

Introduction to React and the Flux Architecture

Presented by Søren Engel

Frontenders Copenhagen Meetup - May 26, 2015

Hi everybody!



- My name is Søren Engel
- I have a background in Software Development from the IT-University of Copenhagen
- I'm currently working with front-end development at Netcompany A/S
- Twitter account: @soren_engel

Agenda

- Introduction to React
- The React Ecosystem
- The Flux Architecture
- Putting it all together (demo)
- Where do I go from here?
- Take aways



Introduction to React

- A brief history of React
- Is React a Framework?
- The Virtual DOM Concept in a Nutshell
- Component-Driven Development
- JSX
- Properties (props) and State
- Lifecycle and Methods

A brief history of React

- Developed by Facebook, Instagram and a large community
- First version landed back in 2013
- Purpose
 1. Performance
 - Challenge the common mantra in the JavaScript community: "DOM manipulation is expensive" – In other words, taking your JavaScript application data and "rendering" it into a browser is a costly operation
 2. Simplify reasoning about the UI
 - One-way (unidirectional) data flow
 3. Make UI components reusable

Is React a Framework?

- The short answer, No!
- React is a **library** for creating user interfaces (no bells and whistles)
- Think of React as the “**View**” in MVC (a blazing fast one!)
- React is **mainly a concept** and a library just secondly
 - Focus on **breaking down** your application into smaller pieces (components)
 - Components are **composed** together to form larger pieces
 - Components are **reusable**

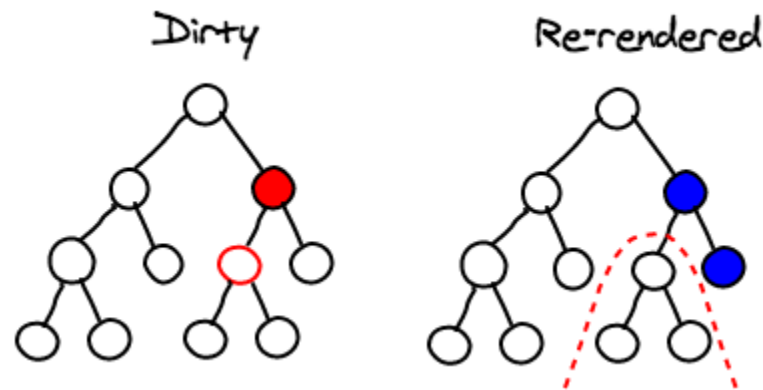
The Virtual DOM Concept in a Nutshell

- To track down model changes and apply (render) them to the DOM, we need to be aware of:
 1. When data has changed
 2. Which DOM element(s) should to be updated
- Change detection (1) is solved by using an **observer model**, rather than dirty checking
- For the DOM changing challenge (2), React builds the **tree representation** of the DOM (The Virtual DOM) in the memory and calculates which DOM element should change
- This makes React blazing fast!

The Virtual DOM Concept in a Nutshell

React's diffing algorithm

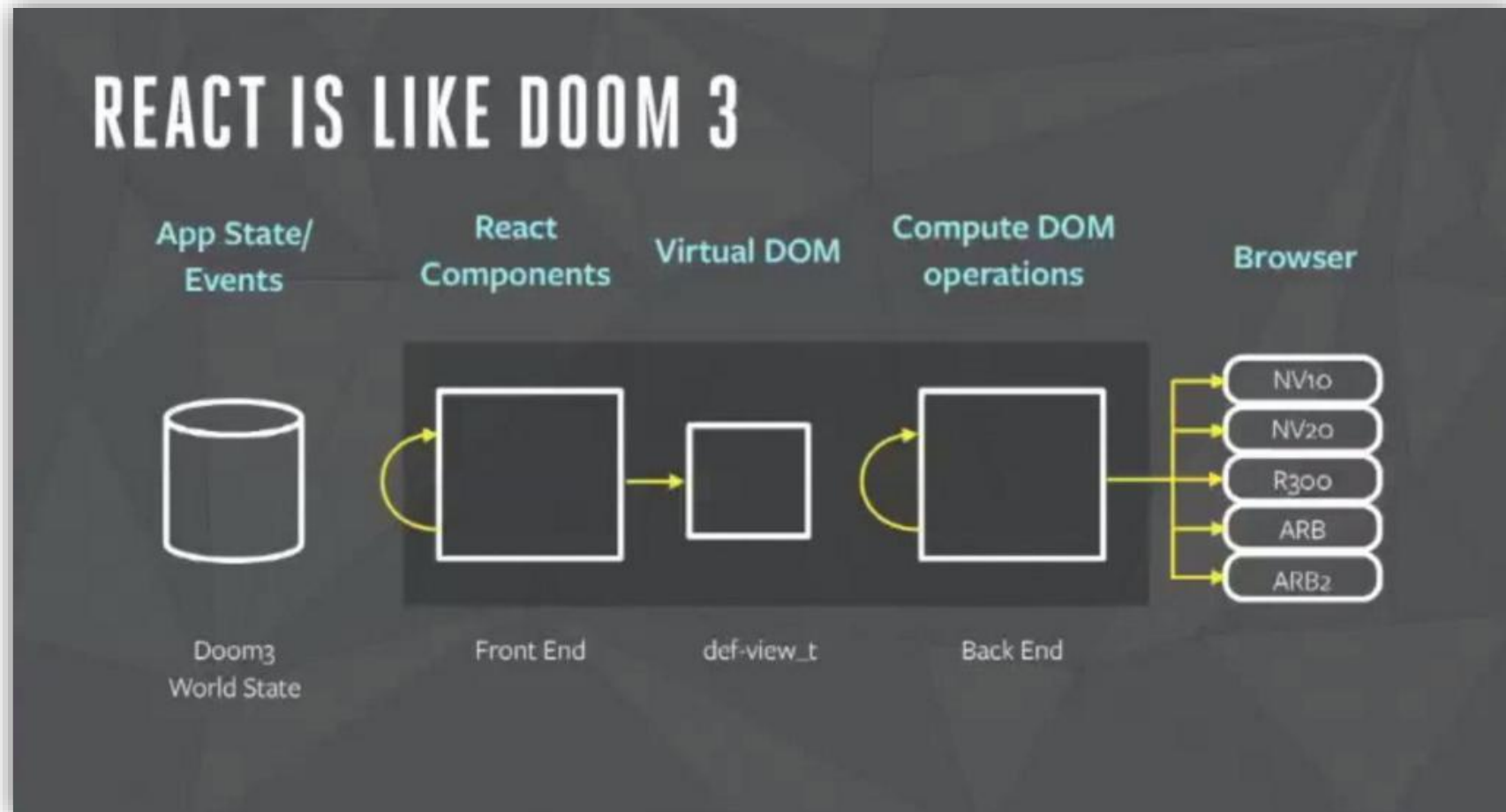
- Uses the tree representation of the DOM
- Re-calculates all subtrees when it's parent get modified (*marked dirty*)
- When the model changes, only the subtree affected will be re-rendered to the DOM



(source: [React's diffing algorithm](#))

The Virtual DOM Concept in a Nutshell

Comparing React to Doom 3



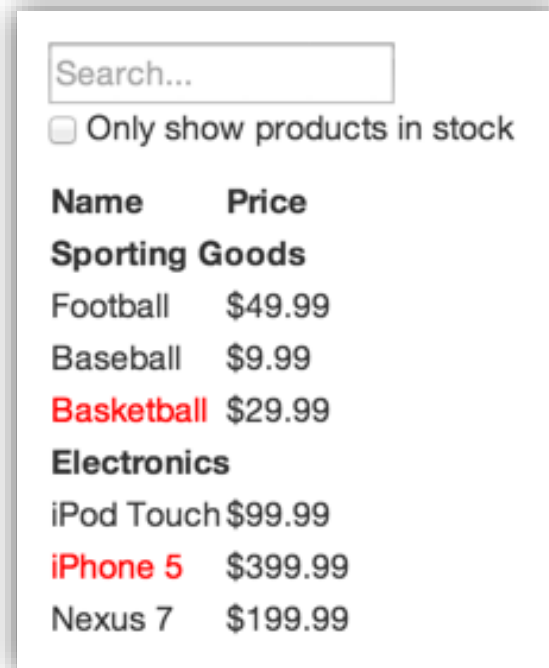
(source: [React: Rethinking Best Practices](#))

Component-Driven Development

- Most difficult part to pick up, when learning React
- Thinking in Components, not Templates
 - In React you won't see the whole thing in one template (like site templates)
- The power of thinking in smaller pieces and work with less responsibility
- Separation of concerns (Low coupling, strong cohesion)
- Easier to **understand**, **maintain** and to **cover with tests**

Component-Driven Development

How should I picture it?



A mockup of a web application interface. At the top is a search bar with the placeholder text "Search...". Below the search bar is a checkbox labeled "Only show products in stock". The main content area displays a list of products under two category headers: "Sporting Goods" and "Electronics". Each product entry consists of a name and a price. The product names "Basketball" and "iPhone 5" are highlighted in red.

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

- It all starts with a mock
- Which is then **broken into a component hierarchy...**

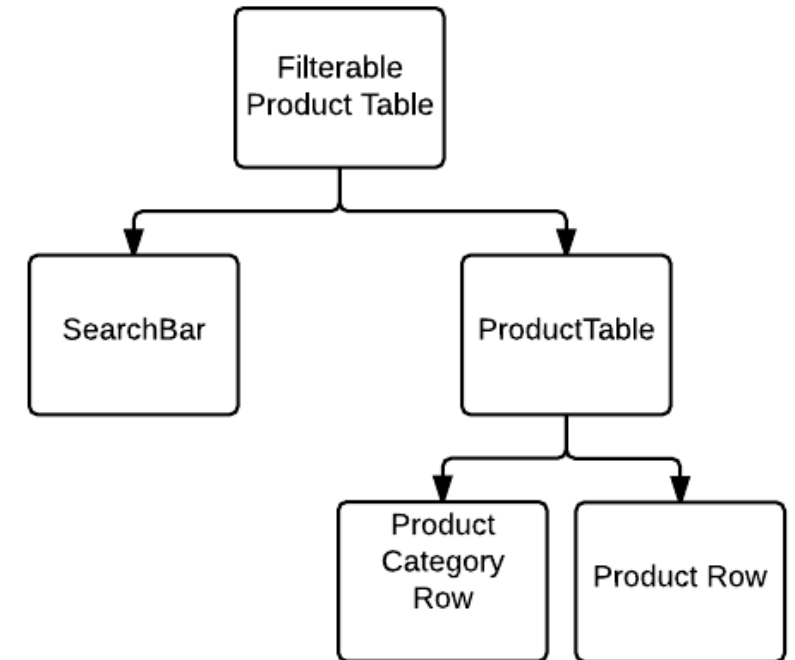
Component-Driven Development

How should I picture it?

- Each of the bordered areas with different colors represents a single type of component
- According to this, you have the component hierarchy, shown on the right side

The image shows a UI component for a product table. It is divided into several colored bordered areas: a blue border for the search and filter section at the top, a green border for the entire table area, and red borders for individual table rows. The table has two columns: 'Name' and 'Price'. It lists 'Sporting Goods' and 'Electronics' as categories, with specific products like 'Football', 'Baseball', 'Basketball', 'iPod Touch', 'iPhone 5', and 'Nexus 7' listed with their prices.

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99



(source: [The React.js Way – Getting Started Tutorial](#))

Component-Driven Development

What should a component contain?

- A component should contain what it's responsible for, and nothing more
 - **Example:** *The SearchBar component should contain all that is needed for the search bar to work, no more or less*
- Design components according to the Single Responsibility Principle
(remember: emphasis on separation of concerns!)
- If components get to big, consider breaking them down into smaller ones
- Components may contain other components (composability)

JSX

- An inline markup that looks like HTML and gets transformed to JS
- A *declarative* syntax that's used to express the **Virtual DOM**
- JSX expressions evaluate to ReactElements (React's building blocks)
- JSX supports
 - JavaScript Expressions
 - Inline Styles
 - Events (synthetic)

JSX

A small example

ES5 (old syntax*)

```
var HelloComponent = React.createClass({  
  render: function() {  
    return <div>Hello {this.props.name}</div>;  
  }  
});
```

ES6 (new syntax)

```
class HelloComponent extends React.Component {  
  render() {  
    return <div>Hello {this.props.name}</div>;  
  }  
}
```

**This version will be deprecated in future versions*

JSX
Wait, what?!



JSX

Components are a mix of JS and HTML code?

- Yes, probably you think it's strange...
- The idea here is to have everything in one place – ***hint***: remember single responsibility
- This makes a component extremely flexible and reusable
- Won't this be spaghetti code? – Just don't write spaghetti code!
 - Keep your components small
 - Only put display logic into your components

Properties and State

- Components work with properties (or props) and state
- **Properties:**
 - They're the data that get passed into the component as **element attributes**
 - Accessed via **this.props**
- **State:**
 - A component can maintain **internal state data**
 - Accessed via **this.state**
- When a component's state data changes, the rendered markup will be updated by re-invoking `render()`

Properties and State

Small examples

```
class HelloMessage extends React.Component {  
  render() {  
    return <div>Hello {this.props.name}</div>;  
  }  
}
```

```
class Counter extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { count: props.initialCount };  
  }  
  tick() {  
    this.setState({ count: this.state.count + 1 });  
  }  
  render() {  
    return (  
      <div onClick={ this.tick.bind(this) }>  
        Clicks: { this.state.count }  
      </div>  
    );  
  }  
}  
Counter.defaultProps = { initialCount: 0 };
```

Lifecycle and Methods

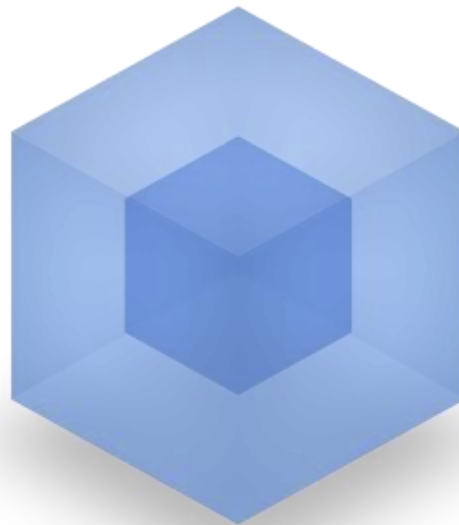
- Each component has a "lifecycle"
- It's possible to hook into different parts of the lifecycle, using:
 - **componentWillMount**
 - Called when the component is about to be mounted
 - We can run code that is necessary to our component functioning
 - **componentDidMount**
 - Called when the component has been mounted (rendered to the screen)
 - We can do DOM manipulations or anything that relies on component actually being in the DOM
 - **componentWillUnmount**
 - When a component is removed from the DOM, this function is called
 - Allows us to clean up after the component, such as removing any event listeners that we've bound

Lifecycle and Methods

- React offers methods for our components to make things easier
- These are called on the creation of the component
- **defaultProps**
 - define the default values for the properties of the component
- **propTypes**
 - specify the type of each property we are expecting

The React Ecosystem

- React + ES6 = ♥
- Tooling
 - Node.js
 - Babel
 - Browserify / Webpack



webpack
MODULE BUNDLER



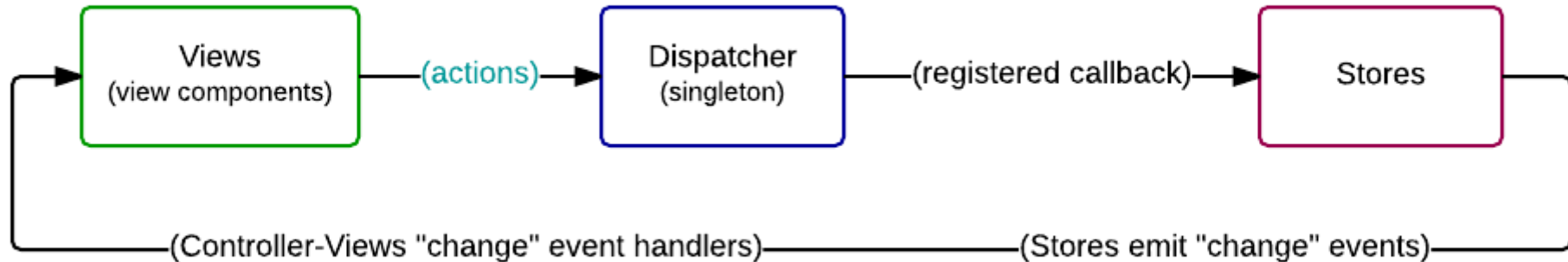
BABEL

The Flux Architecture

- An overview of Flux
 - View
 - Action
 - Dispatcher
 - Store
- Stateful vs. Stateless Components (or don't overfluxify)
- Supporting frameworks

An overview of Flux

- An Application architecture for building user interfaces
- Flux is a **unidirectional data flow** concept



(source: [The React.js Way – Getting Started Tutorial](#))

An overview of Flux

View

- Displays the data to the user
- Listens to the stores in order to get the data to display
- Re-renders itself completely, when the data change from the store is fired
- Passes down data to their child components
- Controller-Views
 - A special kind of view that listens for events broadcasted by the stores
 - Propagates changes top-down to it's children

An overview of Flux Action

- Every data mutation inside application occurs with calling the *Actions*
- Can be user interaction inside component or server response
- Actions can be both synchronous and asynchronous

An overview of Flux Dispatcher

- The central hub that manages all data flow in a Flux application
- A registry of callbacks into the stores with no intelligence of its own
- A simple mechanism for distributing the actions to the stores
- The “Traffic Cop” that ensures only one thing is done at the time
 - **Important:** Throws if you try to dispatch while already dispatching...

This is quite annoying when first starting out using React, but usually a sign that you are doing something wrong!

An overview of Flux Store

- Contain the application state and logic
- They do not represent a single record of data like ORM models do
- They manages the state for certain **domain** within of our application
- Listens to the actions and **mutate** the data within themselves
- State is not changed anywhere from outside the store
- Stores **do not** communicate directly with other stores
- Important to remember that stores are not models, they contain models

Stateful vs. Stateless Component

(or don't overfluxify)

- A lot of debate on when to use stateful vs. stateless components
- Stateful components are often accused for being an **anti-pattern**
- Not entirely true, since having stateful components are not all bad
- General rule of thumb:
 - **If you're putting business rules in components, you're doing it wrong**

Supporting Frameworks

- There are **many** (more than 15 and counting) frameworks supporting Flux
- To name a few:
 - Facebook's own
 - Reflux
 - Alt
 - Flummox (my favorite)
- Almost all libraries use Facebook's [Dispatcher](#) under the hood
- Check out [Flux solutions compared by example](#)

Putting it all together



Where do I go from here?

- [Facebook React](#)
- [Facebook Flux](#)
- [A collection of awesome React libraries, resources and shiny things](#)
- [React.parts – A catalog of React components](#)
- [My React List, by Dan Abramov](#)
- [JSX Looks Like An Abomination – But it's Good for You](#)
- [The Reactiflux Slack channel](#)
- [React Discuss](#)

Take aways

- React makes it possible to create components that are:
 1. Composable
 2. Reusable
 3. Maintainable (and testable)
- Since React makes no assumptions about the rest of your technology stack, it's easy to use it with existing back-end code bases
- The Flux architecture can be used make creating complex UI, that is simple to reason about
- React has a very active community, and there are a lot of components out there to use directly in your app

Thank you for listening!





**KEEP
CALM
AND
ASK
QUESTIONS**