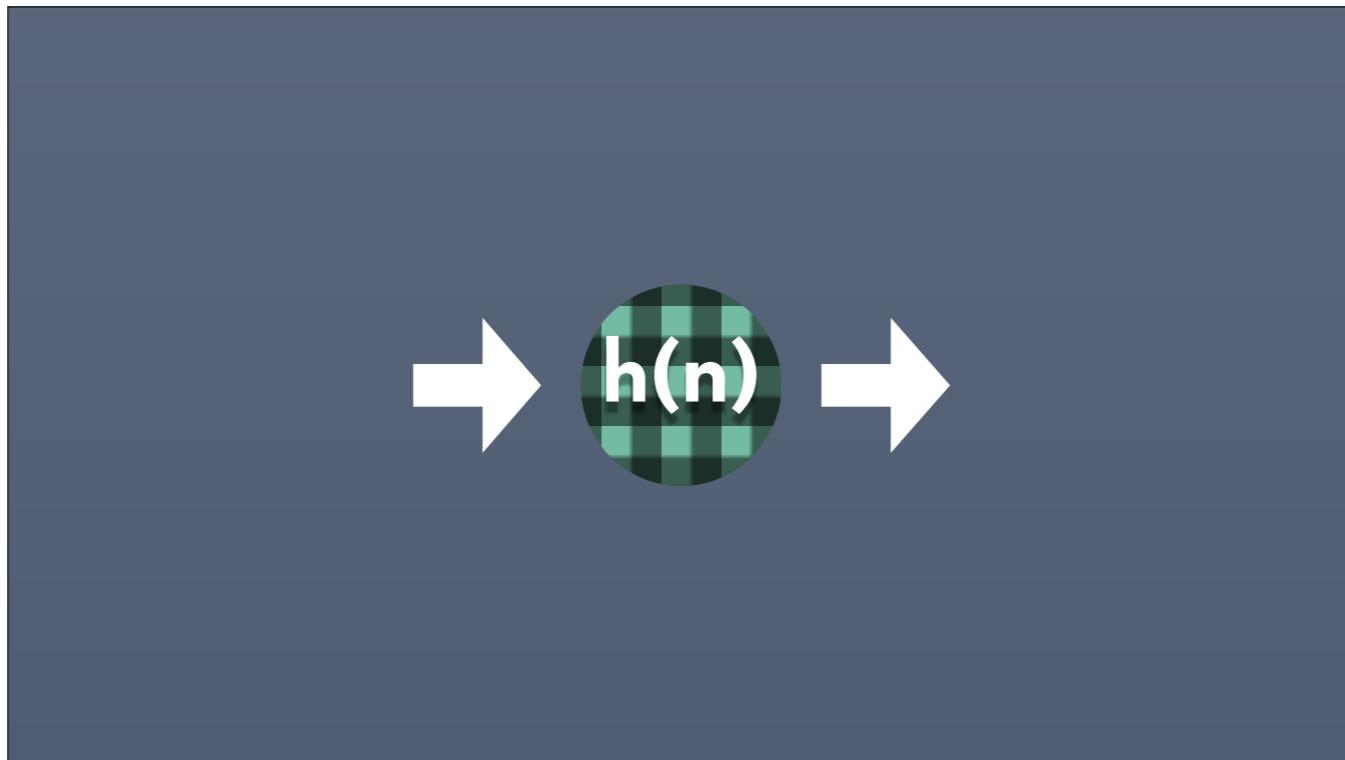
# FUNCTIONAL PROGRAMMING: COMPOSABILITY



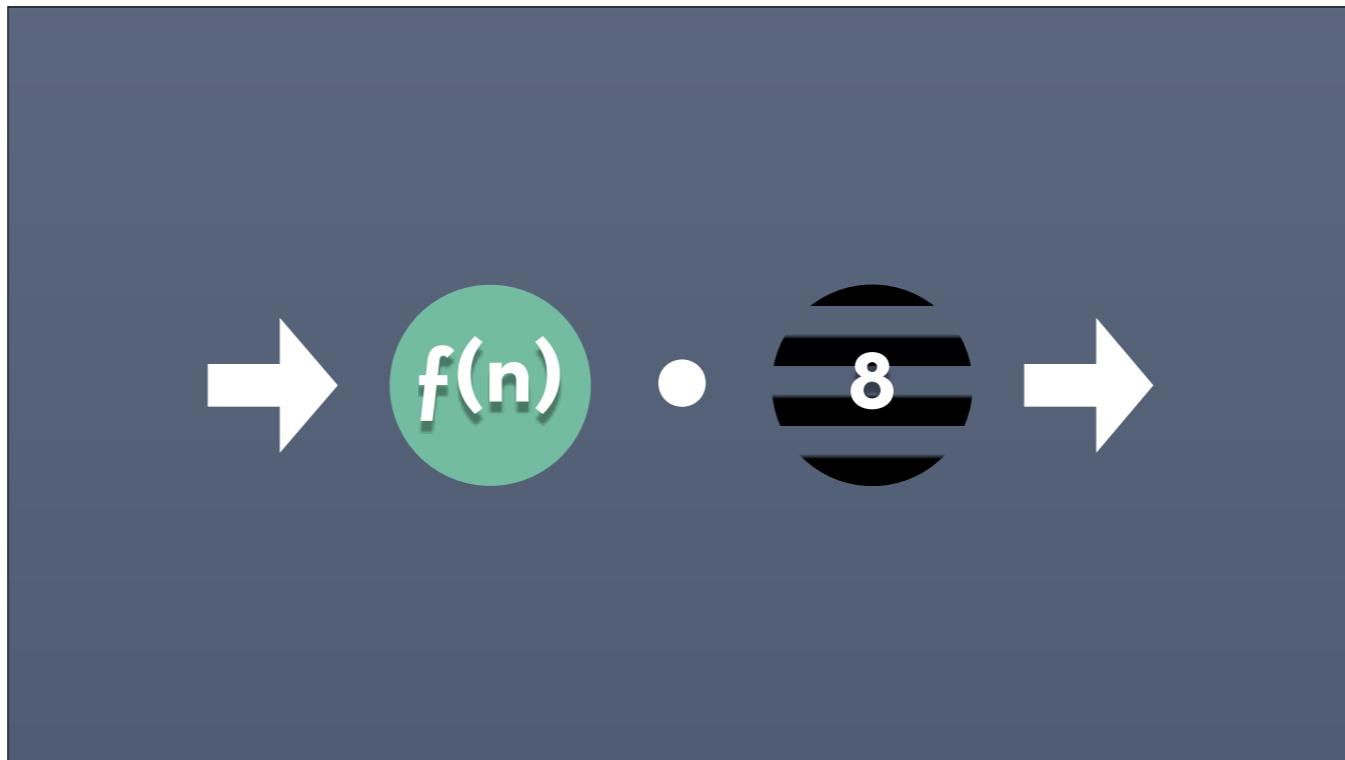
- OK! Da har vi på en måte bygd opp til at vi kan tenke funksjonelt, og vi har snakket om noe fancy ord som referential transparency, purity, structural sharing, tre.
- Men alt dette bringer også opp at vi kan bruke mange flere funksjonelle kode patterns som vi tidligere ikke kunne ha brukt.
- Vi benytter anledningen til å introdusere et nytt fancy begrep: composability.



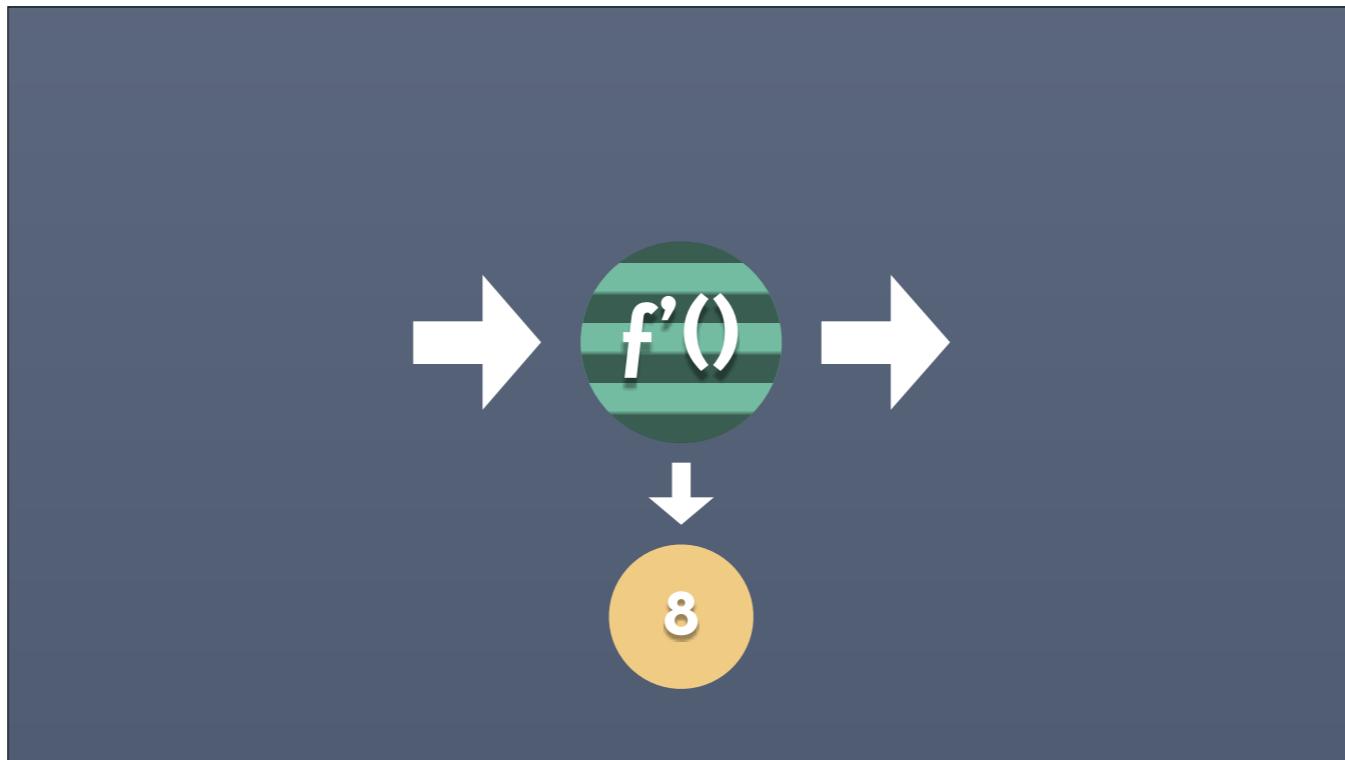
- For eksempel, vi kan ta to funksjoner som har samme input-format/type (**next**)



- og compose de til en ny funksjon. Vi kan si at vi deriverer funksjonen og lager en ny en som har kombinert funksjonalitet fra de to byggeklossene.
- **Compose her er det vi kaller en høyere ordens funksjon!**
- Composability of functions can **allow us to build large applications but by combining simple pieces that are easy to understand**.



- En annen måte å bruke composability på, er å gjøre noe som kalles partial applications.
- **(next)**



- Vi kan lage en ny funksjon av en annen funksjon som tar en eller færre input argumenter enn den originale funksjonen.
- Dette er ganske nyttig for å gradvis bygge opp funksjoner med mer spisset funksjonelitet.

# (HIGHER ORDER) COMPONENTS



- Alt av composability kan vi og nå bruke på komponentene våre. De er jo rene og har referential transparency.

# COMPOSABLE COMPONENTS

```
const Header = ({children}) => (...);  
const Italic = ({children}) => (...);  
const ItalicHeader = compose(Header, Italic);
```

- På samme måte som vi kan smelte sammen to funksjoner, kan vi også gjøre det med komponenter.
- Dette er populært blitt kalt høyere ordens komponenter!
-

```
const MyComponent =({children}) => (
  <article>
    <ItalicHeader>Hello, {children}!</ItalicHeader>
  </article>
);
```

- Ved å definere en mer spisset komponent av to mer generelle, får vi frem mye mer tydelig mening i hva komponentene gjør.

```
const Herolist =  
  ['Wasp', 'Falcon', 'Harry Potter']  
    .filter(isSuperHero)  
    .map((title) =>  
      <MyComponent>{title}</MyComponent>  
    );
```

- Vi kan og bruke innebygde høyere ordens funksjoner som skal operere på komponenter. Dette er ofte kalt kombinatorer.
- Vi kan deklerativt og veldig funksjonelt, transformere en liste av navn til en liste av superhelter og så til en liste av komponenter av superhelter.

# COMPOSABLE COMPONENTS

```
const Header =({type, children}) => (
  <h1 className={type}>{children}</h1>
);
const MainHeader = partial(Header, 'main-title');
const PostHeader = partial(Header, 'post-title');
```

- We can also partially apply components using higher order components.
- By partially applying some bit of information onto a Component we can create a more semantically clear component which is more focused on it's goal and has a clearer intent.
- Components can work exactly like functions. **State-less pieces of code which transforms some input, producing an output.**

# COMPOSABLE COMPONENTS

```
const MainHeader = partial(Header, 'main-title');
const ItalicHeader = compose(MainHeader, Italic);
const FailSafeItalicHeader = maybe(ItalicHeader);
```

- We can of course also use other functional building tools.
- By applying different functional patterns, we can take just **two very simple components, deriving a new component that has more functionality, just by adding small pieces together.**
- And we can still use the individual components separately in different parts of our system.

```
const MainHeader = partial(Header, 'main-title');
const ItalicHeader = compose(MainHeader, Italic);
const FailSafeItalicHeader = maybe(ItalicHeader);
```

```
render(
  <FailSafeItalicHeader>
    Robert Bruce Banner
  </FailSafeItalicHeader>, el);
```

- Og vi kan bruke dette på akkurat den samme måten som vi hadde definert tidligere.
- Vi kan sende resultat outputen til React som håndterer synkningen til DOM.
- Men som tidligere, er det en ting å merke seg her: Vi oppdaterer DOM-en til å representerer vår sannhet, og bruker ikke DOM-en som en sannhet.

```
render(  
  <FailSafeItalicHeader>  
    Robert Bruce Banner  
  </FailSafeItalicHeader>, el);
```

```
render(  
  <FailSafeItalicHeader>  
    Robert Bruce Banner  
  </FailSafeItalicHeader>, el);
```

```
render(  
  <FailSafeItalicHeader>  
    Robert Bruce Banner  
  </FailSafeItalicHeader>, el);
```

- And of course, since our system is pure and referentially transparent, the output would be predictable and testable.
- **We can do the rendering multiple times with the same input and the system wouldn't actually do any DOM operations.**

- FUNCTIONAL PARADIGMS IN UI
- SYSTEM AS A FUNCTION OF STATE
- GLOBAL ALL-KNOWING APP STATE
- RENDER LOOP AND STATIC MENTAL MODEL
- COMPOSABLE (HIGHER ORDER) COMPONENTS

- To summarize
- We've seen **how we can use patterns from functional programming to program views**, what is **often thought of as a poor fit due to the stateful nature of the DOM**.
- We've seen how we can think of a **system as a function of state**, where we have a global state at the top of the application, and using a render loop to achieve a **static mental model** much like we had with good old HTML - **but with the power and flexibility of programming**.
- By using **composable components and functions**, we can build large systems by simple pieces and each piece doesn't have to know about the rest of the system **in any way**.



- All of these ideas can be realised by using React, or any other virtual DOM implementation.
- My hope is, and it seems it is heading that way, is for React to get core support for state-less component functions just as seen in this presentation, but for now it's not as straight forward.
- This is why we created a very small syntactic sugar on top of React and a “smart” logic to check whether or not a component should update, which encourages these patterns of view programming.
- If this talk was interesting to you, check the Omniscient project out and help improving on it's ideas or just see the reading material we have.

## **OMNISCIENT.JS**

THIS ARCHITECTURE AS A HELPING LIBRARY  
[HTTP://OMNISCIENTJS.GITHUB.IO/](http://omniscientjs.github.io/)

## **IMMUTABLE DATA STRUCTURES**

VIDEO ON HOW IMMUTABLE.JS WORKS  
[HTTPS://WWW.YOUTUBE.COM/WATCH?V=I7IDS-PBEGI](https://www.youtube.com/watch?v=I7IDS-PBEGI)

- If you are interested in learning more, these are some links you can check out.
- I didn't get much into how immutable data structures work in this talk, but you should check out Lee Byrons talk from React Conf for more information.
- You should also check out the ClojureScript library Om, a library Omniscient has built most of it's core ideas from.

## SIMPLER UI REASONING

ARTICLE FORMAT OF THIS PRESENTATION

[HTTP://OMNISCIENTJS.GITHUB.IO/GUIDES/01-SIMPLER-UI-REASONING-WITH-UNIDIRECTIONAL/](http://omniscientjs.github.io/guides/01-SIMPLER-UI-REASONING-WITH-UNIDIRECTIONAL/)

## FUNCTIONAL PATTERNS FOR VIEWS

ARTICLE ON FUNCTIONAL PATTERNS IN FOR VIEWS

[HTTPS://BLOG.RISINGSTACK.COM/FUNCTIONAL-UI-AND-COMPONENTS-AS-HIGHER-ORDER-FUNCTIONS/](https://blog.risingstack.com/functional-ui-and-components-as-higher-order-functions/)

## OM

IMPLEMENTATION OF THIS ARCHITECTURE IN CLOJURESCRIPT

[HTTPS://GITHUB.COM/OMCLJS/OM](https://github.com/omcljs/om)

- If you are interested in learning more, these are some links you can check out.
- I didn't get much into how immutable data structures work in this talk, but you should check out Lee Byrons talk from React Conf for more information.
- You should also check out the ClojureScript library Om, a library Omniscient has built most of it's core ideas from.

# FUNCTIONAL PROGRAMMING IN VIEWS

twitter @mikaelbrevik  
github @mikaelbr



Spørsmål?