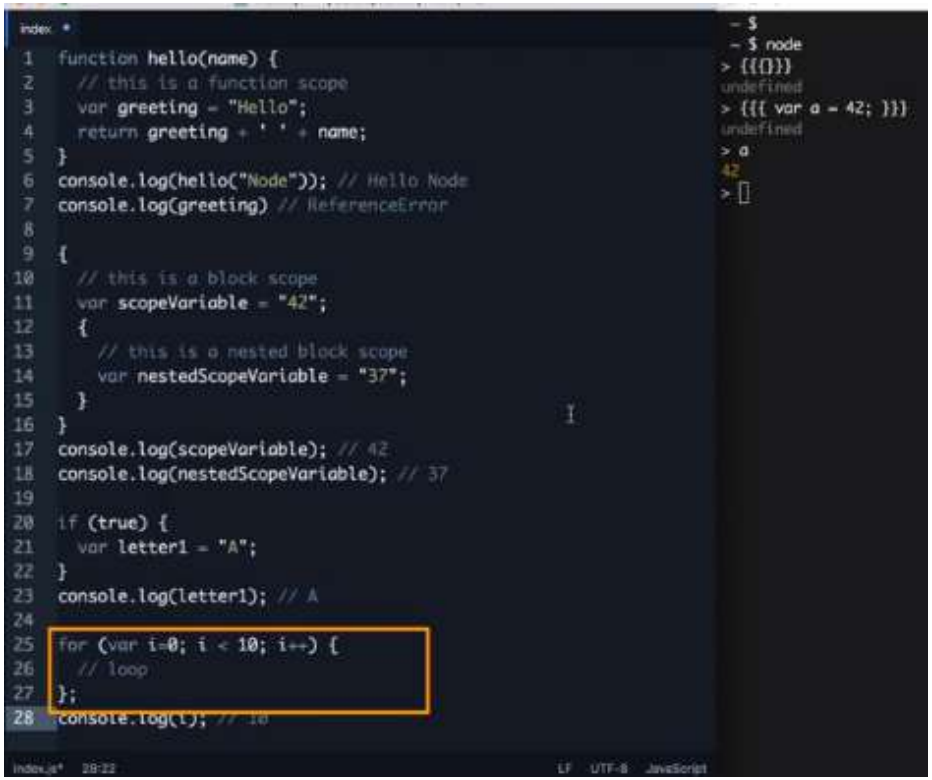
REACT FUNDAMENTALS

June2017

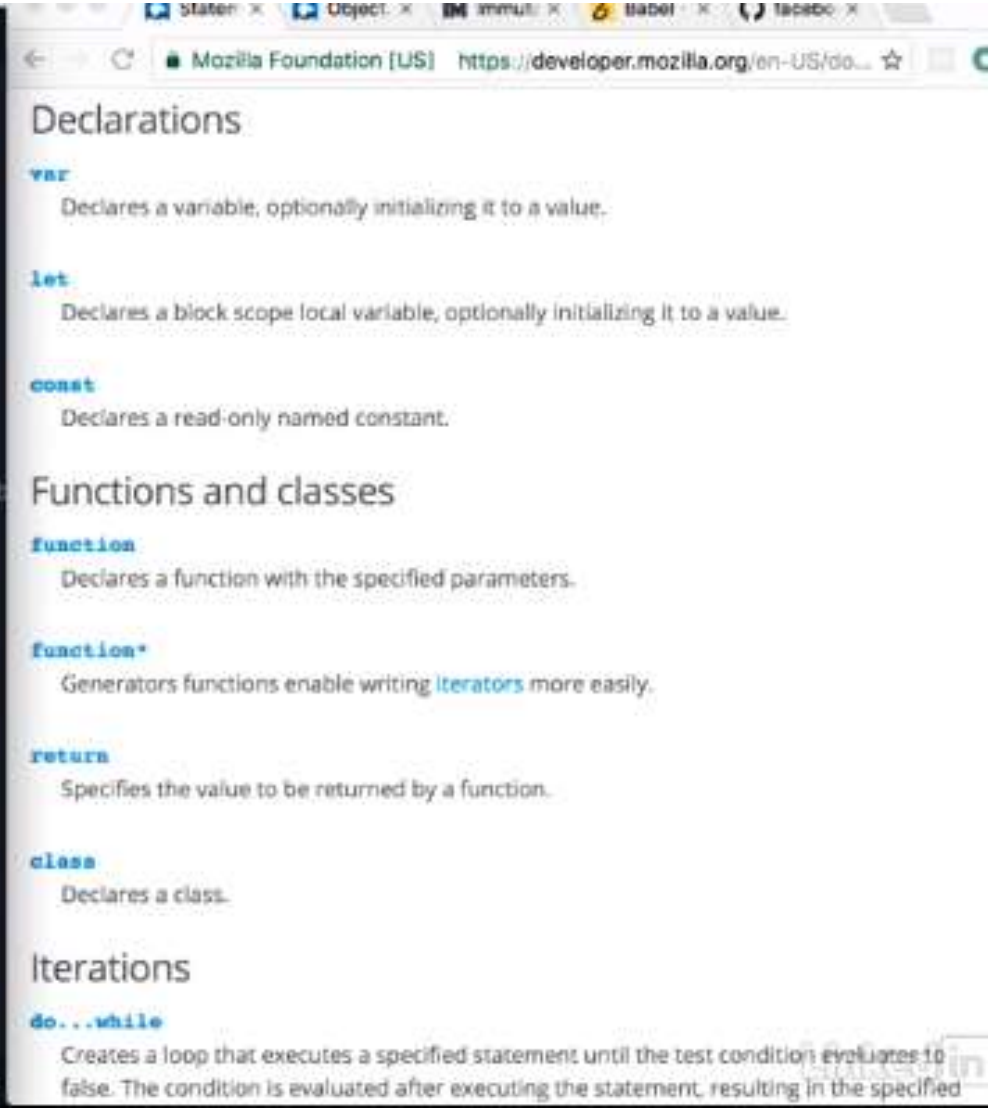
- In Javascript, we have a function scope and block scope.
- The function scope gets created every time we define a new function.
- Block scopes can be simple begin-end curly braces like or when an if statement, or a for statement.
- The var keyword behaves differently in a function scope and a block scope.
- A variable defined with var in a function scope can't be accessed outside the scope, while a variable defined with var in a block scope is available outside of that block scope.

- The i variable that we use in a for loop will continue to exist beyond the scope of the loop, which does not make sense.
- This is why, in modern JavaScript, we have a new keyword to define variables, **let**.
- Variables defined with let inside any scope are only accessible inside that scope, making let the ideal solution for the for loop index variable problem.
- If we use let to initialize the **i** variable in a for loop, the variable will only be defined inside the for loop.



MODERN JAVASCRIPT

```
index *
1  function hello(name) {
2    // this is a function scope
3    var greeting = "Hello";
4    return greeting + ' ' + name;
5  }
6  console.log(hello("Node")); // Hello Node
7  console.log(greeting) // ReferenceError
8
9  {
10   // this is a block scope
11   let scopeVariable = "42";
12   const scopeConstant = "37";
13   scopeConstant = "38" // TypeError: Assignment to constant variab
14 }
15 console.log(scopeVariable); // ReferenceError
16 console.log(scopeConstant); // ReferenceError
17
18 if (true) {
19   let letter1 = "A";
20 }
21 console.log(letter1); // ReferenceError
22
23 for (let i=0; i < 10; i++) {
24   // loop
25 };
26 console.log(i); // ReferenceError
```



Mozilla Foundation [US] https://developer.mozilla.org/en-US/do...

Declarations

var
Declares a variable, optionally initializing it to a value.

let
Declares a block scope local variable, optionally initializing it to a value.

const
Declares a read-only named constant.

Functions and classes

function
Declares a function with the specified parameters.

function*
Generators functions enable writing iterators more easily.

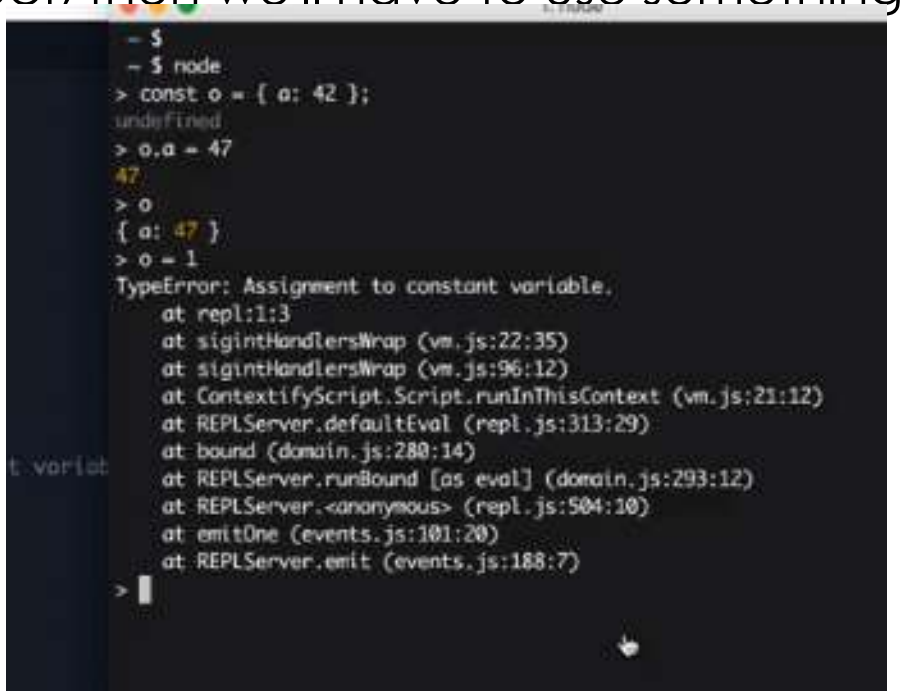
return
Specifies the value to be returned by a function.

class
Declares a class.

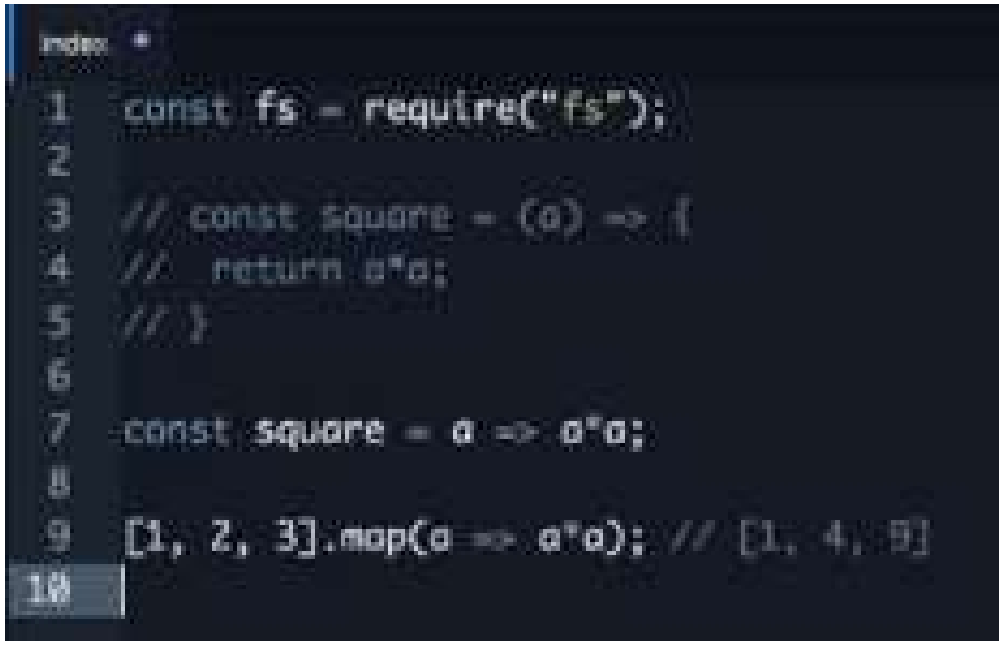
Iterations

do...while
Creates a loop that executes a specified statement until the test condition evaluates to false. The condition is evaluated after executing the statement, resulting in the specified

- Const is exactly like let, but defines a constant reference for a variable.
- We can't change the value of a constant reference.
- If we put a primitive value in a constant, then the value will be protected from getting changed.
- Note that if the constant is an object, we can still change the properties of the object, but we can't change the object itself.
- If we want a completely immutable object, then we'll have to use something else, like object.freeze or Immutable JS.



- Constants are popularly used when importing things from other libraries so that they don't get changed accidentally.
- They're also used to define functions because we rarely need to update a function after we define it the first time.
- Instead, we can use the new arrow functions and text, the arrow comes after the arguments.



- The arrow functions and text is even more concise when the function takes one argument that has one line to return something.
- We can omit the parentheses, the curly braces, and even the return keyword, and the function will be the same.
- These one-liner functions are usually used for callbacks in simple map reduced filter operations and functional programming in general.
- Also, when defining a function as a property on an object, we can use this new shorthand notation, even more than the arrow functions.

INTRODUCTION

- React's official definition states that it's a JavaScript library for building user interfaces.
- It's important to understand the two different parts of this definition.
- React is a JavaScript library, it's not a framework.
- The terms library and a framework mean different things in different contexts, but what we want to remember is that React is small and it's not a complete solution.
- We'll often need to use more libraries with React to form any solution.
- React does not assume anything about the other parts in any full solution, it focuses on just one thing and on doing that thing very well.
- The thing that React does really well is the second part of the definition, building user interfaces.
- A user interface is anything we put in front of users to have them interact with a machine.

- Core Technologies used

Node

Server-side JS

Uses the V8 Engine

Includes npm package manager

npm: Node's Package Manager

.NET	NuGet
Ruby	
Java	maven
Python	PIP
PHP	
The Web	
Node	npm

- Package Managers

CommonJS Pattern

```
//1. Get reference to dependency
var dependency = require('/path/to/file');


//2. Declare module
var MyModule = {
  //code here…
}


//3. Expose to others
module.exports = MyModule;
```

- Gulp
- Gulp is a popular JavaScript based TASK RUNNER
- GRUNT is its popular competitor.
- Gulp is popular because it is easy to configure and very fast.
- Gulp has a rich plugin ecosystem that allows to easily automate all sorts of important tasks like minification, linting, bundling,moving files, running tests and watching files.
- Gulp is fast because it is stream based.
- Gulp allows to glue number of tasks together by taking output from one plugin and using it as the input for the next plugin.
- Gulp keeps all tasks in memory rather than writing to disk and this is what makes gulp faster.

Gulp

Task runner

Rich plugin ecosystem

Stream-based

What About ES2015?

ES5

Approved in 2009

The JS you already know

Excellent browser support

We're using ES5 in this course.

ES2015 (ES6)

Approved June 2015

Browser support is growing

Can transpile to ES5 via Babel

- React Enivronment Setup

Our Goal

Just type "gulp", get all this…

- Compile React JSX
- Lint JSX and JS via ESLint
- Bundle JS and CSS files
- Migrate the built app to the dist folder
- Run a dev webserver
- Open the browser at your dev URL
- Reload the browser upon save

= :)

- Environment Set up for React

Want a Shortcut?

Completed Environment Setup: github.com/coryhouse/react-flux-starter-kit

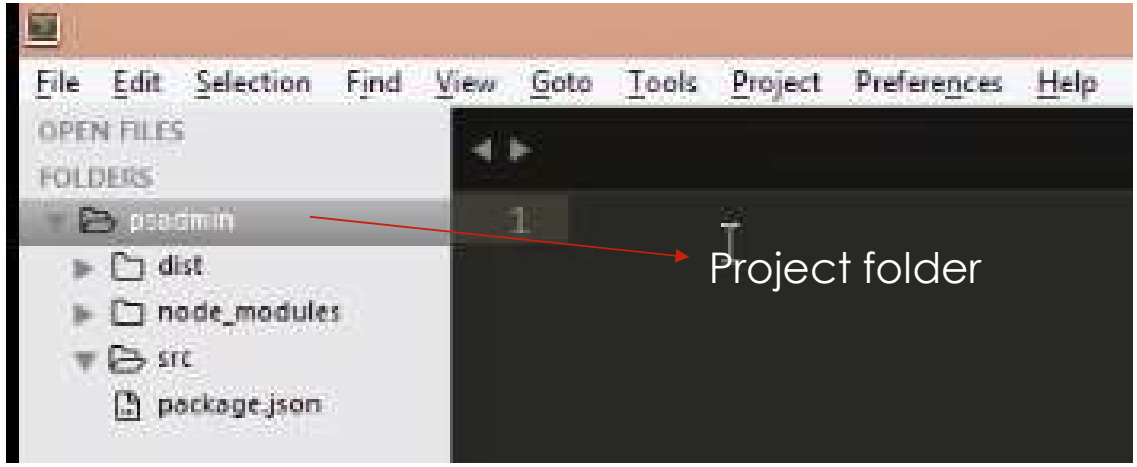
In this module, we'll install and configure:

- Node
- Browserify
- Gulp
- Bootstrap
- ESLint

- Environment Set Up For React
- **Step 1: Node**
  - download Node from Nodejs.org and install node
  - Go to command prompt and check the version of node and npm.
- **Step 2 : package.json**
  - Create project folder and go to the command prompt, to your project folder level.
  - npm init will create a package.json file .
  - This is the file where node keeps track of all the packages downloaded and other pieces of configuration.

```
{
  "name": "psadmin",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Cory House",
  "license": "ISC"
}
```

- Environment Set Up For React
- **Step 3: Installing Gulp**
  - If gulp is being installed for the first time then also run the command "npm install –g gulp".
  - Install gulp in the project folder
    "npm install - -save gulp gulp-connect gulp-open"
  - Choose an editior of your choice and create a folder structure as below



- Environment Set Up For React
- **Step 4: Gulp Configuration**
  - Create a new file at the root level of the project folder name it as gulpfile.js

- *In the gulp file here we have created two tasks.*
- *One to connect to the dev server*
- *Second to open the url in the browser*
- *The config object is mentioning the base url, paths of html files and dist folder.*
- *The open gulp task has a dependency on the connect task which is added in the square brackets.*

gulpfile.js

```
"use strict";

var gulp = require('gulp');
var connect = require('gulp-connect'); //Runs a local dev server
var open = require('gulp-open'); //Open a URL in a web browser

var config = {
    port: 9005,
    devBaseUrl: 'http://localhost',
    paths: {
        html: './src/*.html'
        dist: './dist'
    }
}

//Start a local development server
gulp.task('connect', function() {
    connect.server({
        root: ['dist'],
        port: config.port,
        base: config.devBaseUrl,
        livereload: true
    });
});

gulp.task('open', ['connect'], function() {
    gulp.src('dist/index.html')
        .pipe(open({ uri: config.devBaseUrl + ':' + config.port + '/'}));
});
```

- Environment Set Up For React
- **Step 5: Gulp Configuration**
- Next create an html file **index.html** inside the **src** folder
- Anything we put in **src** folder, should be got into the **dist** folder because we are going to have a build process that takes all the files from **src** folder, bundles it up and places it in the **dist** folder.
- To do this bundling we need another task in the gulpfile.js

```
26  gulp.task('open', ['connect'], function() {
27      gulp.src('dist/index.html')
28          .pipe(open({ uri: config.devBaseUrl + ':' + config.port + '/'}));
29  });
30
31  gulp.task('html', function() {
32      gulp.src(config.paths.html)
33          .pipe(gulp.dest(config.paths.dist))
34          .pipe(connect.reload());
35  });
```

Third task to load '.html' files into dist folder

- Environment Set Up For React
- **Step 6: Gulp Configuration**
- Next we will create a default task, simply to make the development easier
- In this task we just provide an array of tasks that should run by default.
- In the task below just now added to tasks that should run be default
  - Html and open.
- This means that when we type gulp in the terminal it will run the html and open task automatically

```
gulp.task('html', function() {
    gulp.src(config.paths.html)
        .pipe(gulp.dest(config.paths.dist))
        .pipe(connect.reload());
});

gulp.task('default', ['html', 'open']);
```

Default Task

- Environment Set Up For React
- **Step 7: Gulp Configuration**
- Next we create a task to watch files.
- We watch files, so that every time we make a change, gulp knows about it and it reloads the browser.

```
31  gulp.task('html', function() {
32      gulp.src(config.paths.html)
33          .pipe(gulp.dest(config.paths.dist))
34          .pipe(connect.reload());
35  });
36
37  gulp.task('watch', function() {
38      gulp.watch(config.paths.html, ['html']);
39  });
40
41  gulp.task('default', ['html', 'open', 'watch']);
```

Created a watch task

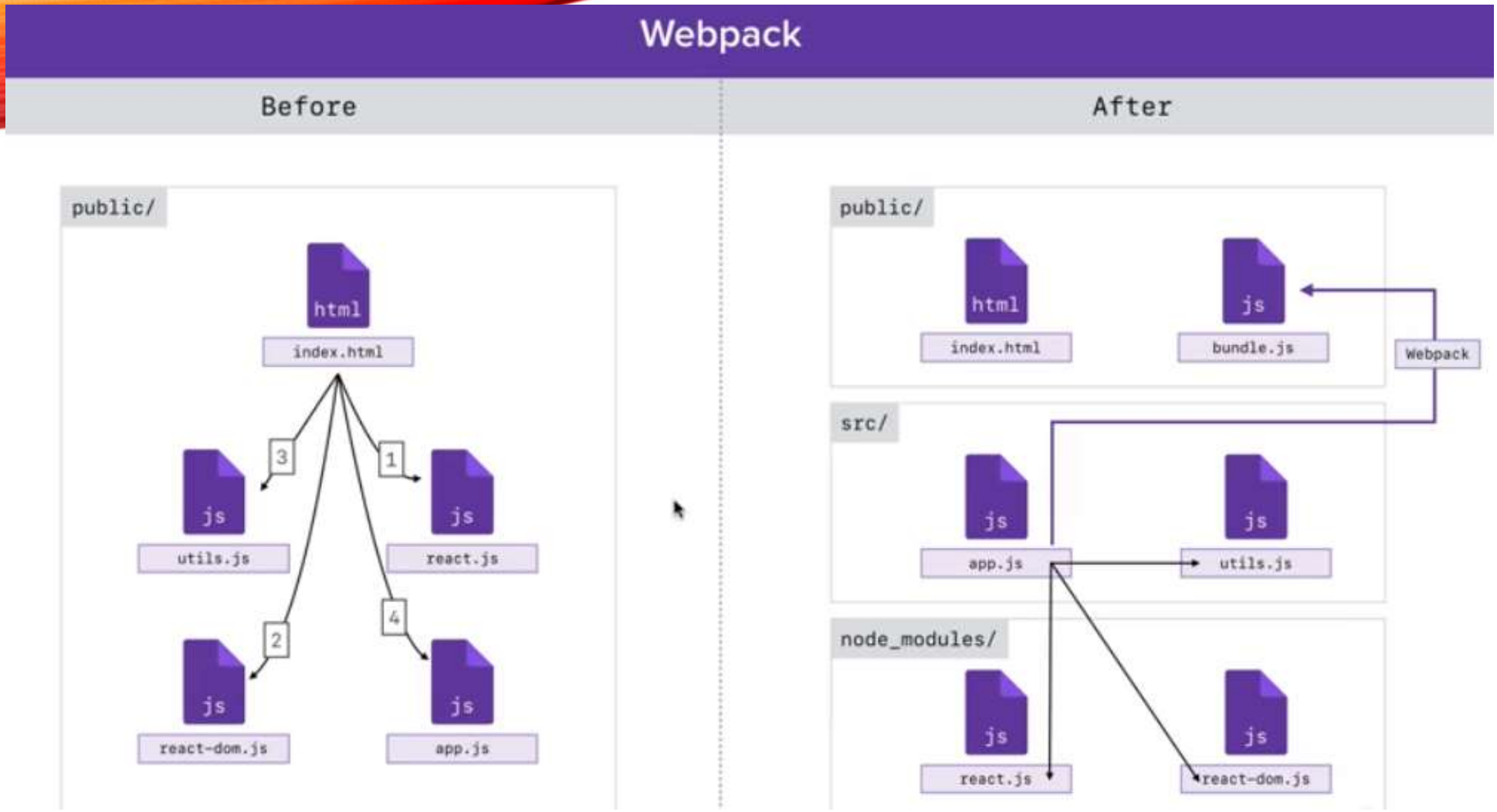
Added watch task to the default task

- Environment Set Up For React
- **Step 8 Browserify Configuration**
- So far we have written configuration to handle html files.
- Now create a file in src folder main.js
- The file, main.js, will bootstrap (connect) the application.
- main.js is used to check if the browserify is working.
- Browserify uses commons pattern, the commons pattern uses module.exports

- To get browserify set up we need some installation.
- "npm install --save browserify reactify vinyl-source-stream"
- Now go to the gulpfile.js and add the require statements

- Environment Set Up For React
- **Step 8 Browserify Configuration (contd…)**

```
"use strict";

var gulp = require('gulp');
var connect = require('gulp-connect'); //Runs a local dev server
var open = require('gulp-open'); //Open a URL in a web browser
var browserify = require('browserify'); // Bundles JS
var reactify = require('reactify');  // Transforms React JSX to JS
var source = require('vinyl-source-stream'); // Use conventional text streams with Gulp
```


Webpack
Before
After
public/
html
index.html
3
1
js
utils.js
js
react.js
2
4
js
react-dom.js
js
app.js
public/
html
index.html
js
bundle.js
Webpack
src/
js
app.js
js
utils.js
node_modules/
js
react.js
js
react-dom.js

- Why React – History

2011	Created by Facebook
2012	Used on Instagram
2013	Open sourced
2014	Embraced by many large companies
2015	React Native released
2016	React 15 released (previous version was 0.14)
Today	Over 30k components at Facebook Full-time dev staff Used by many in Fortune 500

- Why React?

Why React?

Flexibility

Developer experience

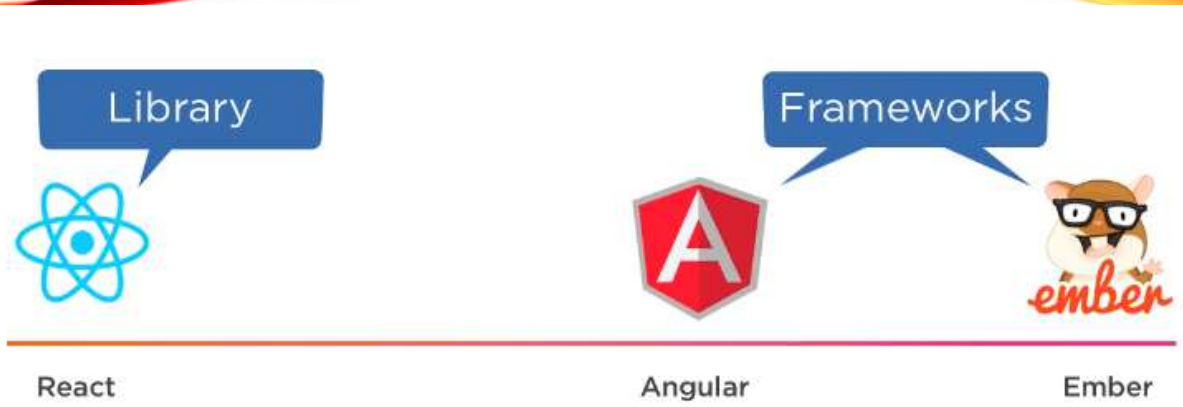
Corporate investment

Community support

Performance

Testability

- Why React? Reason 1 : Flexibility



Where Can I Use React?

Opinion →

Web apps

Static sites

Mobile

Desktop

Server-rendered

Virtual Reality

React Renderers

react-dom

react-native

react-vr

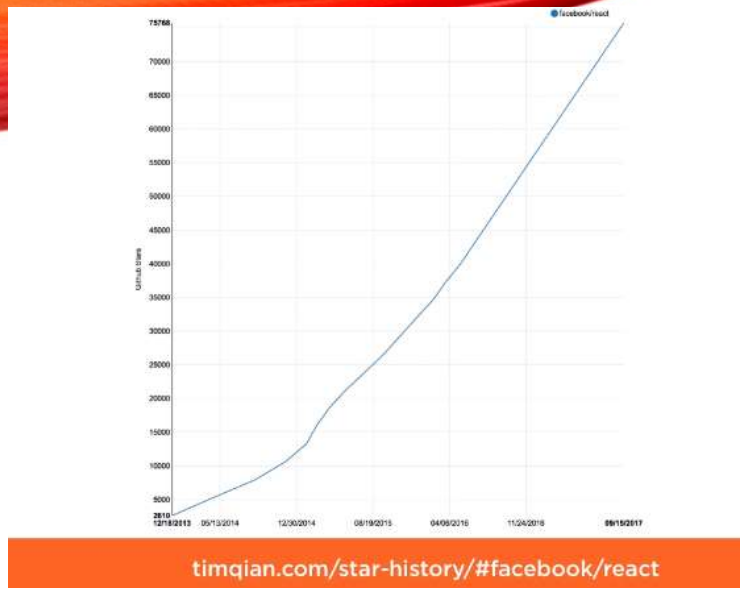
- Why React? Reason2 :Developer Experience

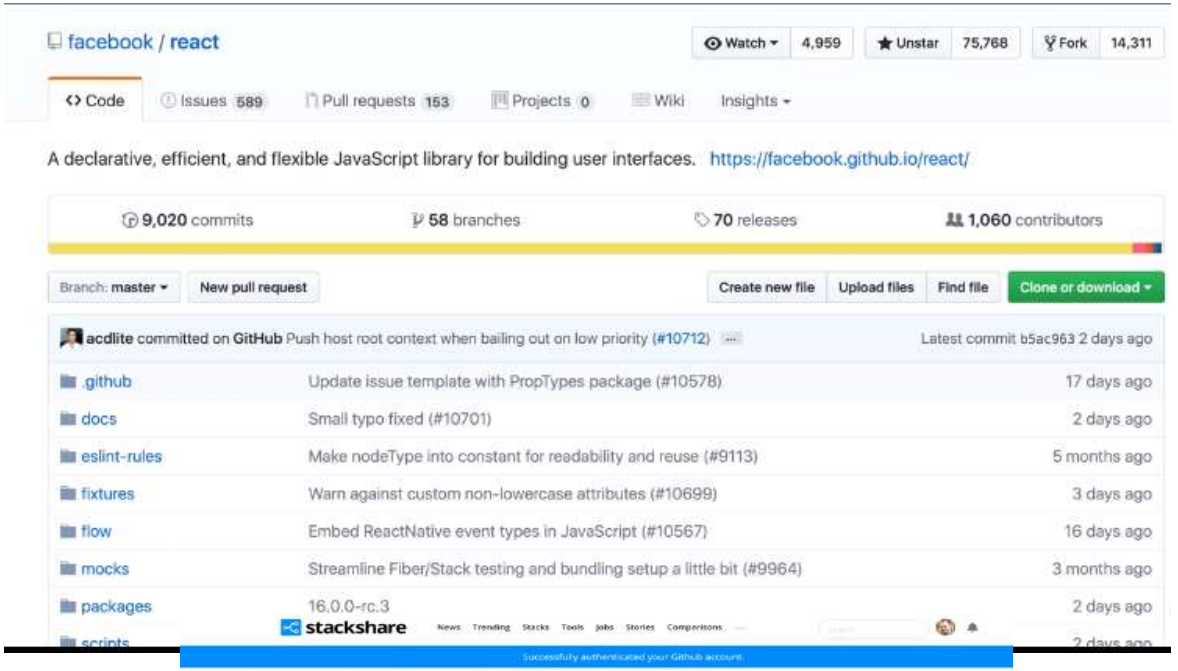
```
import React from 'react';

function HelloWorld(props) {
  return <div>Hello {props.name}</div>
}
```

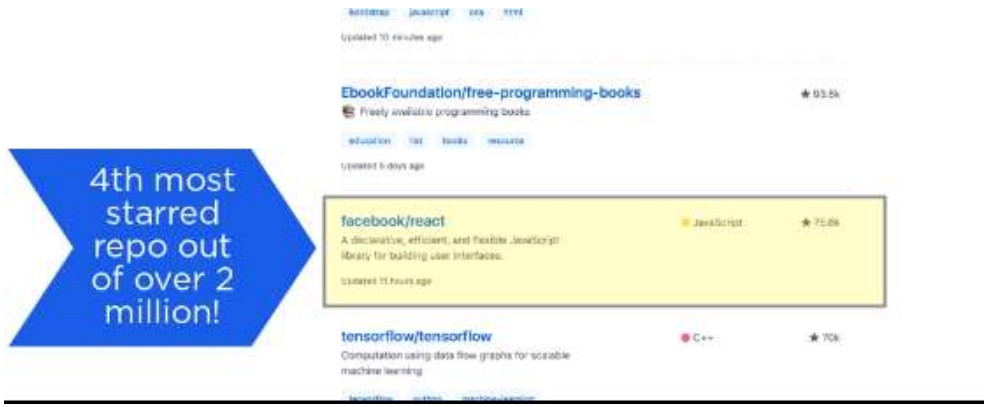
```
import React from 'react';

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

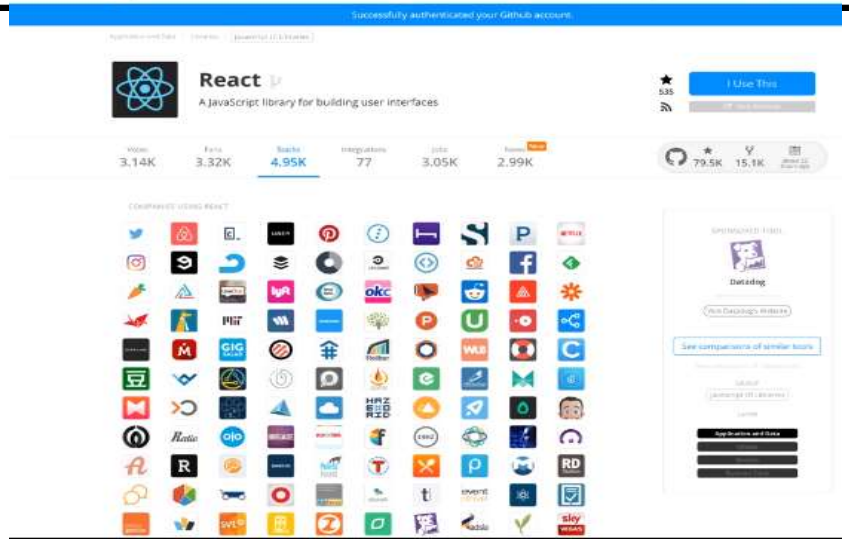
- Why React? Developer Experience and Corporate Investment

- Why React? Community









- Why React? Community

An answer to your use case likely exists online.

Companies Using React

- Why React? Community

Ecosystem

React Router

Redux

Mobx

Jest

GraphQL

- Why React? Performance

Why Virtual DOM?

Updating the DOM is expensive

So React minimizes DOM changes

Without Virtual DOM

Blindly update DOM using new state.

With Virtual DOM

Update the DOM in the most efficient way.

Avoids layout thrash
Saves battery and CPU
Enables simple programming model

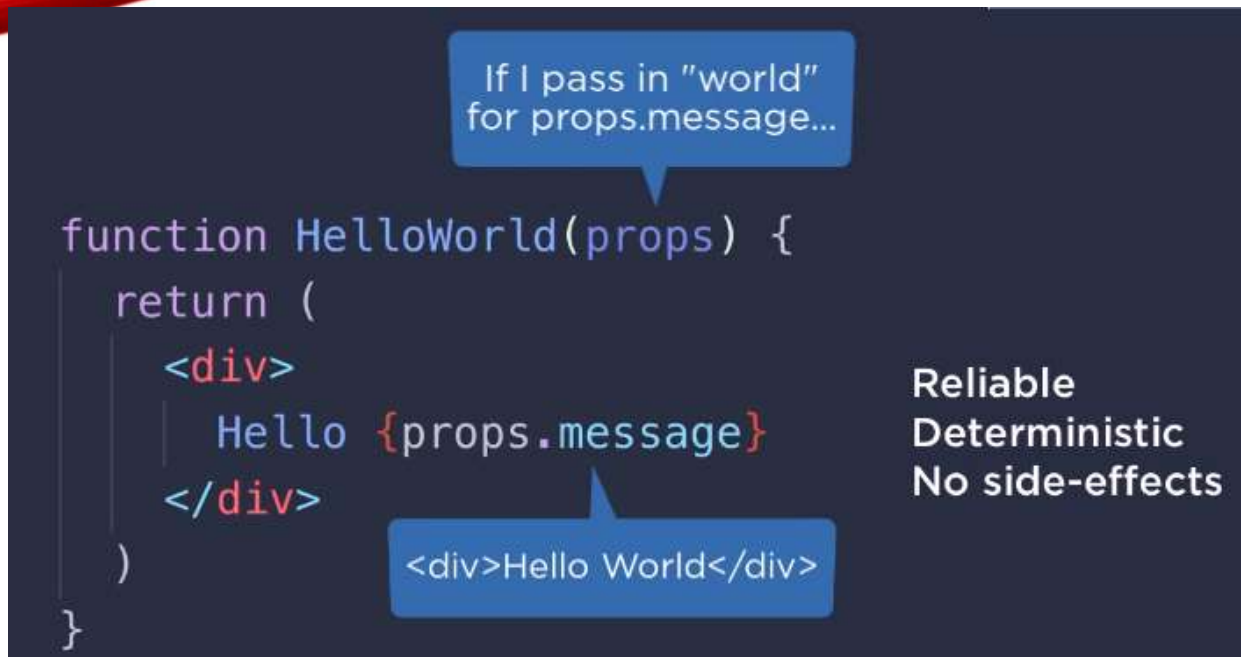
- Why React? Testing

Traditional UI tests	React
Hassle to set up	Little to no config required
Requires browser	Run in-memory via Node
Slow	Fast
Brittle integration tests	Reliable, deterministic unit tests
Time-consuming to write, maintain	Write quickly, update easily

```
function HelloWorld(props) {
  return (
    <div>
      Hello {props.message}
    </div>
  )
}
```

Pure Function: Always returns same output for a given input.

- Why React? Testing



- Why React? Testing

Testing Frameworks

Mocha

Jasmine

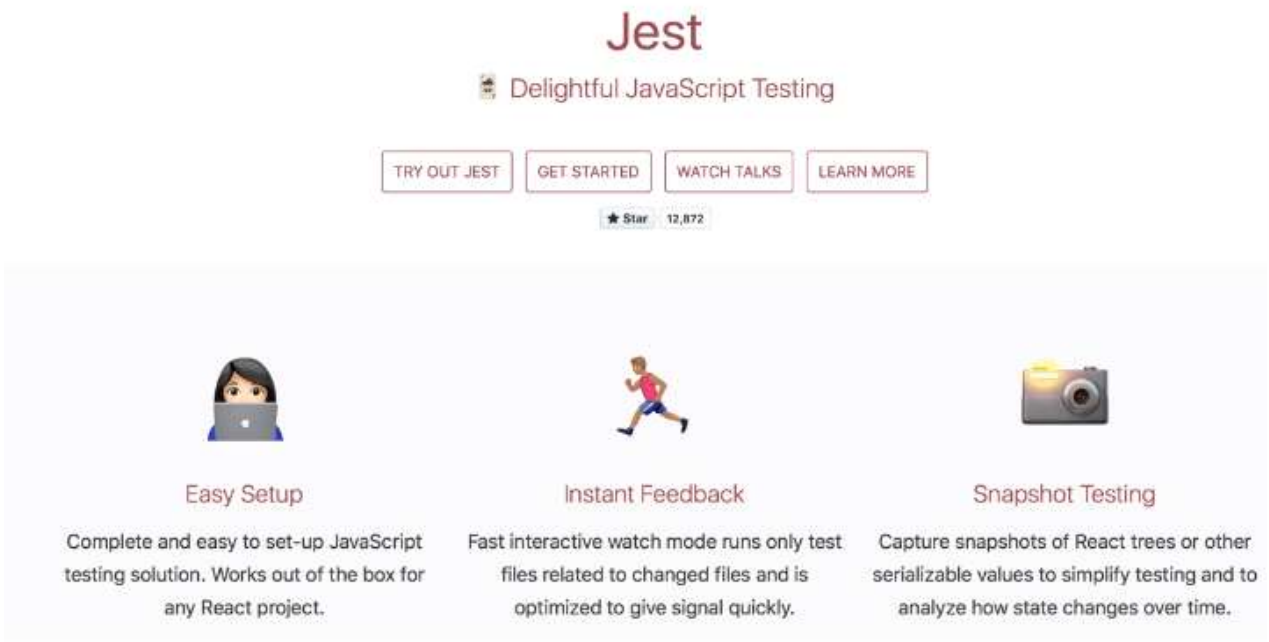
Tape

QUnit

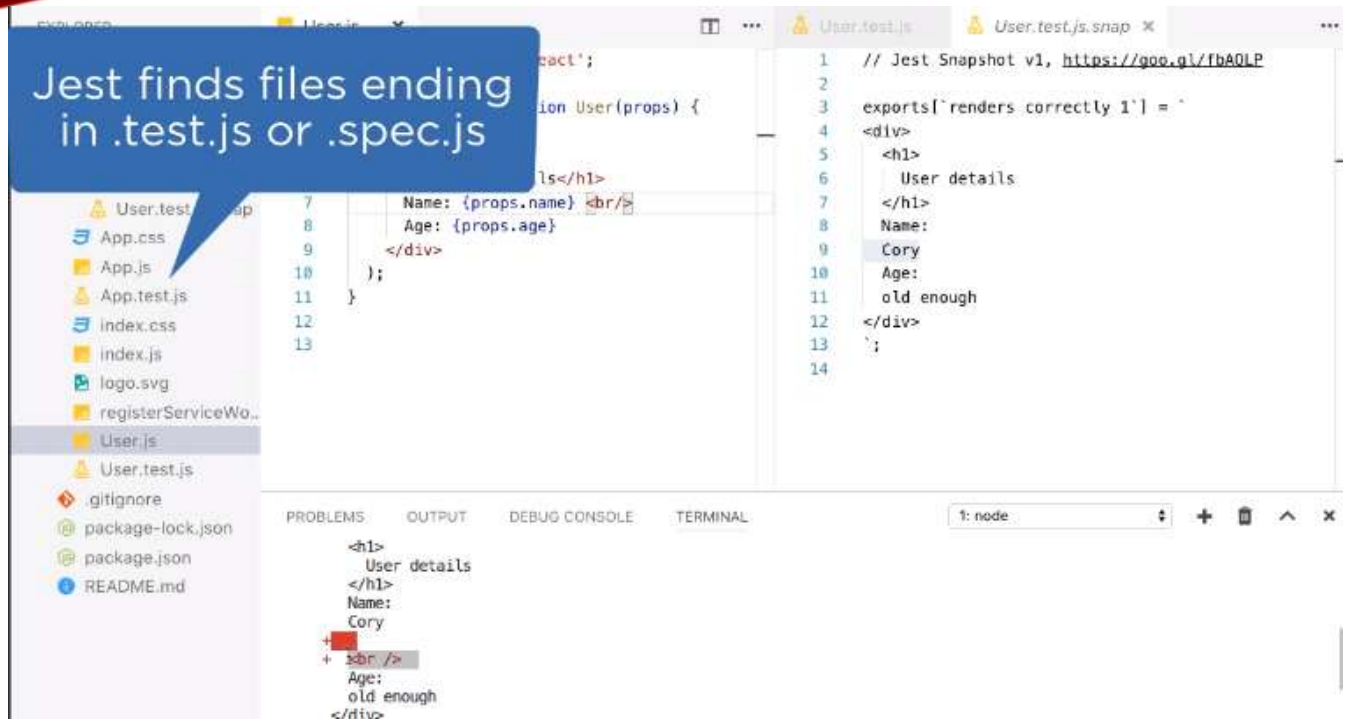
AVA

Jest

- Why React? Testing



- Why React? Testing



- Why Not React?

Why not React?

- JSX differs from HTML
- Build step
- Potential version conflicts
- Old features in searches
- Decision fatigue
- ~~License~~

React now uses a standard MIT open source license. 👍

- Why Not React? Html and JSX differences

HTML	JSX
for	htmlFor
class	className
<style color="blue">	<style={{color: 'blue'}}>
<!-- Comment -->	{*/ Comment /*}

- Why Not React? Build step required

JSX Must Be Compiled to JS

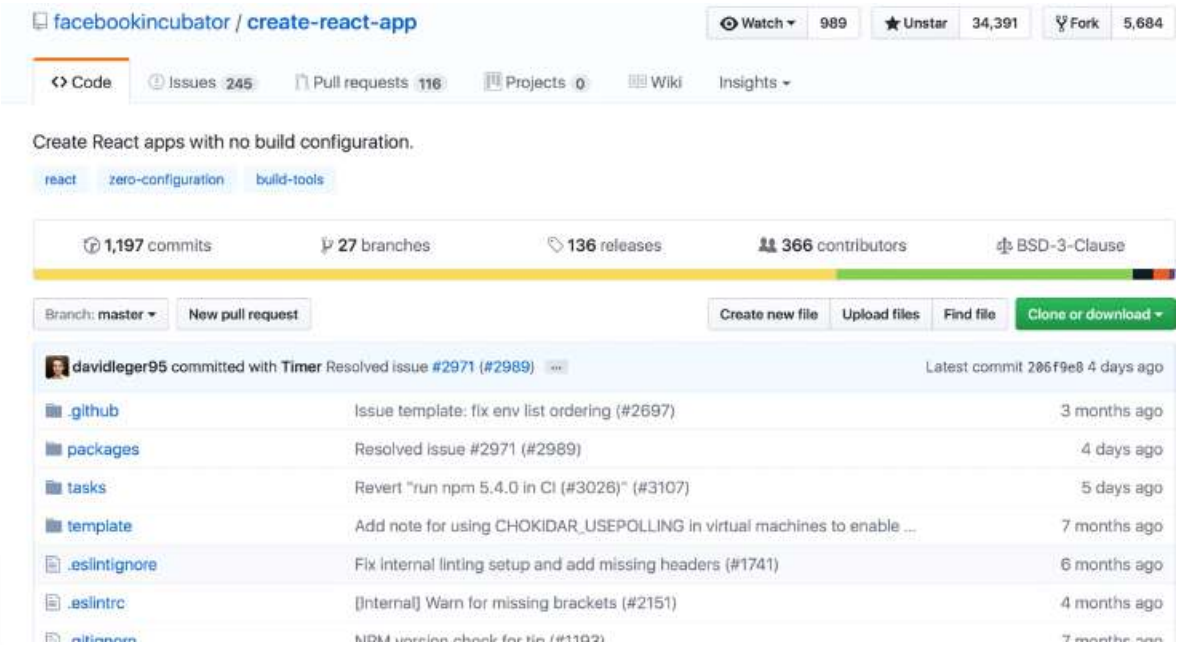
```
<h1 color="red">Heading here</h1>
```

```
React.createElement("h1", {color: "red"}, "Heading here")
```

Popular Transpilers Compile JSX

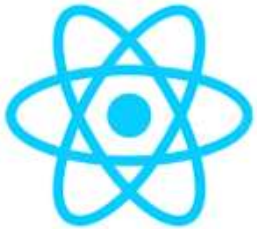
Babel

TypeScript



- Why Not React? Version Confilct

Version Compatibility

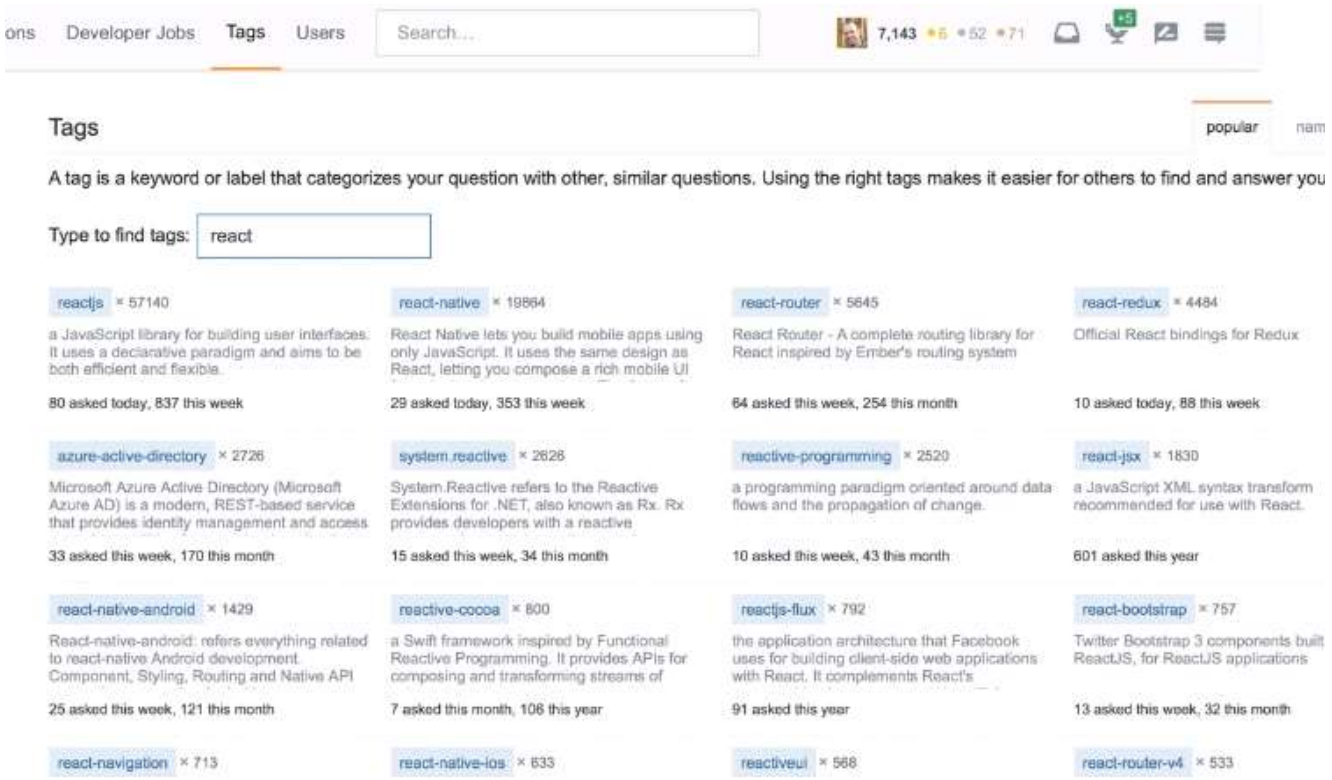


React

React Router

Latest requires React 15+

- Why Not React? Outdated resources



- Why Not React? Outdated resources

Features Extracted from React Core

Old

```
import {render} from 'react';

React.createClass

import {PropTypes} from 'react';

mixins: [mixinNameHere]
```

New

```
import {render} from 'react-dom';

var crc = require('create-react-class');

import PropTypes from 'prop-types';
```

Higher order components, render props

- Why Not React? Decesion Fatigue

Dev environment

ES class or createClass

Types

State

Styling

- Why Not React?

- Why Not React?

- Why Not React?

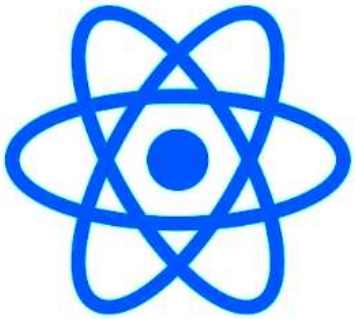
- Why Not React?

- Why Not React?

INTRODUCTION

- If the device we're trying to interface can understand JavaScript, we can use React to describe a user interface for it.
- Since web browsers understand JavaScript, we can use React to describe web user interfaces.
- It is appropriate to use the word describe because that's what we basically do with React, we just tell it what we want and React, with its ReactDOM library for browsers, will build the actual user interfaces on our behalf in the web browser
- We describe user interfaces with React and tell it what we want, not how to do it.
- React will take care of the how and translate our declarative descriptions, which we write in the React language to actual user interfaces in the browser.
- React shares this simple declarative power with HTML itself, but with React we get to be declarative for HTML interfaces that represent dynamic data, not just static data.

- **React's Main Design Concepts**



Components
- Like functions
- Reusable and composable
- Can manage a private state

Reactive updates
- React will react
- Take updates to the browser

Virtual views in memory
- Write HTML in JavaScript
- Tree reconciliation

- React has three main design concepts that drive its popularity.
- The first one is its **components**.
- In React, we describe user interfaces using components.
- We can think of components as simple functions in any programming language.
- We call functions with some input that gives some output.
- We can reuse functions as needed and compose bigger functions from smaller ones.
- Components are exactly the same; we call their input properties and state.
- The component output is a description of a user interface, which is similar to HTML for browsers.
- We can reuse a single component in multiple user interfaces, and components can contain other components.
- A full React component can have a private state that hold data that may change over time.

- **React's Main Design Concepts**
- The second concept is React's **nature of reactive updates**.
- React's name is the simple explanation for this concept.
- When the state of a component, the input, changes, the user interface it represents, the output, changes as well.
- Remember, it's just like a function.
- This change in the description of the user interface needs to be reflected in the device we're working with.
- In the browser, we need to regenerate the HTML views in the Document Object Model.
- With React, we do not need to worry about how to reflect these changes, or even manage when to take changes to the browser.
- React will simply react to the state changes and automatically update the DOM when needed.

- **React's Main Design Concepts**
- The third concept is the **virtual representation of views in-memory**.
- With React, we write HTML using JavaScript.
- We rely on the power of JavaScript to generate HTML that depends on some data rather than enhancing HTML to make it work with that data.
- Enhancing HTML is what other JavaScript frameworks usually do, for example, Angular extends HTML with features like loops, conditionals, and others.
- When we receive just the data from the server, in the background with AJAX, we need something more than HTML to work with that data.
- It's either using an enhanced HTML or using the power of JavaScript itself to generate the HTML.
- Both approaches have advantages and disadvantages, and React embraces the latter one, with the argument that the advantages are stronger than the disadvantages.
- In fact, there is one major advantage is, using JavaScript to render HTML allows React to have a virtual representation of HTML in-memory, which is known as the virtual DOM.

- **React's Main Design Concepts**
- React uses this concept to render an HTML tree virtually first, and then every time a state changes and we have a new HTML tree that needs to be written back to the browser's DOM.
- Instead of writing the whole tree, React will only write the difference between the new tree and the previous tree, since React has both trees in-memory.
- This process is known as tree reconciliation, and it's the best thing that has happened in web development since AJAX.

- What is React ?

- React is an open source client side web library for building composable user interfaces developed by Facebook and Instagram.
- The Facebook's like and comment components are built using react.
- One of the benefits and goals of the react project is to make developing a large scale single page application easier .

The URL of react.js home page has documentation, URLI's and useful links.

Another exciting feature of react is the virtual DOM which efficiently re-renders the DOM whenever change happens

Collaboration between

- Facebook
- Instagram

- http://facebook.github.io/react/

- What is React?

React is a JavaScript library in which components are defined and eventually become HTML.

"React abstracts away the DOM from you, giving a simpler programming model and better performance."

WHAT IS REACT?

- React is a library for creating user interfaces and that means that it's not necessarily concerned with the whole process of creating an application.
- So it's not so much about the data or the controllers, it's just about the view.
- It's simply focused on the view and how an interface connects to those other elements.
- What makes React really different is that it completely takes care of managing the DOM for you.
- React uses virtual DOM

- What is React ?

- React is an open source client side web library for building compo sable user interfaces developed by Facebook and Instagram.
- The Facebook's like and comment components are built using react.
- One of the benefits and goals of the react project is to make developing a large scale single page application easier .

The URL of react.js home page has documentation, URLI's and useful links.

Another exciting feature of react is the virtual DOM which efficiently re-renders the DOM whenever change happens

Collaboration between

- Facebook
- Instagram

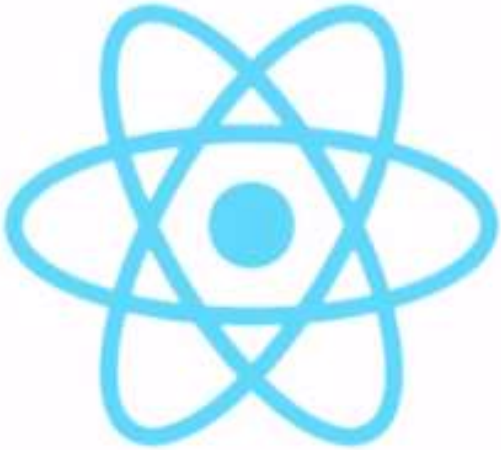
- http://facebook.github.io/react/

WHAT IS REACT?

- It's a representation of the DOM in JavaScript that is controlled by what your component is doing.
- React efficiently updates the real DOM so you never have to worry about it.
- **Differences between React and other tools.**
  - Now first, unlike other frameworks, everything in React is done by working with components.
  - That's instead of models, views, and controllers.
  - An application in REACT is a main component with a bunch of sub-components.
  - Another thing that makes react unique is the one-way data flow structure.
  - In Angular JS 1.x and other frameworks, you create a two-way relationship between the model and the view.
  - React can do the same things but keep the flow of data going one way.
  - The reason that's good is because you end up managing everything through the main component state.
  - The main component stores information about what's going on with your application and sub-components and what this framework does is quote, unquote react to changes in that information.

- **Why should you learn React ?**
- React has a very small learning curve
- React is super fast and agile. React renders DOM much quicker than any other alternative frameworks
- React uses JSX, a language that specifies the DOM elements for your components right inside the JS file. This means the logic behind the components and the visuals are all in one place.
- React has a great community, companies like AIRBNB, New York Times, Netflix, Facebook are using react for production. These companies are also contributing to the react core and creating associated 3rd party libraries that go great with any React application

Why React?



Renders views ultra-quickly

Easy to test

Enforces good coding practices

Supported and used by Facebook

WHAT IS REACT?

- **Differences between React and other tools.**
- If something changes in the State or in the data, then React with re-render the DOM for automatically.
- The main component passes information to sub-components via something called Props, or properties.
- Props, or properties, can both pass information down to sub-modules but also trigger actions in the main module.
- Finally, although you could write React with regular JavaScript, most professional React development uses a special language called JSX, which is a combination of JavaScript and XML.
- JSX allows to combine the best parts of JavaScript with HTML tags.
- JSX is little bit like having a templating language.
- JSX makes the language a lot easier to use.

REACT FEATURES

- React aims at JS's functional programming feature over Object oriented
- It follows the principle of immutable data, in which copies of objects are created and replaced instead of mutating the originals.
- React is commonly used in enterprise applications by companies including Paypal, AirBnB, Apple, Uber, and of course facebook.
- React is easy and will make your work easier in developing UI's.

EFFICIENT RENDERING WITH REACT

- The DOM or Document Object Model is the structure of HTML elements that make up a web page.
- The DOM API refers to how these page elements are accessed and changed.
- React made updating the DOM faster by using DOM Diffing.
- DOM Diffing happens when you compare the currently rendered content with the new UI changes that are about to be made.
- React optimizes this by making only the minimal changes necessary.
- With DOM Diffing, JavaScript objects are compared and this is faster than writing to or reading from the actual DOM.

EFFICIENT RENDERING WITH REACT

- Whenever functions like Get Element by ID, are used in JS DOM elements are read from the DOM.
- When you change any of those elements, change classes, change content, you're writing to the DOM.
- Writing to and reading from the DOM is slow but working with JavaScript objects is faster.

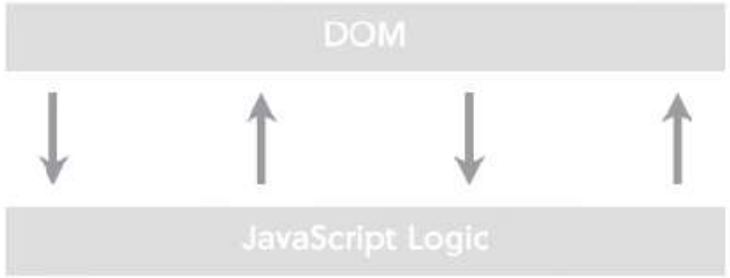
EFFICIENT RENDERING WITH REACT

- A slower DOM is rendered with other frameworks and libraries like Backbone.
- The way we typically program with Backbone is that we re-render the same data over and over again, even when the changes haven't happen everywhere.
- This re-rendering slows down apps a lot.
- With React, only the minimal changes will be rendered so this is really efficient.
- Let's say, it is needed to update a single item in a bulleted list, REACT will just make the one required change.
- We never read from the DOM and we only write to it when a change is required.

WHY IS REACT SO FAST?

- One reason why REACT is popular is because of its rapid DOM rendering speed.
- Reading and writing to the DOM is slow and every time we read or write to the DOM becomes expensive, this effects the speed of the application.
- Reading and writing to JS objects is faster because it has a virtual DOM that writes to the browser only when it needs to.
- React never reads from the real DOM and DOM updates are only made if there has been a change.
- React has a very small learning curve, agile, fast
- The JSX that allows you to specify the DOM elements right inside the JS file
- With JSX the logic and visuals are all in one place.
- React has a great community

WORKING OF JS AND OTHER LIBS

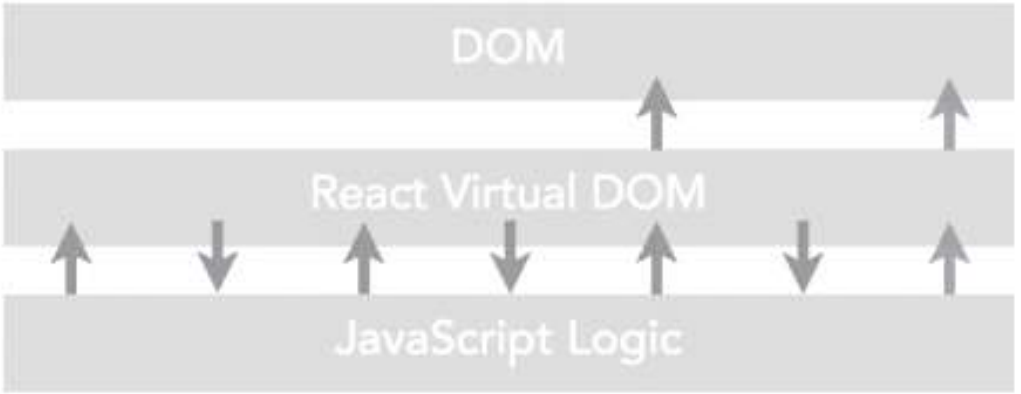


Here JS is interacting with DOM to make updates
Whenever the functions like getElementById is used the DOM is being read.
When one of the element of DOM is changed you are writing to the DOM and all these connections are going back and forth.

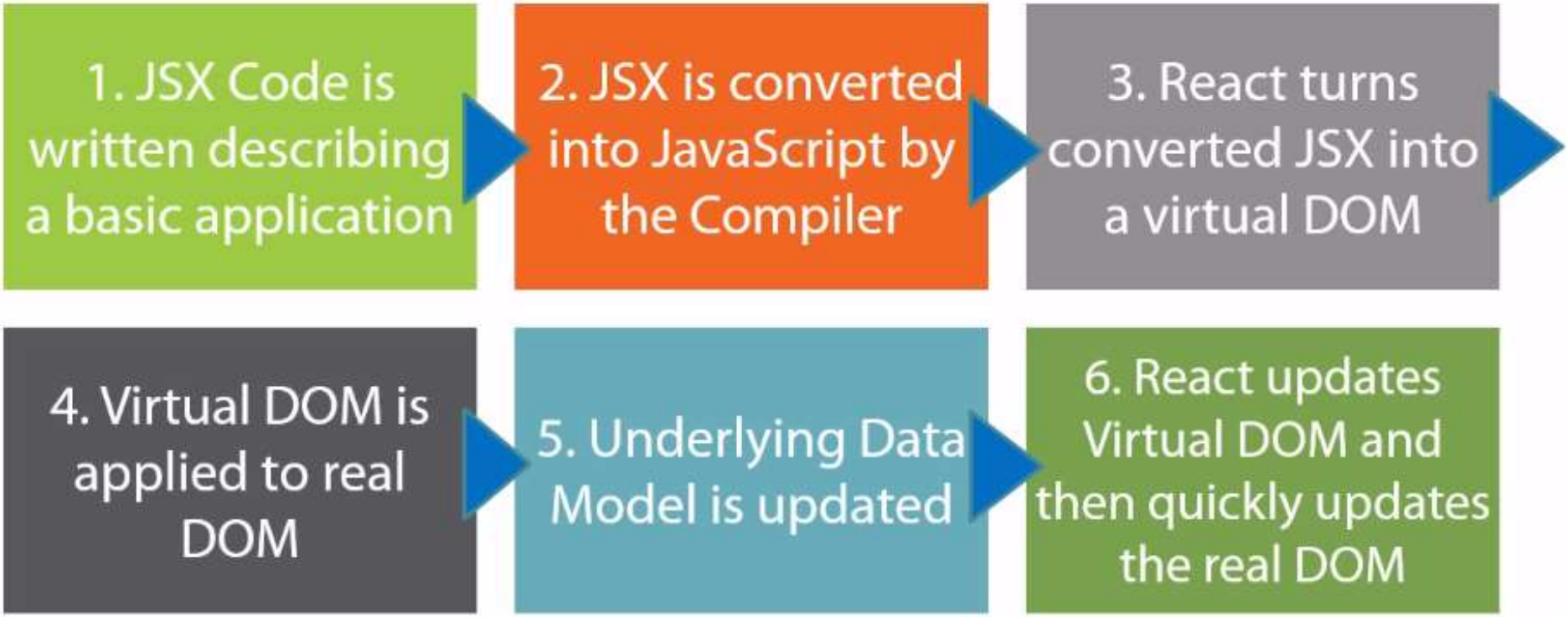
Frameworks like backbone also deal with the same concepts where the same data is re-rendered over and over again.
Even if changes do not occur we still continue to render.
This re-rendering slows the app down

WHY IS REACT SO FAST?

- React only interacts with the virtual DOM, that JS object in between the JS logic and the actual DOM.
- So when the render function is called React will update the virtual DOM and that will pull only the necessary changes all the way to the real DOM



How React Renders the View



SETTING UP REACT TOOLS WITH CHROME

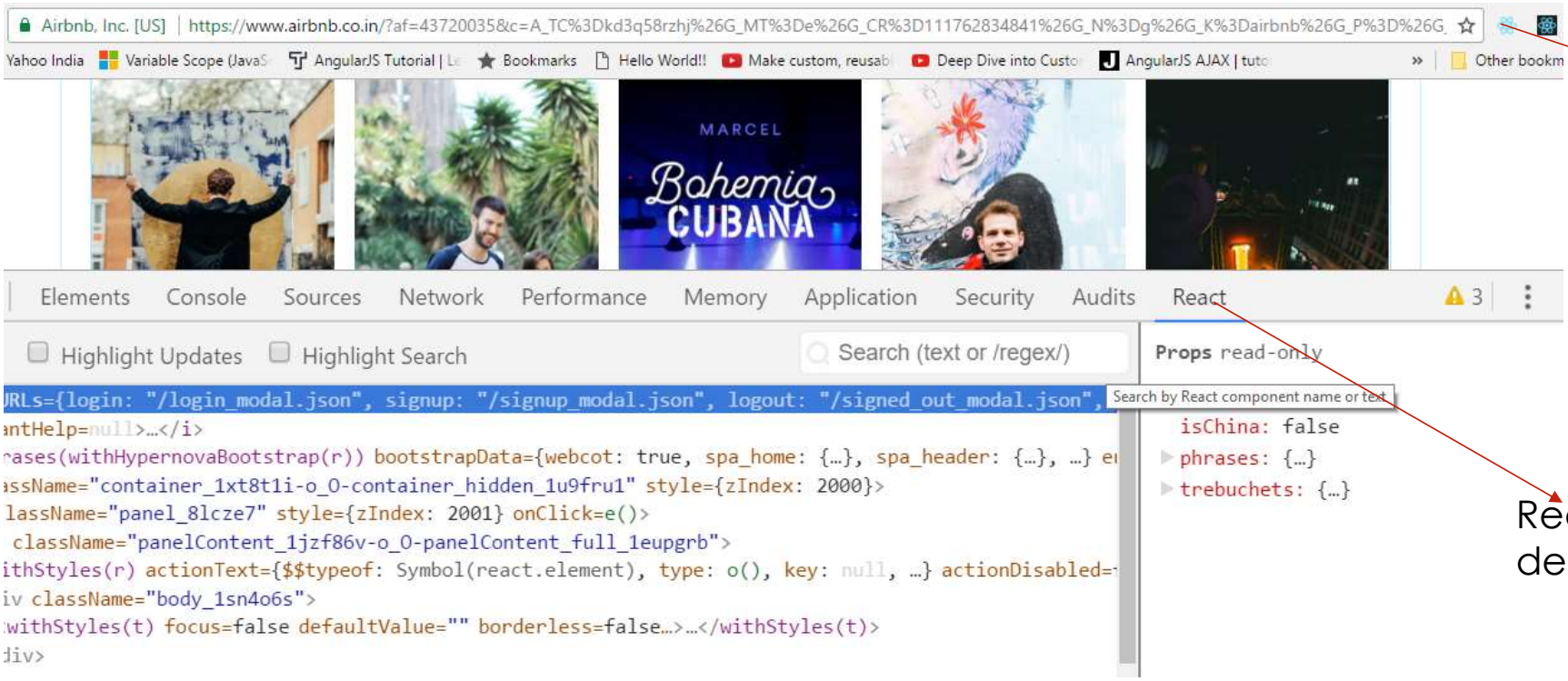
There are some great developer tools available to work with react that can be added as chrome extensions.

https://chrome.google.com/webstore/category/apps

a. First install react detector: It detects the websites where react is used

b. React developer tools:

REACT EXT. ADDED IN CHROME



React used in website

React added in developer tools

- Advantages of React

- The advantages that react offers over the competing JS user interface libraries.
- SPEED is a major design goal of react.
- The react team claims that react is capable of producing user interfaces that render 60 frames per second on non-git iphones.
- That means to achieve this speed the react render process has to complete in less than 16ms.
- The reason react is very fast is because it uses novel rendering algorithm that minimizes the amount of work that has to be done.
- Unlike backbone's imperative style, react chooses a declarative style.
- A react application is a set of components each of which declaratively defines a mapping between some state and the desired UI.

- Advantages

- The interface in react changes only when the state changes.
- The declarative style has the advantage of being easier and prevents bugs.
- In react its easy to see how code changes or the events will affect the programs outcome.
- Reacts components are self contained units of functionality.
- The react components publish a simple interface that defines their input as properties and their outputs as callbacks.
- By implementing the interface, the react components can be freely nested within each other.

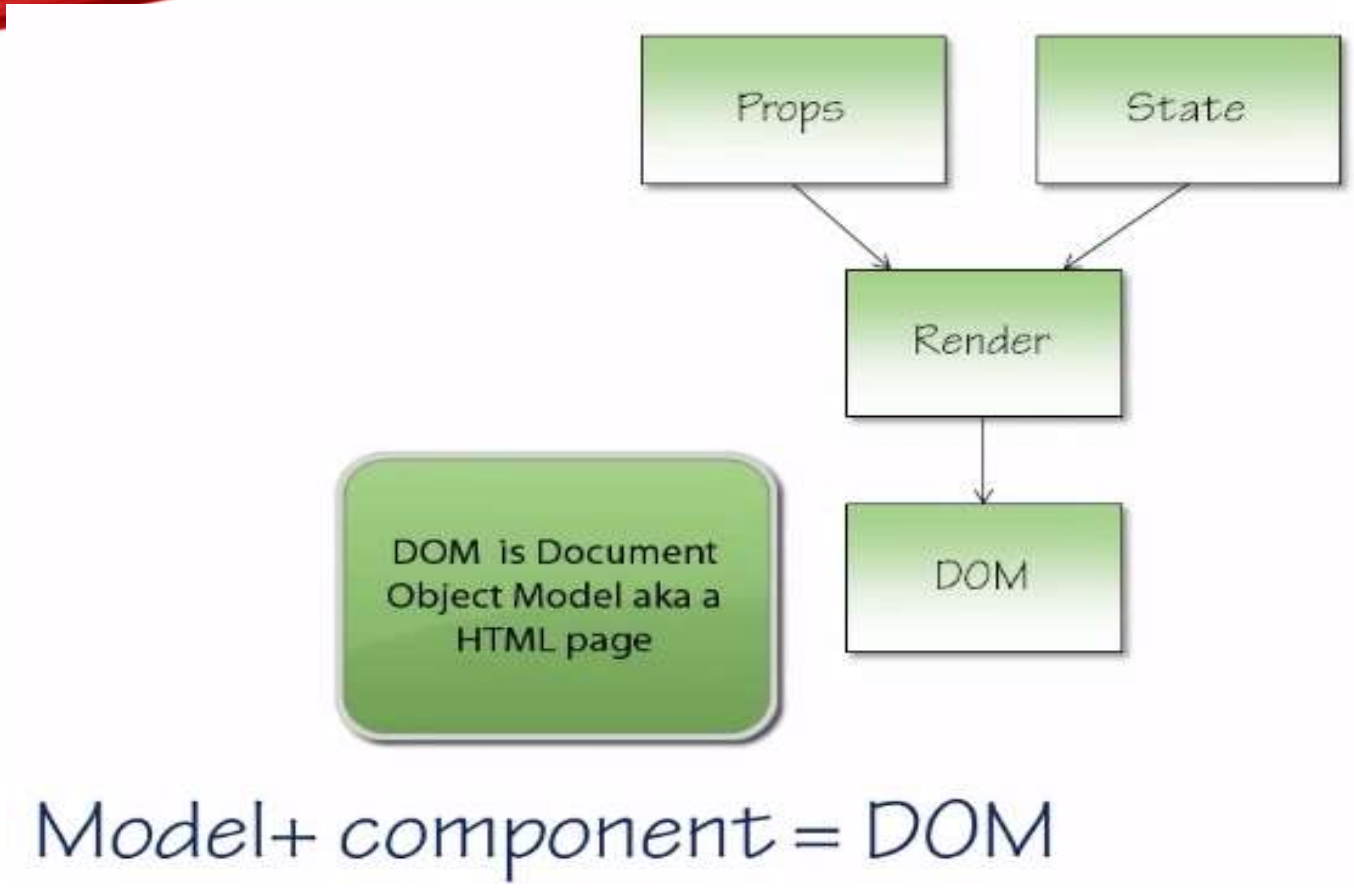
- Advantages

- **Speed!**
  - 60 fps
  - Render < 16ms
- **Declarative**
  - Easier to reason about
  - Easier to prevent bugs
- **Composable**

- Prerequisites

- Basic of JavaScript
  - Varibles (const/let)
  - Classes and functions
  - Loops and conditionals
- HTML 5
- CSS 3
- npm

- **React Architecture**



- **React Functional Component**
- **Functional components are stateless components**
- **Performance of functional components is better than class components**

Rendering a Component

```
import ReactDOM from 'react-dom';
import React from 'react';

function Hello(props) {
  return <h1>Hello at {props.now}</h1>;
}

ReactDOM.render(<Hello now={new Date().toISOString()} />,
  document.getElementById('root')
);
```

- Class Component And Functional Component

Functional Component Benefits

- Easier to understand
- Avoid `this` keyword
- Less transpiled code
- High signal-to-noise ratio
- Enhanced code completion / intellisense
- Easy to test
- Performance
- Classes may be removed in future

- Class vs Functional Components

When Should I Use Each? Pre 16.8...

Class Component	Function Components
State	Everywhere else ☺
Refs	
Lifecycle methods	

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page describes the APIs for the built-in Hooks in React.

If you're new to Hooks, you might want to check out the overview first. You may also find useful information in the frequently asked questions section.

- Basic Hooks
  - `useState`
  - `useEffect`
  - `useContext`
- Additional Hooks
  - `useReducer`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`
  - `useDebugValue`

With react hooks introduced in version 16.8 functional components have been improved

When Should I Use Each? After 16.8...

Class Component	Function Components
componentDidError	Everywhere else ☺
getSnapshotBeforeUpdate	

- Container Components vs Presentation Components

The "backend" for the frontend

Container	Presentation
Little to no markup	Nearly all markup
Pass data and actions down	Receive data and actions via props
Knows about Redux	Doesn't know about Redux
Often stateful	Often no state

Alternative Jargon

Container	Presentational
Smart	Dumb
Stateful	Stateless
Controller View	View

Container Components vs Presentation Components

"When you notice that some components don't use props they receive but merely forward them down...it's a good time to introduce some container components."

Dan Abramov

- Props

```
ReactDOM.render(<div id="mydiv"></div>,
  document.getElementById('root')
);
```

Props

```
function Sum(props) {
  return (
    <h1>{props.a} + {props.b} = {props.a + props.b}</h1>
  );
}

ReactDOM.render(<Sum a={4} b={2} />,
  document.getElementById('root')
);
```

```
props = {
  a: 4,
  b: 2
}
```

- Props

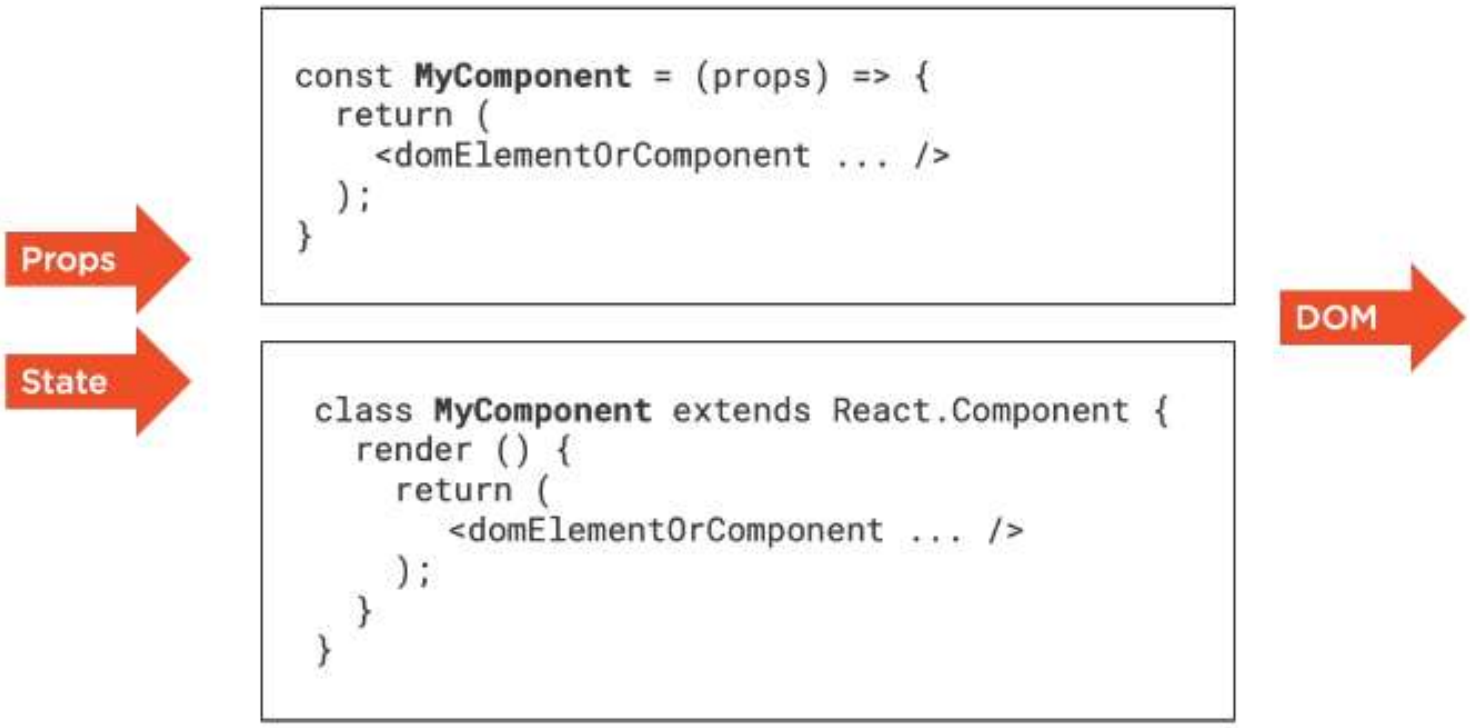
"All React components must act like pure functions with respect to their props."

React documentation

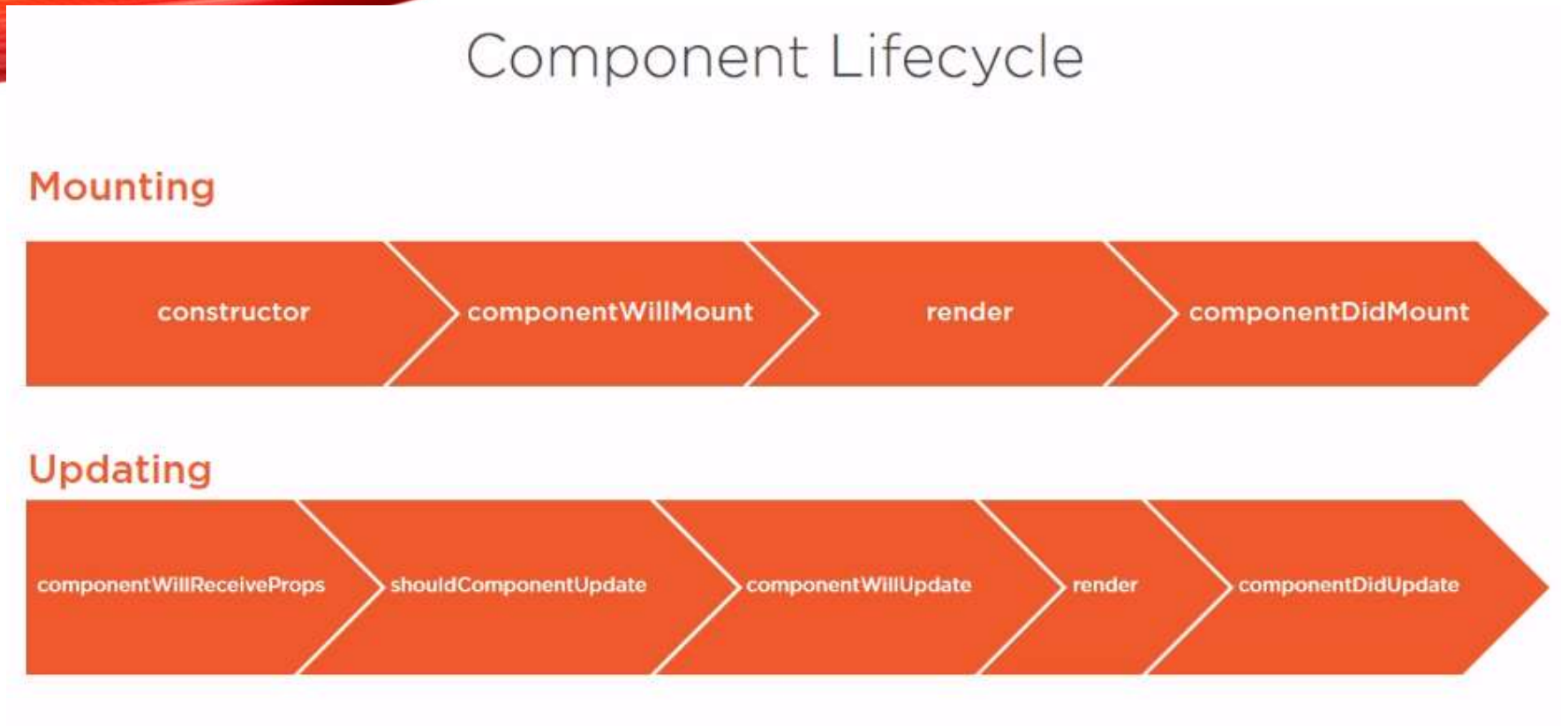
- React Class Component
- Class components are also called as stateful components

Class Components

```
class Sum extends React.Component {
  render() {
    return <h1>
      {props.a} + {props.b} = {props.a + props.b}
     </h1>;
  }
}

ReactDOM.render(<Sum a={4} b={2} />,
  document.getElementById('root')
);
```


Props
State
const MyComponent = (props) => {
return (
<domElementOrComponent ... />
);
}
class MyComponent extends React.Component {
render () {
return (
<domElementOrComponent ... />
);
}
}
DOM

- Component Life Cycle Methods



- Component Life Cycle methods

componentWillReceiveProps

When

When receiving new props. Not called on initial render.

Why

Set state before a render.

shouldComponentUpdate

When

Before render when new props or state are being received.
Not called on initial render.

Why

Performance. Return false to void unnecessary re-renders.

- Component Life Cycle methods

componentWillUpdate

When

Immediately before rendering when new props or state are being received.
Not called on initial render.

Why

Prepare for an update

componentDidUpdate

When

After component's updates are flushed to the DOM.
Not called for the initial render.

Why

Work with the DOM after an update

- Component Life Cycle methods

componentWillUnmount

When

Immediately before component is removed from the DOM

Why

Cleanup

- React Routing

- React Routing

Route Configuration

Route – Declaratively map a route

DefaultRoute –For URL of "/". Like "index.html".

NotFoundRoute – Client-side 404

Redirect – Redirect to another route

- React Routing

Params and Querystrings

```
// Given a route like this:
<route path="/course/:courseId" handler={Course} />


// and a URL like this:
'/course/clean-code?module=3'


//The component's props will be populated
var Course = React.createClass({
   render: function() {
      this.props.params.courseId; // "clean-code"
      this.props.query.module; // "3"
      this.props.path; // "/course/clean-code/?module=3"
      // ...
   }
});
```

- React Routing

Links

URL: /user/1

Route:
```
<route name="user" path="/user/:userId" />
```

JSX:
```
<Link to="user" params={{userId: 1}}>Bobby Tables</Link>
```

```
<a href="/user/1">Bobby Tables</a>
```

```
var Header = React.createClass({
  render: function() {
    return (
        <nav className="navbar navbar-default">
          <div className="container-fluid">
            <a href="/" className="navbar-brand">
              <img src="images/pluralsight-logo.png" />
            </a>
            <ul className="nav navbar-nav">
              <li><Link to="app">Home</Link></li>
              <li><Link to="authors">Authors</Link></li>
              <li><Link to="about">About</Link></li>
            </ul>
          </div>
        </nav>
```

- React Routing

Redirects

Need to change a URL? Use a Redirect.

1. Alias Redirect

```
var Redirect = Router.Redirect;
```

2. Create a new route

```
<Redirect from="old-path" to="name-of-new-path" />
```

- Component Life Cycle methods

Lifecycle Methods

componentWillMount
componentDidMount
componentWillReceiveProps
shouldComponentUpdate
componentWillUpdate
componentDidUpdate
componentWillUnmount

componentWillMount

When
Before initial render, both client and server

Why
Good spot to set initial state

componentDidMount

When
After render

Why
Access DOM, integrate with frameworks, set timers, AJAX requests

- State
- Another way to hold the data is with states
- Props are values passed in by the components parent
- State is local mutable data that can be created and modified within the component.
- States increases complexity and limits the composability of the component.
- If possible usage of states should be avoided.
- To use state we always have to use class component.
- In the class constructor we set the default value of the state.
- To display the values now we read from the state and not from props.
- In the onClick event we set the state to new value using the function setState().
- It is important to set the new value of sate using setState() so that react knows that the sate has changed and hence the component needs to be re rendered.

- State

```
class ClickCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
  }

  render() {
    return <div onClick={() =>
        { this.setState({clicks: this.state.clicks + 1}); }}>
        This div has been clicked {this.state.clicks} times.
      </div>;
  }
}
```

- setState
- The setState method merges the new state with the old state.
- Addition to changing the state of the component setState() also causes the component to be re-rendered.

- Prop Validation
- React components can validate the props they passed by defining a propTypes object.
- propTypes specifies the allowed values for each property.

```
import PropTypes from 'prop-types';

function Sum(props) {
  return <h1>{props.a} + {props.b} = {props.a + props.b}</h1>;
}
Sum.propTypes  = {
  a: PropTypes.number.isRequired,
  b: PropTypes.number.isRequired,
};

ReactDOM.render(<Sum a={"a"} b={2} />,
  document.getElementById('root')
);
```

- Jsx

JSX

```
<Sum a={4} b={3} />


<h1>
  <Sum a={4} b={3} />
</h1>
```

JavaScript

```
React.createElement(
    Sum,
    {a:4, b:3},
    null
)
React.createElement(
  'h1',
  null,
  React.createElement(
    Sum,
    {a:4, b:3},
    null
  )
)
```

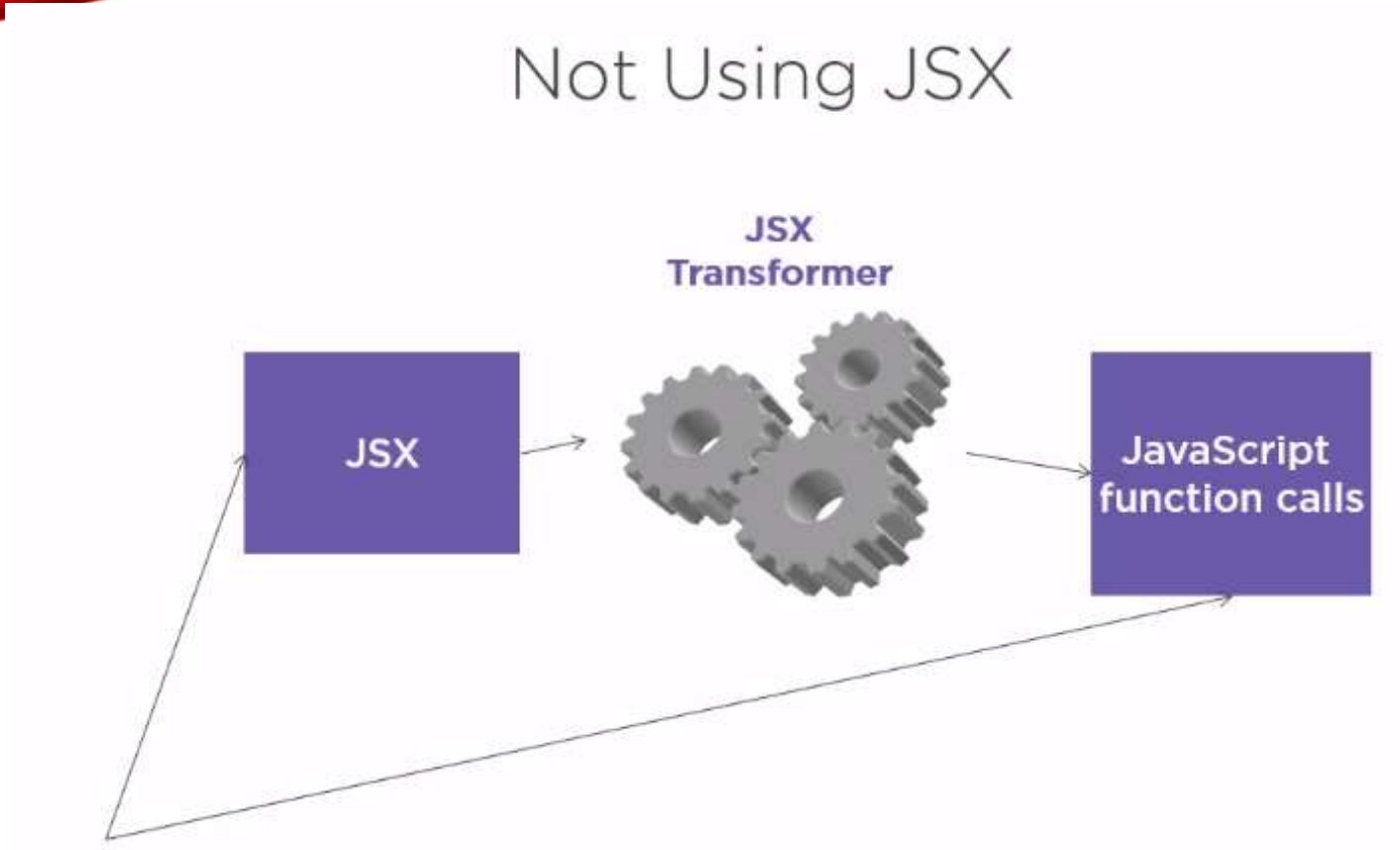
JSX

```
<label
   htmlFor="name"
   className="highlight"
   style={{
    backgroundColor:
    "yellow"
   }}
>
  Foo Bar
</label>
```

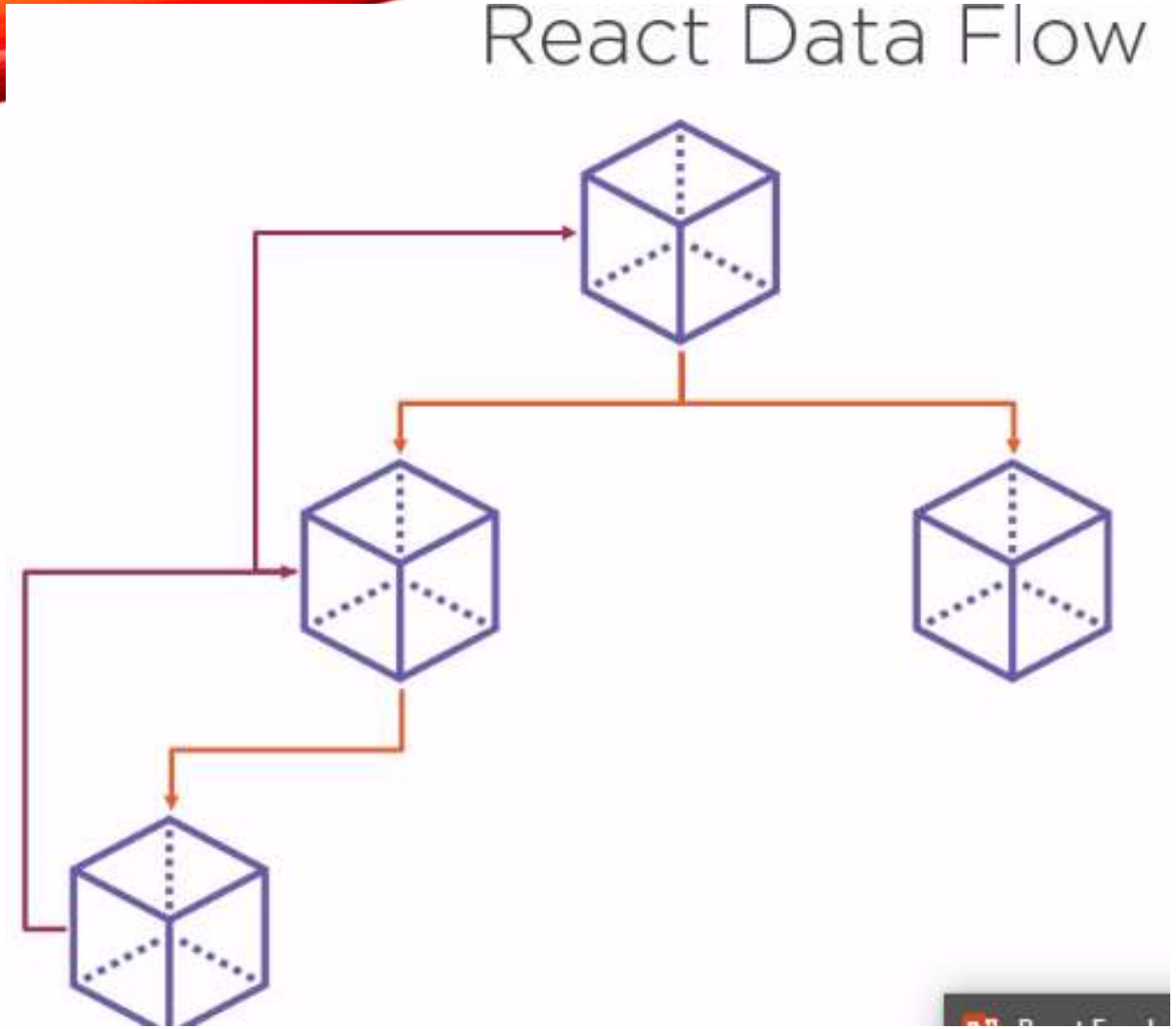
HTML

```
<label
   for="name"
   class="highlight"
   style="background-
color: yellow"
>
  Foo Bar
</label>
```

- Not To use JSX



React Data Flow



Data is passed from parent to child using PROPS
Data is passed using events in the reverse direction.

- **React Architecture**
- A React application is a composed set of components.
- Each component has the design represented in this diagram.
- The inputs to a component are properties, referred to as props, and state.
- The difference between the two is that state can change.
- As much as possible, we should design components that don't use state.
- Note that the flow of data is all in the same direction that is from the model, via the render function, to the DOM.
- Props and state collectively represent the model.
- The DOM is the direct result of rendering that model.

- **React Architecture**
- For a given model, the rendered DOM will always be the same.
- The way to change the DOM is to change the model.
- Once the DOM is rendered it can generate events, which feed back into the component state, and trigger another render cycle.
- Conceptually, we can imagine that for any state change React will regenerate the entire DOM, but if React actually worked that way the performance would be poor.
- Instead, React maintains its own document abstraction.
- A component's render function updates this fake DOM, which is extremely fast.
- Once that happens, it's the job of the React framework to compare its fake DOM to the current state of the real DOM and update the real DOM in the most efficient way possible.
-

- **React Architecture**
- React is a small, specialized tool.
- It provides two features.
- The first is rendering a model to the DOM and keeping the DOM synchronized with changes to the model.
- The second is publishing, handling, and managing events.
- React provides a declarative system for generating user interfaces as a function of model data.

- **React Architecture**
- React requires the programmer to notify components when the model has changed, at which time it regenerates its DOM abstraction and merges the changes into the real DOM.
- React and Knockout provide similar behavior, but a very different design.
- AngularJS and Ember both provide a much larger and more generalized set of features.
- AngularJS and Ember are intended to be frameworks that provide everything you need to build a single page application like routing, two-way binding, server data services, and much more.
- React is only responsible for the restricted, but very important task of transforming a model to a UI and handling events.

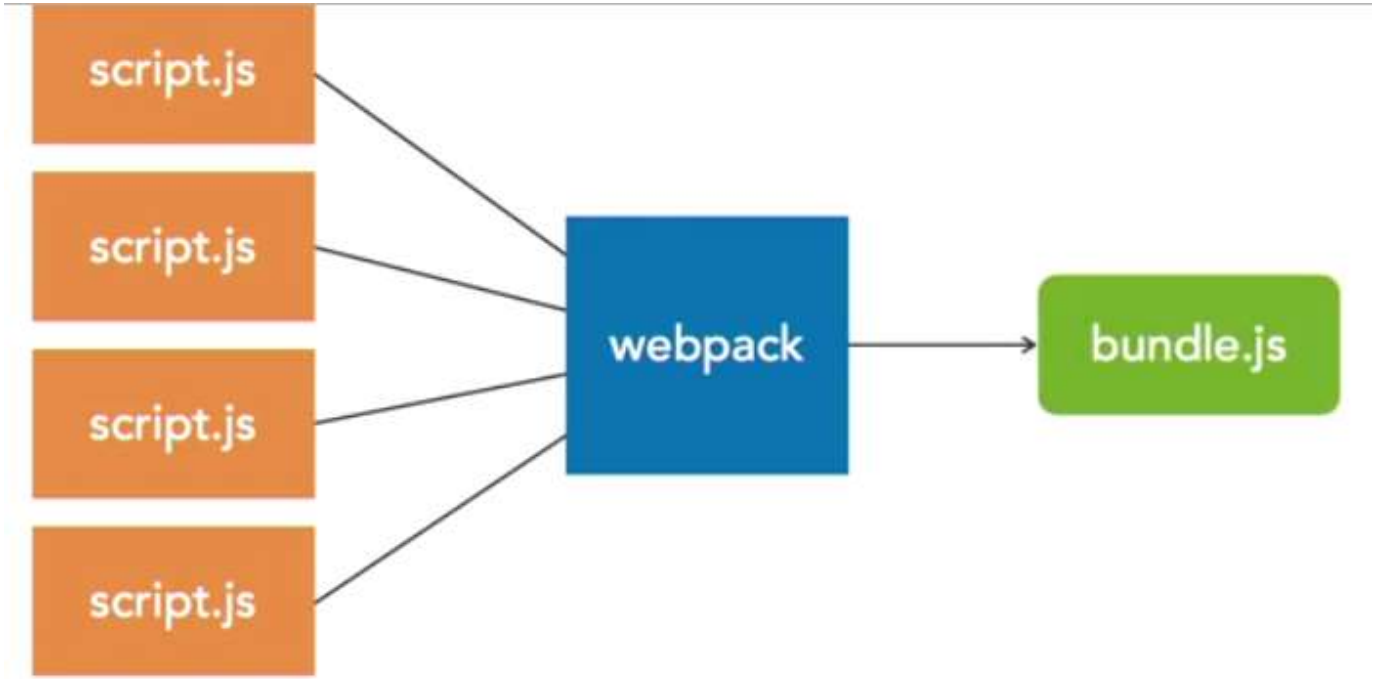
- **React Architecture**
- Large portions of the Facebook and Instagram websites are built with React.
- Although React has only recently been released as an Open Source project, it has been through some serious production testing on what is currently the second busiest website on the net.
- React is a good choice for your project if any of the following features are priorities for you, speed, declarative binding or composability.

- **React Architecture**
- Backbone is unique as the only library mentioned on this slide that does not provide declarative binding.
- We can have to imperatively specify how changes to a model affect the view.
- This is the advantage of allowing low-level control, but at the cost of more work for the programmer, and a higher risk of bugs.
- Again, Backbone is a much fuller featured library than React.
- In fact, React is roughly comparable to Backbone's view component and a lot of people choose to get started with React by using it as an alternative view component for Backbone.

BUILDING WITH WEBPACK

- we can use module bundlers like Webpack to handle some of this work for us.
- Webpack is a module bundler that helps us create static files and automate processes that need to happen before our files can go into production.
- we can use module bundlers like Webpack to handle some of this .

BUNDLING WITH WEBPACK



Think about a typical HTML file. We might load several different scripts, making several HTTP requests.

Webpack will run several commands at a time to create a bundle file.

This bundle packages scripts, dependencies, and even CSS into one file and one file means one request.

- Introduction

- Reacts official definition states that it's a JS library for building user interfaces.
- React is a JS library and not a framework.
- React is small and not a complete solution.
- Often we need more libraries with react to form any solution.
- React is a description of user interface in JS
- As react is declarative, we describe UI with react and tell what is required and not how to do it.
- Reacts takes care of the how and translates the declarative description which we write in react.
- With react we write declarative UI for HTML that represent dynamic data

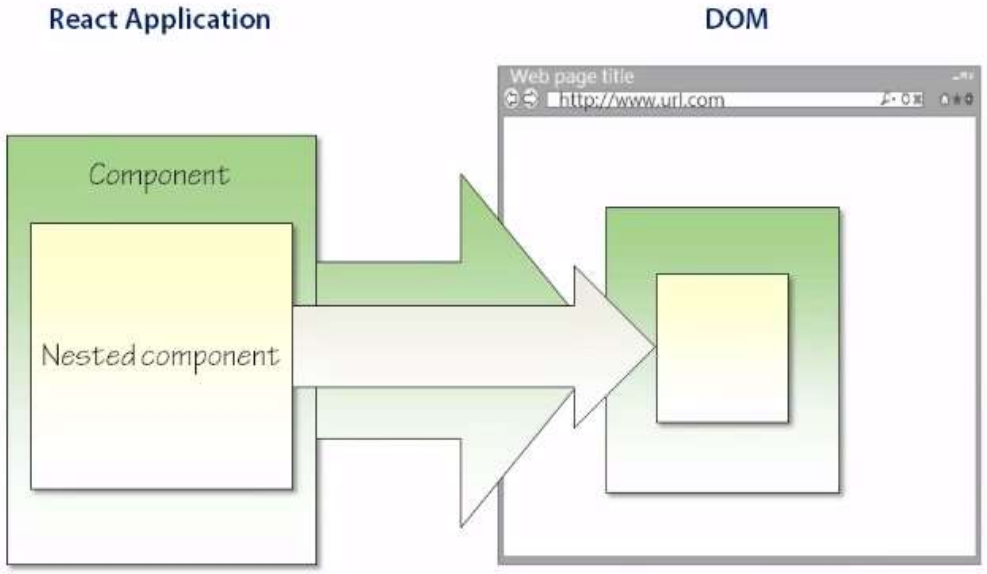
- React's three main design concepts that drive its popularity

- 1. Components
  - In react the UI's are described using the components.
  - Components are simple functions.
  - The input properties and states of components are called and the output what a component generates is the description of UI.
  - A single component can be reused in multiple user interfaces.
  - Components can be nested.
  - A react component can have a private state that hold data that may change over time.

Components

- Like functions
- Reusable and composable
- Can manage a private state

What is a Component?



- A React application, is built from a set of components.
- What is a component?
- Components are the fundamental unit of a React application.
- They are both simple and powerful.
- Each component corresponds to an element in the DOM.
- The component is responsible for rendering the content of that element and for handling any events that occur within it.
- Components can be nested inside each other and this is meant by composing components and it's a powerful technique for achieving reuse.
- Nested components correspond to nested DOM nodes.
- The outer component is said to own the inner component.

- React's three main design concepts that drive its popularity

- 2. Reactive updates
  - React's name is a simple explanation for this concept of react.
  - Reactive updates means when the state of a component i.e. the input, changes, the UI it represents, the output, changes as well.
  - The change in description of UI needs to be reflected in the device we're working with.
  - In react we need not worry about how to reflect these changes.
  - React will simply react to the state changes and update the DOM automatically when needed.

Reactive updates

- React will react
- Take updates to the browser

- React's three main design concepts that drive its popularity

- 3. the virtual representation of views in memory
  - With react we write HTML using JS.
  - The power of JS is relied upon to generate HTML that depends on some data rather than enhancing HTML to make it work with data.
  - Other JS frameworks like Angular (directives)usually uses the approach of enhancing the html to work with data .
  - React uses JS to render HTML and hence allows react to have a virtual representation of HTML in-memory, which is known as virtual DOM.
  - React uses this concept to render HTML virtually first and then every time a state changes and a new HTML tree needs to be written to the DOM, instead of writing the whole tree, react will only write the difference between the new tree and the old HTML tree since react has both trees in the memory.
  - This process is called **Tree reconciliation,** which is one of the best thing that has happened in web development after AJAX.

- React's three main design concepts that drive its popularity

Virtual views in memory

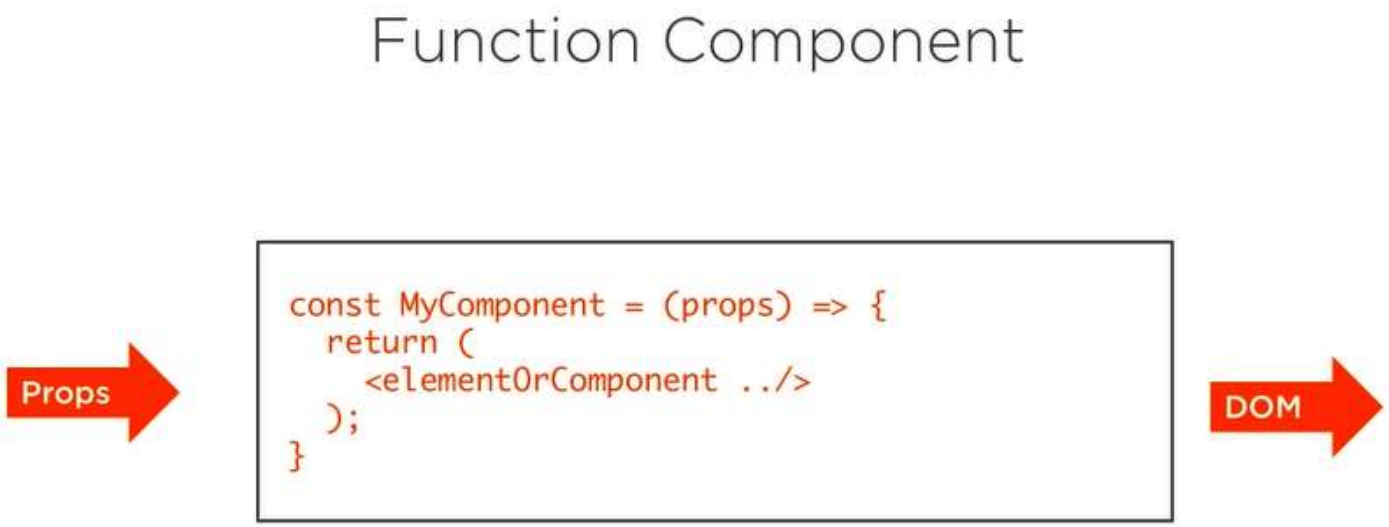
- Write HTML in JavaScript
- Tree reconciliation

• React Components

- React components can be one of the following two types
  - Function Component
  - Class Component
  - They are alternatively called as stateless and state full respectively
- To create a component in React we
  - Need a react library
  - Need a React-DOM library to render the components
  - React component needs at least one method called render.
  - The render method sets up what will be sent to the virtual DOM for rendering.
  - Render method using JSX Syntax.
  - Defining what's going to be rendered is not the end.
  - After using Render method, the ReactDOM's render method is called, which is finally renders the html to DOM.

- Function Component

- A function component is the simplest form of a React component.
- It's a function with simple contract.
- It receives an object of properties, called as **"props"** in react and returns what looks like HTML, but its really a special JavaScript syntax called **"JSX".**
- JSX stands for JavaScript XML

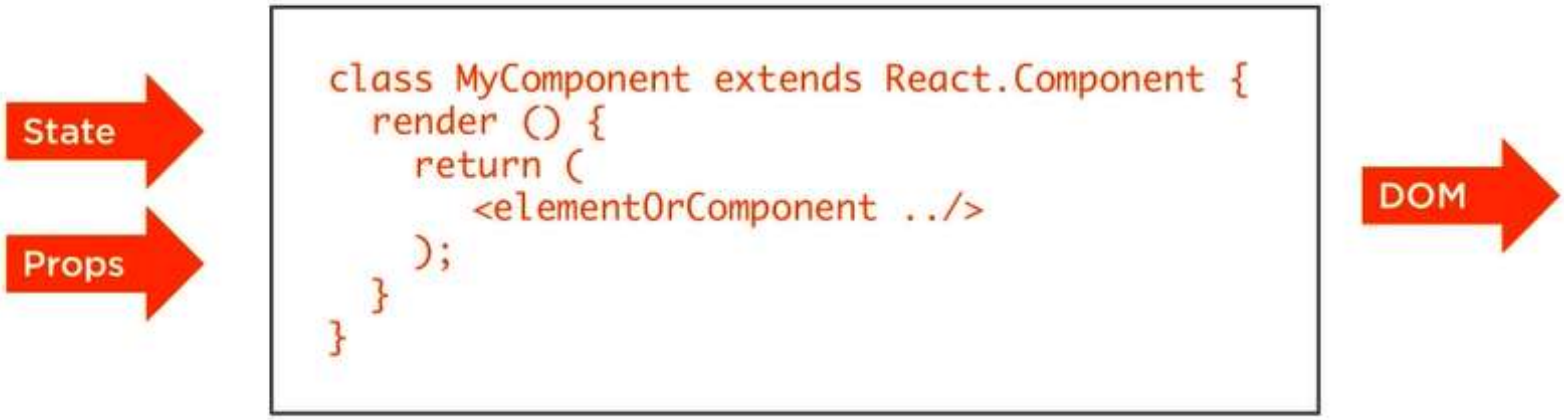


- Class Component

- A class component is a more featured way to define react component.
- It also acts like a function that receives props, but also considers a private internal state as additional input that controls the returned JSX.
- This private internal state is what gives React its reactive nature.
- When the state of the class component changes React will automatically re-render that component.
- One important difference between state and props is state can be changed while the props are all fixed values.
- Class components can only change their internal state, not their properties.

- Class Component

Class Component



- Virtual DOM and JSX

Virtual DOM and JSX

```
class Hello extends React.Component {
  render () {
    return (
      <div className="container">
        <h1>Getting Started</h1>
      </div>
    );
  }
}

ReactDOM.render(<Hello />, mountNode);
```

```
class Hello extends React.Component {
  render () {
    return (
      React.createElement("div",
        { className: "container"},
        React.createElement("h1", null, "Getting Started")
      )
    );
  }
}
ReactDOM.render(React.createElement(Hello, null), mountNode);
```

- Introduction

```
class Hello extends React.Component {
  render () {
    return (
      <div className="container">
        <h1>Getting Started</h1>
      </div>
    );
  }
}

ReactDOM.render(<Hello />, mountNode);
```

Here the component is written in special JSX syntax.
JSX allows to describe the DOM syntax very close to the DOM's we are used to.
Using JSX is optional when working with React.

- Introduction

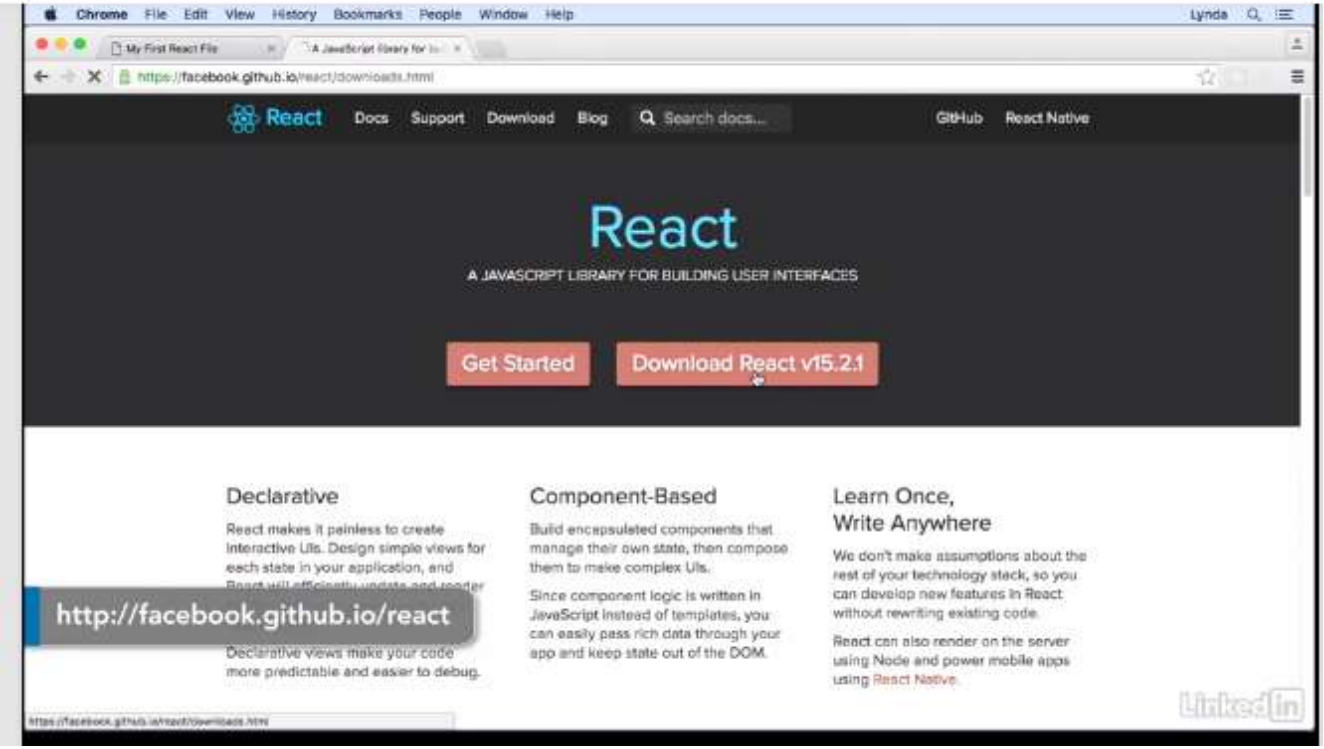
```
class Hello extends React.Component {
  render () {
    return (
      React.createElement("div",
        { className: "container"},
        React.createElement("h1", null, "Getting Started")
      )
    );
  }
}
ReactDOM.render(React.createElement(Hello, null), mountNode);
```

React compiles JSX to pure javascript and works with the JS in the browser.

The arrow is pointing to the JS representation of the DOM which react has translated to DOM operations that it performs in the browser.

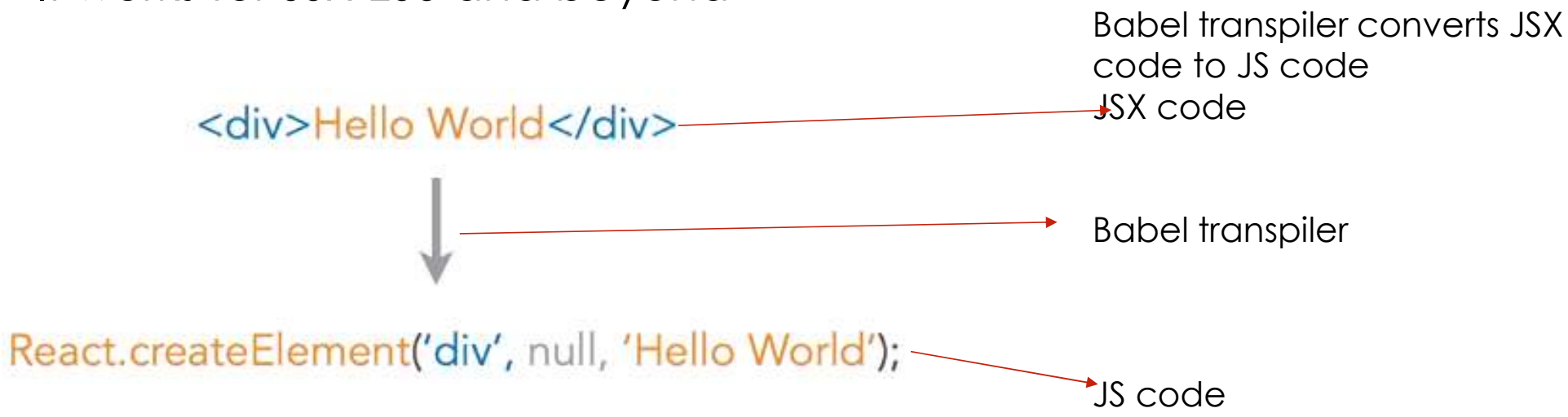
• React.js Syntax

Download react from
http://facebook.github.io/react



• Preprocessing JSX with babel

- Babel is a transpiler that will transpile JS code
- It works for JSX ES6 and beyond



- Introduction

- What is areact component

- When we are talking about react applications we should be thinking about components
- Components in React are small user interface elements that displays data as it changes over time
- These components are the composed together nested inside one another to create an entire user interface.
- Websites can use react for all of the UI or just bits or pieces of it
- Nordstrom.com is a website that uses react

- Components

- Components can be created in three ways
  - A simple react component
  - A react component with ES6 syntax
  - A stateless react component: A stateless functional component is just a simple function that returns react elements.

- Props And States

- Using Properties:
  - The Components of react can be made more dynamic by adding properties.
  - Sending properties to a component is similar to adding attributes to HTML
  - The " props " property can be used to create reusable component.
  - React is all about displaying dynamic content and props helps to display this dynamic content very easily

  - Using States:
    - One of the most important react concepts is state.
    - When the components state data changes the render function will be called again to re-render the change in state.
  - Using Refs:
    - Refs are a reliable way to access the values of an underlying DOM node
    - So if the form element is to reached where values cannot be accessed with props and state refs might be useful.

- PropTypes

- React has a really nice built in utility to handle data types of props.
- The first way that we can handle this is by adding prop types. Prop types are an optional feature.
- It serves as documentation about how you wish your components to work, and what values you expect for them.
- Prop types is going to be set equal to an object, and then you just define your properties inside as different keys.
- This is going to be set equal to a function that will take in props.

- **Keys**
- Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:
- const numbers = [1, 2, 3, 4, 5];
- const listItems = numbers.map((number) => <li key={number.toString()}> {number});
- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings.
- Most often you would use IDs from your data as keys
- const todoItems = todos.map((todo) =>
  - <li key={todo.id}> {todo.text}
  -);

- KEYS

- When you don't have stable IDs for rendered items, you may use the item index as a key as a last resort:

- KEYS
- **Recursing On Children**
- By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.
- For example, when adding an element at the end of the children, converting between these two trees works well:
-
- first
- second
-
- first
- second
- third
-

- KEYS
- React will mutate every child instead of realizing it can keep the Duke and Villanova subtrees intact. This inefficiency can be a problem.

- KEYS

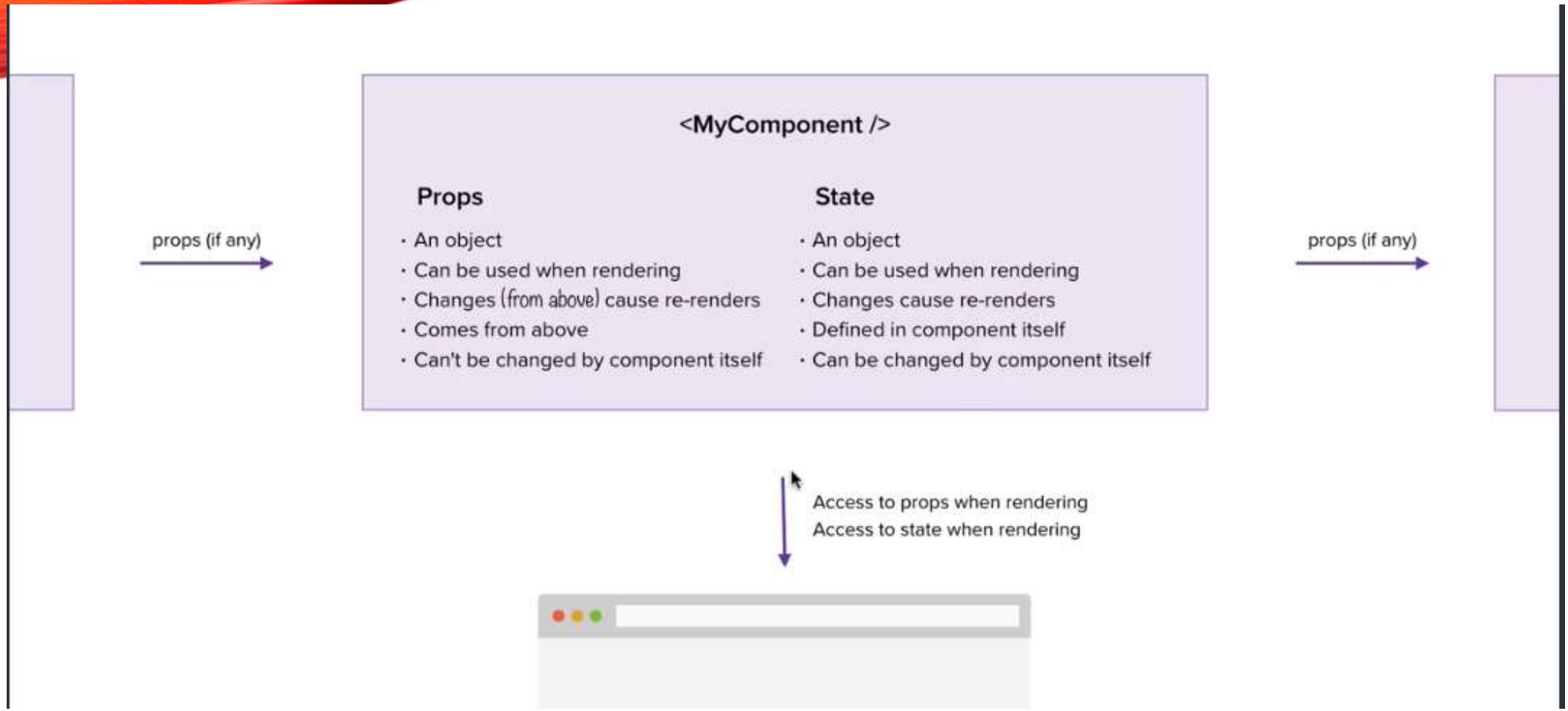
React will match the two first trees, match the two second trees, and then insert the third tree.

f you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

 Duke
Villanova

Connecticut
Duke
Villanova

- Summary of Props and State



- The Component Life Cycle
- The component lifecycle provides hooks for creation, lifetime, and teardown of components.
- These methods allow you to do things like add libraries, load data, and more at very specific times.
- The **Mounting** lifecycle has several methods for example,
  - getInitialState is going to be called once and will set the default for a state.
  - componentWillMount is called right before the render, and it's the last chance to effect state prior to the render.
  - The render method is the only required method.
  - componentDidMount is going to fire right after the render, so after a successful render, we can now access the DOM, the component has been rendered, and now the user can interact with it.

- Life cycle methods
- The following methods are called when a component is re-rendered to the DOM

- componentDidUpdate(): called when the state of a component changes. Perfect place to update UI or make network calls based on previous state before update, and current state

- Life Cycle Methods
- The following methods are called when a component is being added to the DOM:

- constructor() : called before component is mounted. NEVER put side effect code inside of the constructor. Use ComponentDidMount for that instead. Commonly used to initialize state or bind methods.
- componentWillMount(): invoked immediately before mounting occurs. Called before render. Once again, DO NOT put any side effect code inside of componentWillMount, and do not make API calls in this method .
- render(): never fetch data inside of render. Should only be used to return elements.
- componentDidMount(): Perfect place to fetch data. It gets called after render. This makes it clear that the initial state is empty at first, until we fetch it and re-trigger render. This forces us to set up our initial state properly, otherwise you're likely to get undefined states.

- The **Mounting** Life Cycle
- The component lifecycle also provides methods for updating.
  - componentWillReceiveProps, once this method is called, we get the opportunity to change the object and effect state.
  - shouldComponentUpdate and componentWillUpdate are invoked right before rendering, and are often used for optimization. These methods are called if something has changed.
  - Finally, we'll have the render method here again, and that's going to be part of the updating lifecycle as well
  - componentDidUpdate is going to fire right after everything in the DOM has been updated.

- The **Mounting** life cycle
  - componentWillUnmount is called right before the component is unmounted.This can help us do things like clean up DOM elements and invalidate timers. So when componentWillUnmount is called on the parent, all of the children are unmounted as well.
- The component lifecycle has many different methods that can be used to optimize your applications.

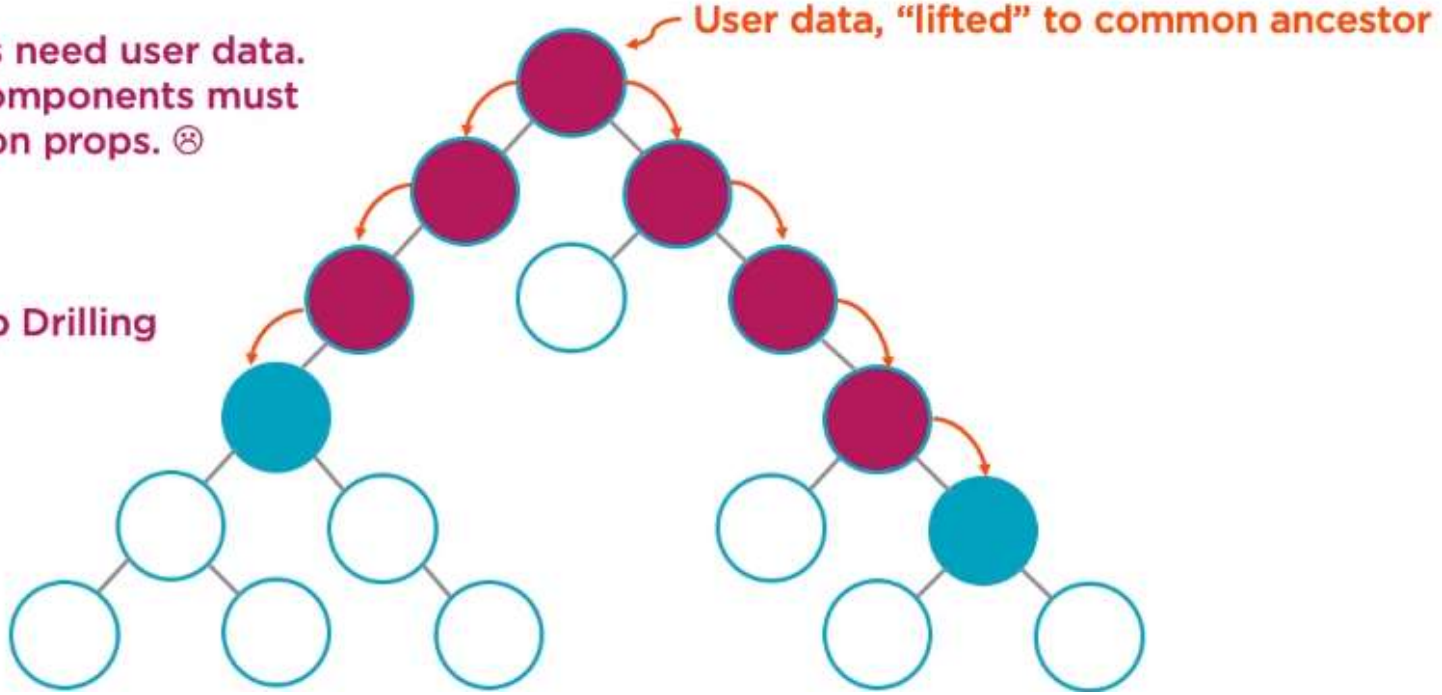
- Intro To Redux : Do I need Redux?

3 Solutions

1. Lift State

2 components need user data.
But 6 other components must
pass it down on props. ☹

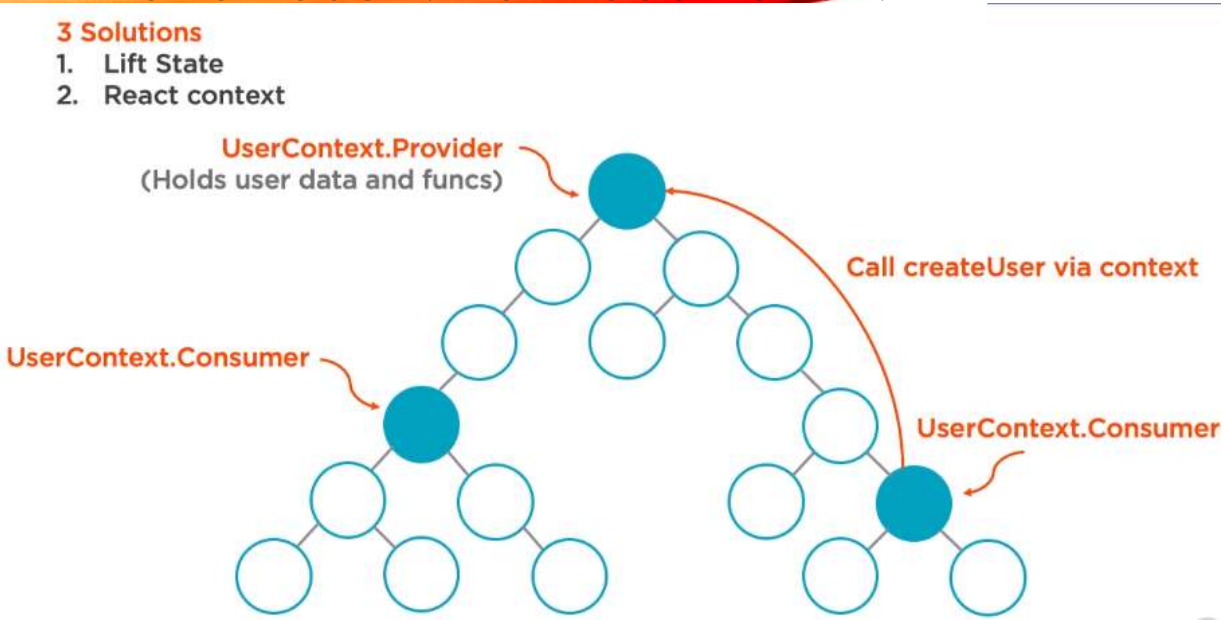
Problem: Prop Drilling



- Intro To Redux : Do I need Redux?
- Imagine this is your app's React component tree.
- Each circle represents a React component in your app.
- The top circle represents your top-level app component, which passes data down to these child components via props.
- What if these two components need to work with the same data.
- They're in very different parts of your app, so how can they work with the same data? For example, imagine these components in blue need user data.
- Each could make an API call to retrieve user data from a database, but that could be wasteful and redundant.
- And how would they communicate to assure that the user data that they're using stays in sync?

- Intro To Redux : Do I need Redux?
- There are three popular options to handle this.
- The first option is to **lift state** to their common parent.
- In this case, it means moving the user data all the way up to the top-level component since that's their only common ancestor.
- This is annoying though because it means that you have to pass data down on props through all these other components.
- These components didn't need the user's data.
- We're just passing the data down on props to avoid storing the same data in two spots.
- We added the user prop to these six components merely to pass the data down to the two components that need it.
- This problem is commonly called prop drilling.
- That said, lifting state does work, and it's a good first step for small and mid-size applications.
- But on larger apps that need to display the same data in many spots, lifting state becomes tedious, and it leads to components with many props that exist merely to pass data down.

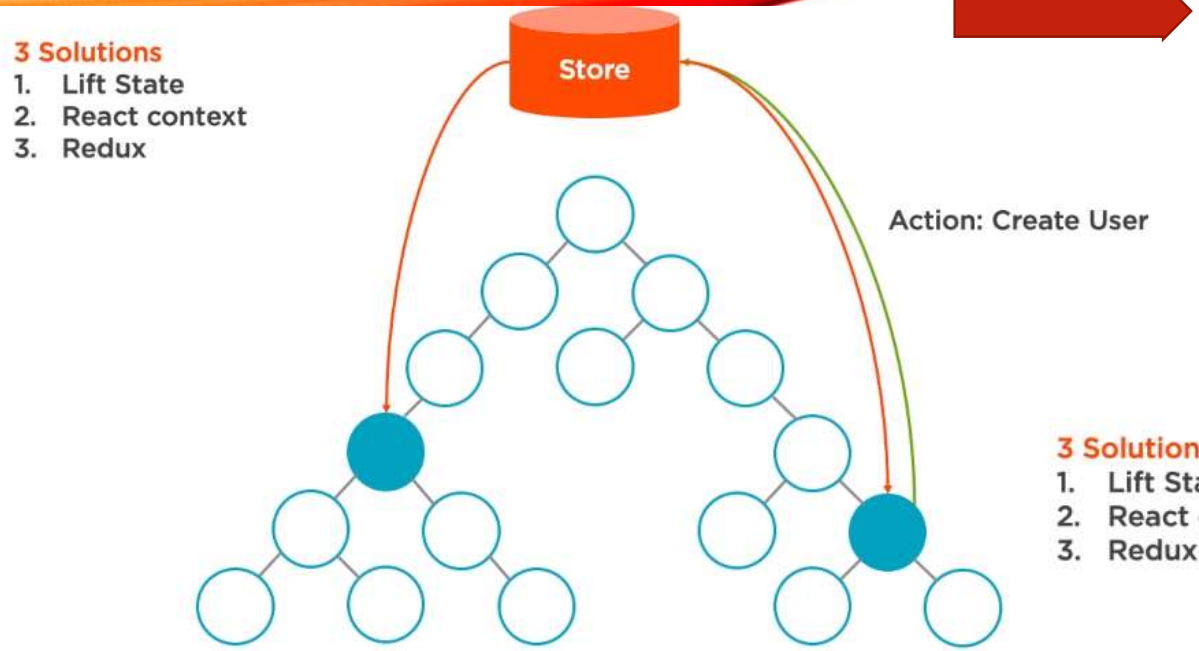
- Intro To Redux : Do I need Redux?

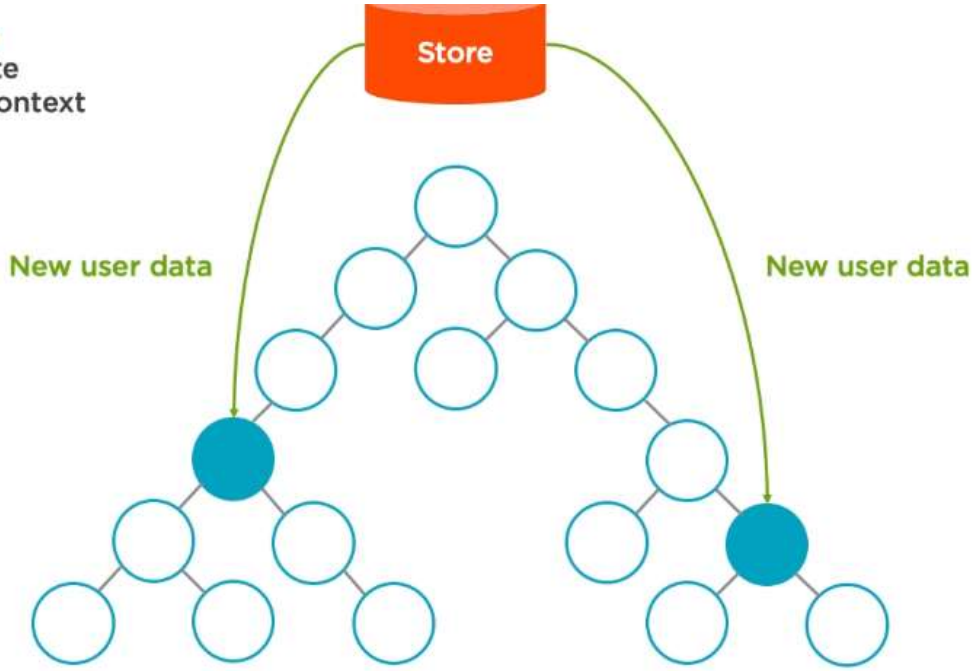


- The second way to solve this problem is to consider **React's context.**
- With React's context, we can expose global data and functions from a given React component.
- To access this global data, you import the context and consume it in your component.
- So it's not truly global data.
- You must explicitly import the context to consume it.

- . For example, the top-level component could declare a UserContext.Provider.
- This provider would provide user data and relevant functions to any components that want to consume it.
- So the two components that need the user data can import that UserContext and thus access the user's information via UserContext.Consumer.
- Since context can also expose functions for global usage, the consuming component can also call the createUser function that's actually declared way up here in the top- level component if it's exposed via the UserContext.Provider.
- . If you haven't tried context, it's certainly worth a look. It's build into React, it's quite elegant, and it's easy to learn

- Intro To Redux : Do I need Redux?





- Intro To Redux: Do I need Redux?
- The third option to consider is Redux.
- With Redux, there's a centralized store.
- Think of the store like a local client-side database.
- This is a single spot where the app's global data is stored.
- Any component can connect to the Redux store.
- The Redux store can't be changed directly though.
- Instead, any components can dispatch an action, such as Create User.
- When the action is dispatched, the Redux store is updated to reflect the new data.
- And any connected components receive this new data from the Redux store and rerender.
- So does Redux sound useful to you? If so, let's move on and learn about the three principles at the foundation of Redux's design.

When is Redux Helpful?

Complex data flows

Inter-component communication

Non-hierarchical data

Many actions

Same data used in many places

- The bottom line is you can build impressive applications with just React.
- But Redux can be useful for applications that have complex data flows.
- If you're writing an app that merely displays static data, then Redux isn't likely to be useful.
- If you need to handle interactions between two components that don't have a parent- child relationship, then Redux offers a clear and elegant solution.
- When you find two disparate components are manipulating the same data, Redux can help.
- This scenario is often the case when your application has non-hierarchical data.
- Also, as your application offers an increasing number of actions, such as writes, reads, and deletes for complex data structures, Redux's structure and scalability becomes increasingly helpful.

- When Is Redux Helpful?
- The most obvious sign that you'll want something like Redux is if you're utilizing the same data in many places.
- If your components need to utilize the same data, and they don't have a simple parent- child relationship, Redux can help.
- Pete Hunt boils all this down. You'll know when you need Redux.

> "...If you aren't sure if you need it, you don't need it."
>
> **Pete Hunt**

- When Is Redux Helpful?

My take

1. Start with state in a single component
2. Lift state as needed
3. Try context or Redux when lifting state gets annoying

Each item on this list is more complex, but also more scalable.
Consider the tradeoffs.

- Three Core Redux Principles

Redux: 3 Principles

One immutable store

Actions trigger changes

Reducers update state

Example action:

```
{
  type: SUBMIT_CONTACT_FORM,
  message: "Hi."
}
```

- Three Core Redux Principles
- Redux has three core principles.
- The first is that your application's state is placed in a single, immutable store.
- By immutable, we mean that state can't be changed directly.
- Having one immutable store aids debugging, supports server rendering, and it makes things like undo and redo easily possible.
- In Redux, the only way to change state is to emit an action, which describes a user's intent.
- For example, a user might click the Submit Contact Form button, and that would trigger a SUBMIT_CONTACT_FORM action.
- The final principle is that state changes are handled by pure functions.
- These functions are called reducers, which sounds complicated, but it's not.
- **In Redux, a reducer is just a function that accepts the current state in an action, and it returns a new state.**

- Flux vs Redux

- We don't have to know Flux to work with Redux.
- But if you happen to know Flux, that will certainly help you pick up Redux more quickly.
- Flux was the precursor to Redux.
- But today, Redux is much more popular than Flux.
- Let's begin our comparison of Flux and Redux by discussing what they have in common.
- Flux and Redux are two different ways that you can handle state and data flows in your React applications.
- Both Flux and Redux have the same unidirectional data flow philosophy.

- Flux vs Redux - Similarities
- Data flows down, and actions flow up. So the first similarity is that both Flux and Redux enforce unidirectional data flows.
- All data flows in one direction.
- They also both utilize a finite set of actions that define how state can be changed.
- We define action creators to generate those actions and use constants that are called action types in both as well.
- They both have the concept of a store that holds state, although Redux typically has a single store, while Flux typically has multiple.
- While these core concepts exist in both, they differ in a variety of ways.
- Let's explore how they're different.

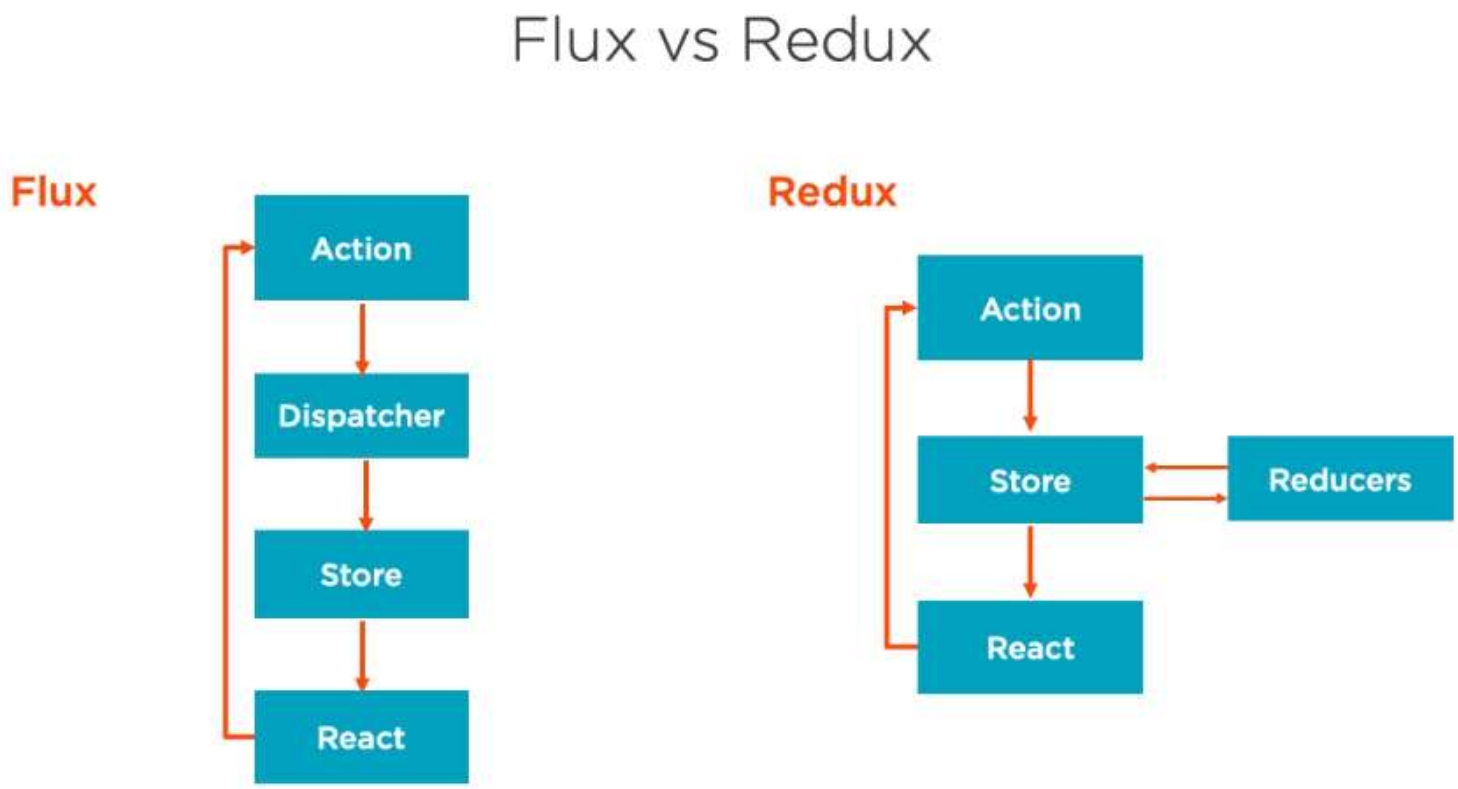
- Flux vs Redux :Differences
- Redux introduces a few new concepts.
- Reducers are functions that take the current state in an action and then return a new state.
- So reducers are pure functions.
- Containers are just React components, but their use is specific.
- Container components contain the necessary logic for marshalling data and actions, which they pass down to dumb child components via props.
- This clear separation helps keep most of the React components very simple, pure functions that receive data via props.
- This makes them easy to test and simple to reuse.
- The third new concept is immutability.
- The Redux store is immutable.

- Flux vs Redux : differences

Redux: New Concepts

- Flux vs Redux : Differences
- Flux has three core concepts: actions, dispatchers, and stores.
- When actions are triggered, stores are notified by the dispatcher.
- So Flux uses a singleton dispatcher to connect actions to stores.
- Stores use eventEmitter to connect to the dispatcher.
- So in Flux, each store that wants to know about actions needs to explicitly connect itself to the dispatcher by using eventEmitter.
- In contrast, Redux doesn't have a dispatcher at all.
- Redux relies on pure functions called reducers, so it doesn't need a dispatcher.
- Pure functions are easy to compose, so no dispatcher is necessary.
- Each action is ultimately handled by one or more reducers, which update the single store.
- Since state is immutable in Redux, the reducer returns a new, updated copy of state, which updates the store.

- Flux vs Redux : Differences



- Flux vs Redux : Differences
- Let's contrast Flux and Redux further by exploring the specific ways that they differ.
- In Flux, stores do more than one thing.
- They don't just contain application state but they also contain the logic for changing state.
- Redux honors the single responsibility principle by separating the logic for handling state. Redux handles all state- changing logic with reducers.
- A reducer specifies how state should change for a given action. So a reducer is a function that accepts the current state in an action and returns an action.
- Flux supports having multiple stores.
- In a Flux app, we may have a user store and a product store.
- We can have as many stores as you like.
- In Redux, you typically only have one store. This sounds constraining, but as we'll see, there are some significant advantages to the simplicity of having a single store.

- Flux vs Redux : Differences
- Having a single source of truth helps avoid storing the same data in multiple places and avoids the complexity of handling interactions between stores.
- One common struggle in Flux is how to deal with stores that interact with one another.
- In Flux, the stores are disconnected from each other, though Flux does at least provide a way for stores to interact in order via the waitFor function.
- In contrast, Redux's single store model avoids the complexity of handling interactions between multiple stores.
- This conceptually simpler model also provides some unique .
- In order to handle more complex stores with many potential actions, you can utilize multiple reducers, and you can even nest them.
- See in Flux, stores are flat.
- But in Redux, reducers can be nested via functional composition just like React components can be nested.
- So Redux gives you the same power of composition and nesting in your reducers as you have in your React component model.

- Flux vs Redux
- In Flux, a dispatcher sits at the center of your app.
- The dispatcher connects your actions to your stores.
- In Redux, there is no dispatcher because Redux's single store just passes actions down to the reducers that you define.
- It does so by calling a root reducer that you define.
- Reducers are pure functions.
- So in Redux, there's no need for Flux's eventEmitter pattern.
- In Flux, you have to explicitly subscribe your React views to your stores using onChange handlers in eventEmitter.
- In Redux, this can be handled for you using React-Redux.
- React-Redux is a companion library that connects your React components to the Redux store.
- Finally, in Flux, you manipulate state directly.
- It's mutable.
- In Redux, state is immutable, so you need to return an updated copy of state rather than manipulating it directly.

- Flux vs Redux Differences

Flux	Redux
Stores contain state and change logic	Store and change logic are separate
Multiple stores	One store
Flat and disconnected stores	Single store with hierarchical reducers
Singleton dispatcher	No dispatcher
React components subscribe to stores	Container components utilize connect
State is mutated	State is immutable

- Action Stores and Reduces:
- Actions
- In Redux, the events happening in the application are called actions.
- Actions are just plain objects containing a description of an event.
- An action must have a type property.
- The rest of its shape is up to you.
- The data under a property called rating which could be a complex object, a simple number, a Boolean, or really any value that's serializable.
- The only thing that you shouldn't try passing around in your actions are things that won't serialize to JSON like functions or promises.
- Actions are typically created by convenience functions that are called action creators.
- Action creators are considered convenience functions because they're not required, but recommend.
- By using these action creators to create your actions, the spot where you dispatch the action doesn't need to know about the action creator structure.

- Actions, stores and Reducres
- . By using these action creators to create your actions, the spot where you dispatch the action doesn't need to know about the action creator structure.
- The app that we're creating will work with course data, so it will have actions like loadCourse, createCourse, and deleteCourse.
- When actions are dispatched, it ultimately affects what data is in the store.
-

Action Creators

```
rateCourse(rating) {
   return { type: RATE_COURSE, rating: rating }
}
```

- Actions,Stores, reducers
- Store
- In Redux, we create a store by calling createStore in your app's entry point.
- We pass the createStore function to the reducer function.
- The Redux store honors the single responsibility principle because the store simply stores the data, while reducers,handle state changes.
- We might be concerned that there's only one store in Redux, but this is a key feature.
- Having a single source of truth makes the app easier to manage and understand.
- The Redux store API is quite simple.
- The store can dispatch an action, subscribe to a listener, return its current state, and replace a reducer.
- The most interesting omission here is that there's no API for changing data in the store.
- It means that the only way that you can change the store is by dispatching an action.
- The store doesn't actually handle the actions that are dispatched, actions are handled by reducers.

- Actions,Stores, reducers

- Actions,Stores, reducers
- What is Immutability?
- What Is Immutability?
- Immutability is a fundamental concept in Redux.
- It just means that instead of changing your state object, we must return a new object that represents your application's new state.
- It's worth noting that some types in JavaScript are immutable already, such as numbers, string, Boolean, undefined, and null.
- In other words, every time you change the value of one of these types, a new copy is created.
- Mutable JavaScript types are things like objects, arrays, and functions.

What's Mutable in JS?

Immutable already! ☺	Mutable
Number	*Objects*
String,	*Arrays*
Boolean,	*Functions*
Undefined,	
Null	

```
state = {
    name: 'Cory House'
    role: 'author'
}
```

◀ Current state

Muttable

```
state.role = 'admin';
return state;
```

◀ Traditional App - Mutating state

```
state = {
    name: 'Cory House'
    role: 'author'
}
```

◀ Current state

Immutable

```
return state = {
    name: 'Cory House'
    role: 'admin'
}
```

◀ Returning new object.
Not mutating state! ☺

- Actions,Stores, reducers
- What is Immutability
- To help understand immutability, let's consider an example.
- Imagine that our app state holds my name and my role.
- In a traditional app, if I wanted to change state, we'd assign a new value to the property that we want to change.
- So here we're mutating state because we're updating an existing object to have a new value for role.
- Let's contrast this approach with the immutable approach to updating state. Here you can see that I'm not mutating state. Instead, I'm returning an entirely new object.

- Actions,Stores, reducers

- Actions,Stores, reducers

- Actions,Stores, reducers

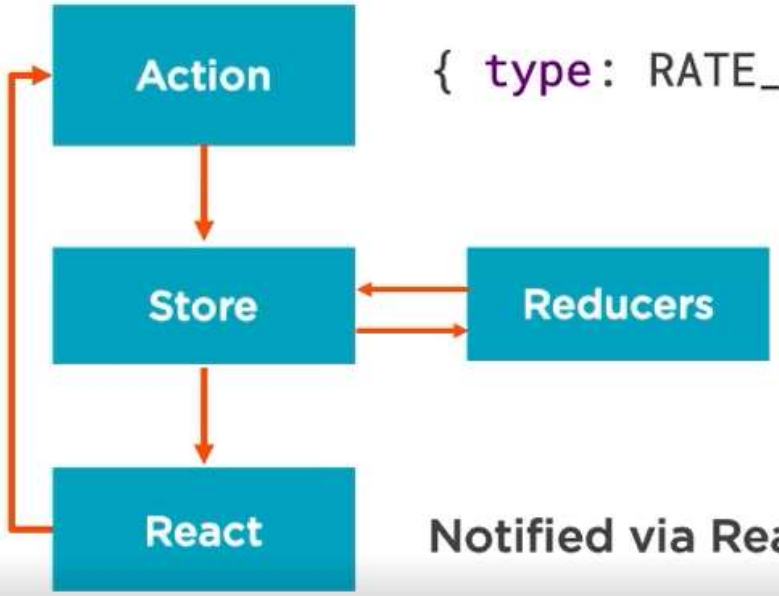
- Actions,Stores, reducers

- Actions,Stores, reducers

- History Of Redux
- In order to understand Redux, we must first take a look at Flux.
- Flux is a design pattern developed at Facebook to provide an alternative to MVC or MVP or MVVM.
- These are all variations of the model-view-controller design pattern.
- Each of these patterns allow for two-way data communication between models and views.
- For example, these patterns have models that are used to manage the data within an application.
- Now, the data from the model is used in a view that are objects that present data as user interface.
- Models can feed data to multiple views.
- When the user interacts with a view, the view, through a controller or a presenter, may update the data stored in a model.
- Changing data in a model can trigger data changes in other views.

- The Redux Flow

Redux Flow

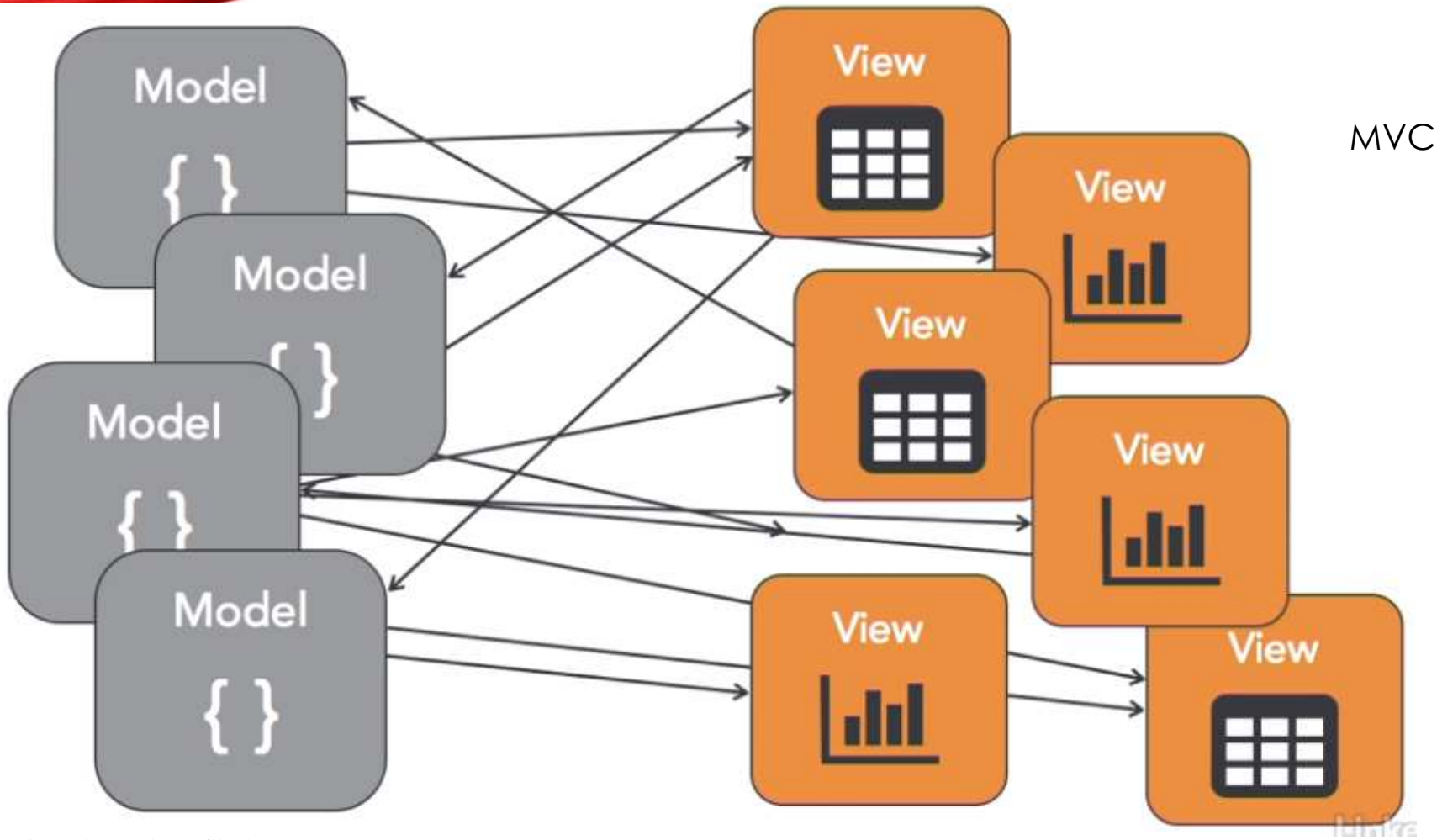



```
{ type: RATE_COURSE, rating: 5 }
```

```
function appReducer(state = defaultState, action) {
    switch(action.type) {
        case RATE_COURSE:
        //return new state
    }
}
```

Notified via React-Redux

- History Of Redux

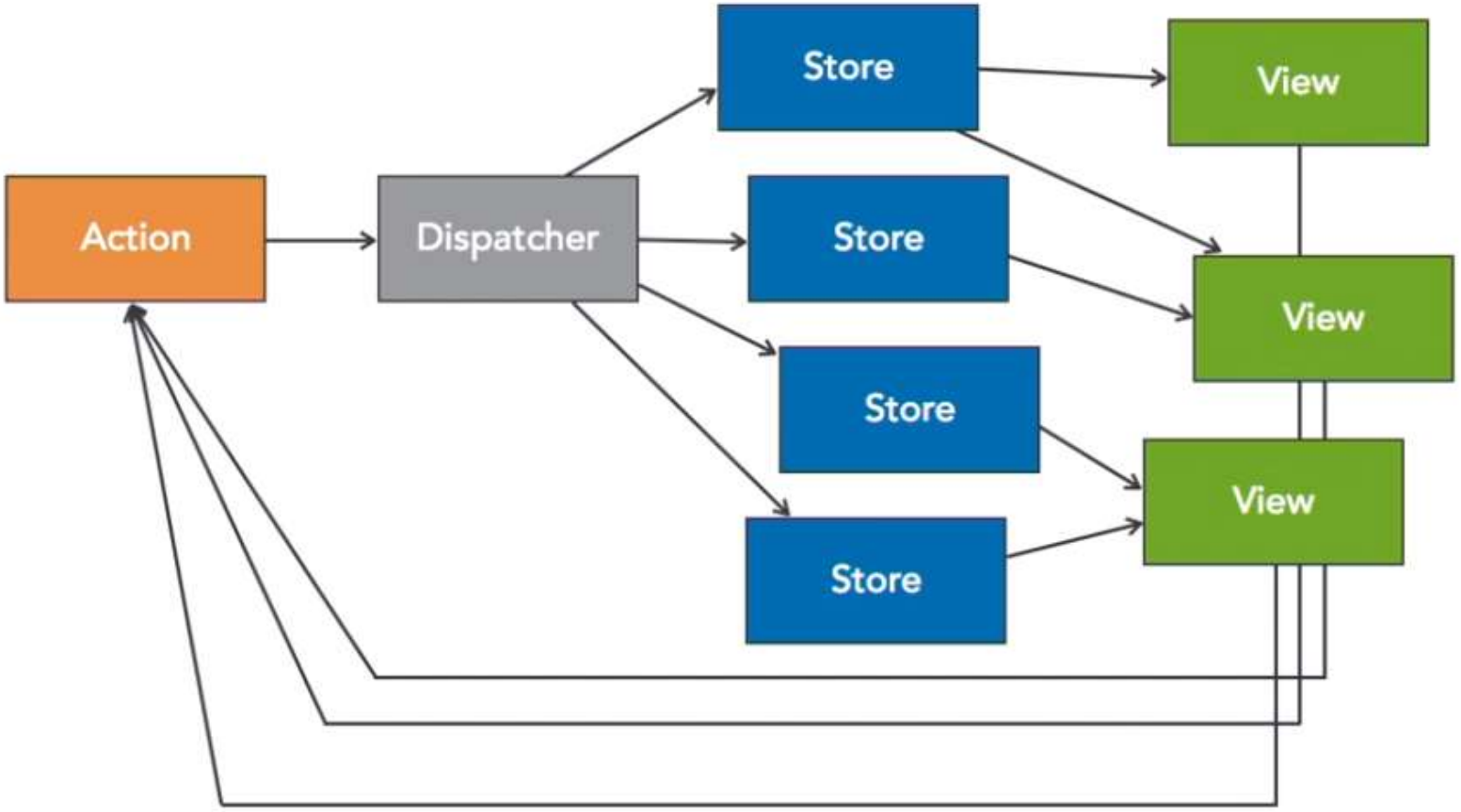


MVC Pattern

- History Of Redux
- In addition to having many models and many views, large-scale applications also have many developers and teams working on the same codebase.
- Making changes can be intimidating at this level.
- So, Facebook developed a new approach, Flux.
- In Flux, the data flows in one direction.
- Changes are initiated with actions.
- Actions are objects that describe what should change about the data.
- Actions are dispatched with the dispatcher.
- The dispatcher is an object that sends the action to the appropriate store.
- The store holds the data. It's like a model, but they're not exactly the same.
- The store is responsible for updating or changing its data.

- History Of Redux
- Finally, when the store updates the data, that change updates the view.
- The screen changes, reflecting the data back to the user.
- Now, if the user interacts with the view, a new action is generated and the process starts all over again.
- Data flows in one direction.
- As a Flux application grows, it may include more stores and more views, but the dataflow remains unidirectional.
- All changes in a Flux application begin by dispatching actions and end with updating views.
- Flux itself is not a library, it's a design pattern that can be implemented with JavaScript.
- The Facebook Flux library only includes a generic dispatcher that you can use in your Flux applications.

- History Of Redux



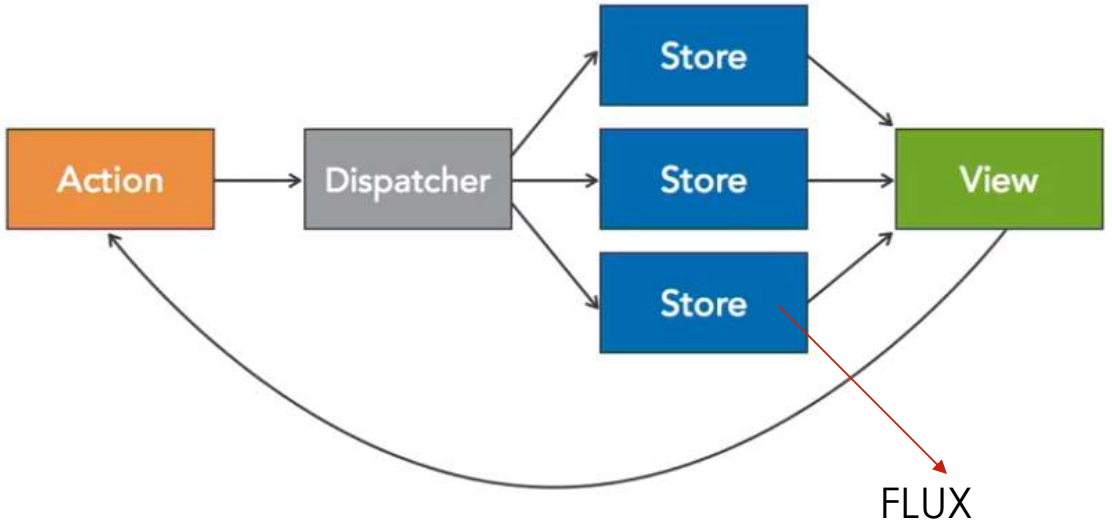
The Flux Pattern

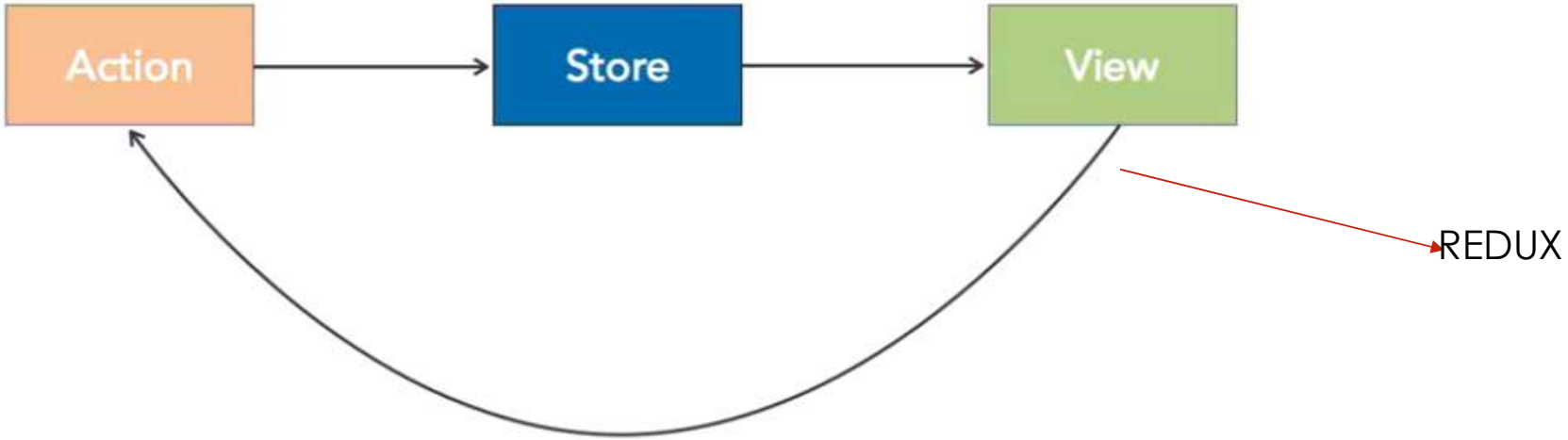
- History Of Redux
- So, the community got to work, and GitHub exploded with Javascript libraries that represent variations of this design pattern.
- Each of these libraries represent a different implementation of Flux.
- We have Flummox, Alt, Reflux, Fluxxer, McFly, there are too many to name.
- Redux is one of these Flux implementations.
- Now, due to its simplicity and ease of use, it has emerged as one of the winners in the Flux library shakedown.
- Redux can help you build applications that are easier to understand.
- Redux makes it easier to collaborate with other developers about the codebase and the dataflow in an application. It should also make it easier for you to find and fix bugs.

- History Of Redux
  - Reflux
  - Flummox
  - Fluxxor
  - Alt
  - Redux
  - Marty.js
  - McFly
  - DeLorean
  - Lux
  - Fluxy
  - Material Flux

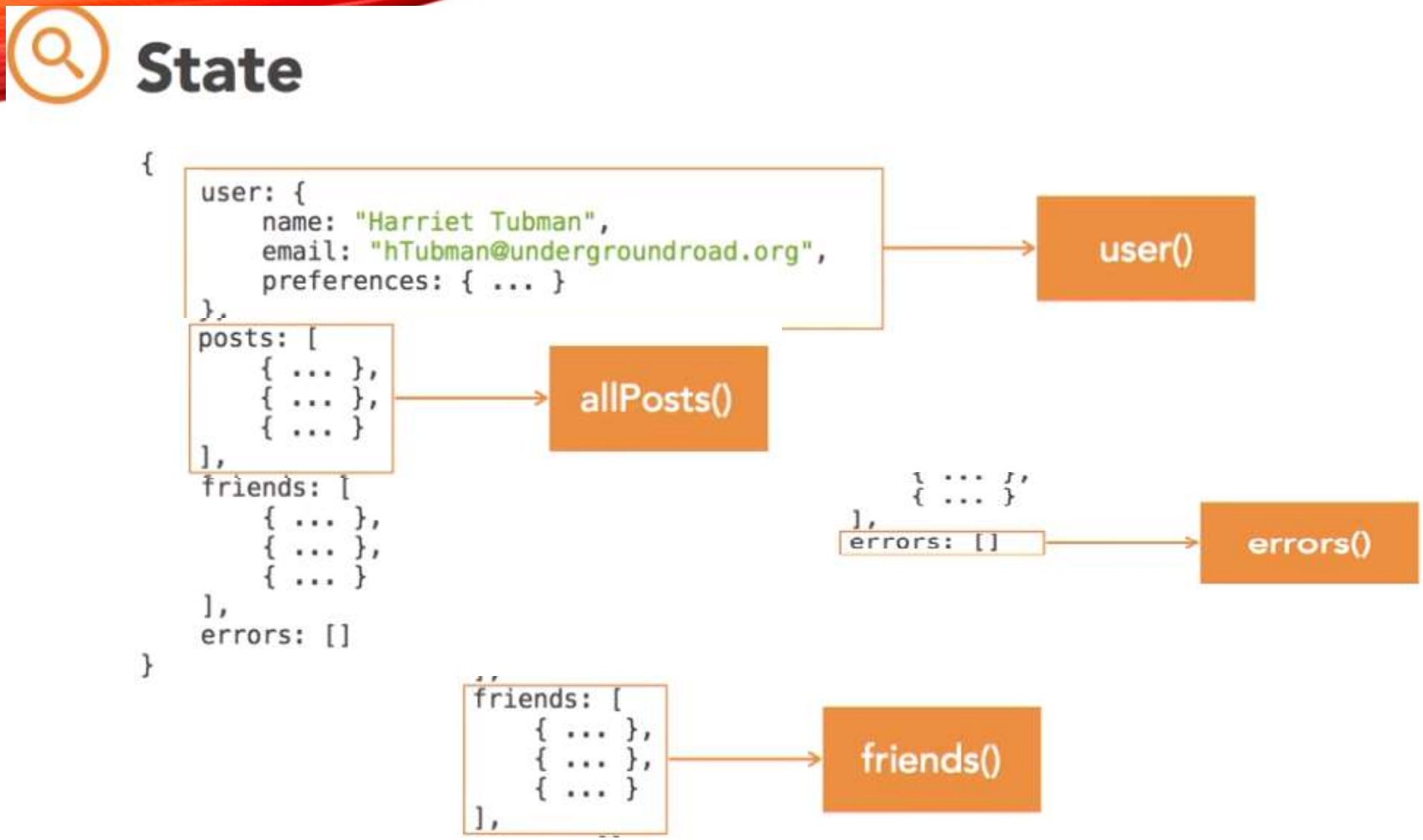
- How Does Redux Work?
- Redux isn't exactly Flux, it's Flux-like, and Flux data flows in one direction.
- An action is sent to the dispatcher, and then dispatched to one or more stores before the view is updated.
- In Redux data still flows in one direction, but there's a big difference.
- There's only one store.
- With Redux you cannot use multiple stores, and because there is only one store, there's no need for a dispatcher.
- The store will dispatch the actions directly.
- Having one stores means that all of your state will be located in one place.

- History Of Redux





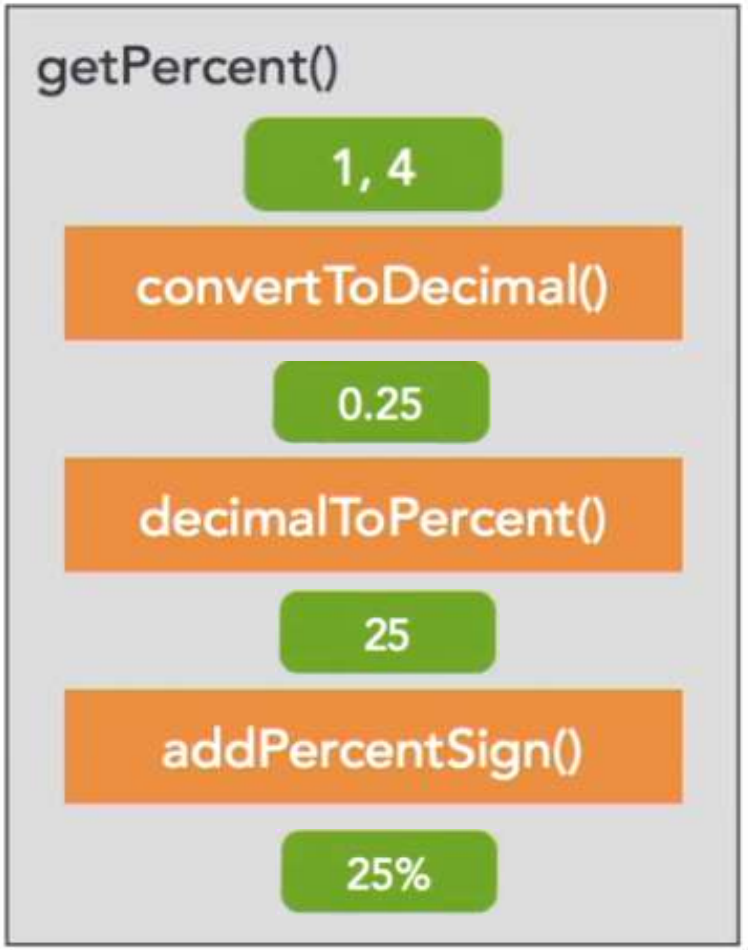
- History Of Redux



- History Of Redux
- The idea of using functions for modularity comes from the functional programming paradigm.
- It's the paradigm that was used to develop Redux.
- We don't have to be an expert in functional programming to work with Redux, but there are a few important concepts that you want to keep in mind.
- The first are **pure functions**, Pure functions are functions that do not cause side effects.
- Everything the function needs to operate is sent as arguments, and a new result is returned.
- Pure functions do not change any of their arguments or any global variables.
- They simply use the information to produce a new result.

- History Of Redux
- **Immutability**. To mutate is to change, so to be immutable is to be unchangeable.
- We do not want to change the variables and objects in a functional application instead we want to produce new ones.
- The third and final point is composition.
- **Composition** refers to the ability to put functions together in a way that one function's output becomes the next function's input.
- This means that the values returned from one function become the arguments of the next function, until eventually the last function returns the value we were looking for.

- History Of Redux
- Composition Example



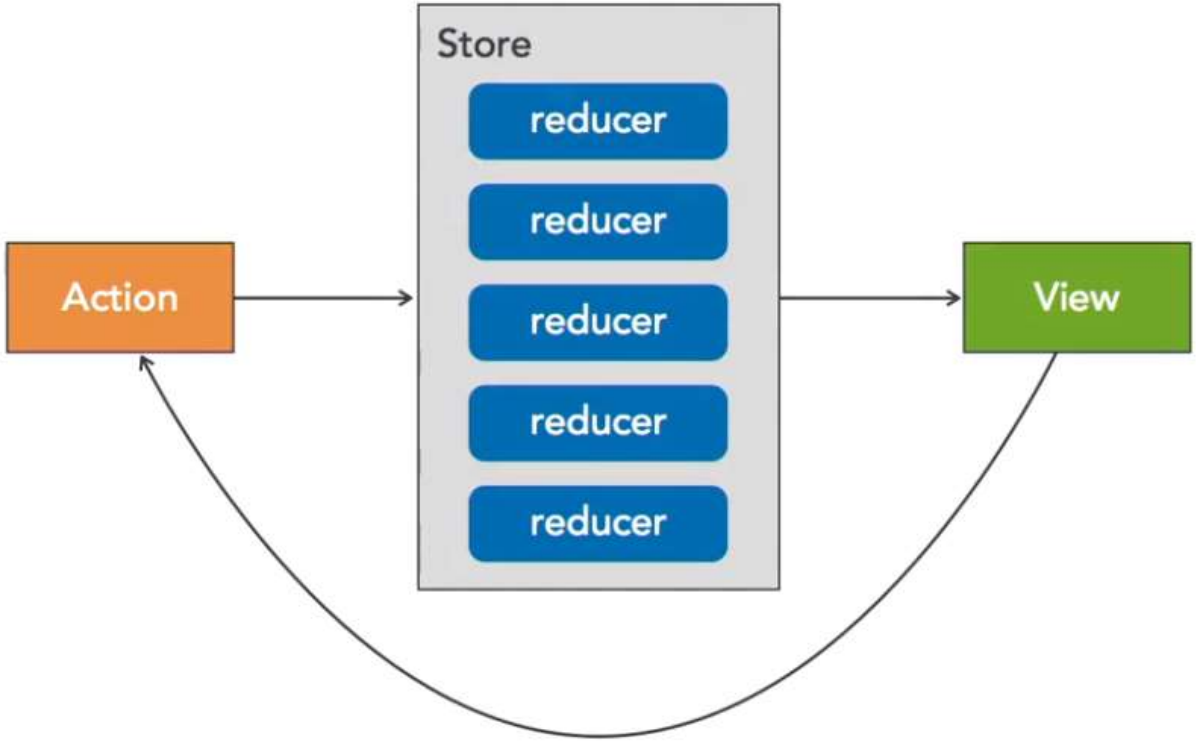
- History Of Redux

Compose

```
import { compose } from 'redux'

const getPercent = compose(
    addPercentSign
    decimalToPercent,
    convertToDecimal
)

getPercent(1, 4)

// 25%
```

- This is the idea behind composition.
- In Redux, composition is used in the store.
- The reducer functions that we create to manage specific parts of the state tree are composed, and the action and state is sent to and piped through each of these reducers until a state is eventually mutated.
- Now we don't have to worry about how these reducers are composed.
- All we have to do is identify state, write good reducers, and let the Redux store handle the rest.

- History Of Redux



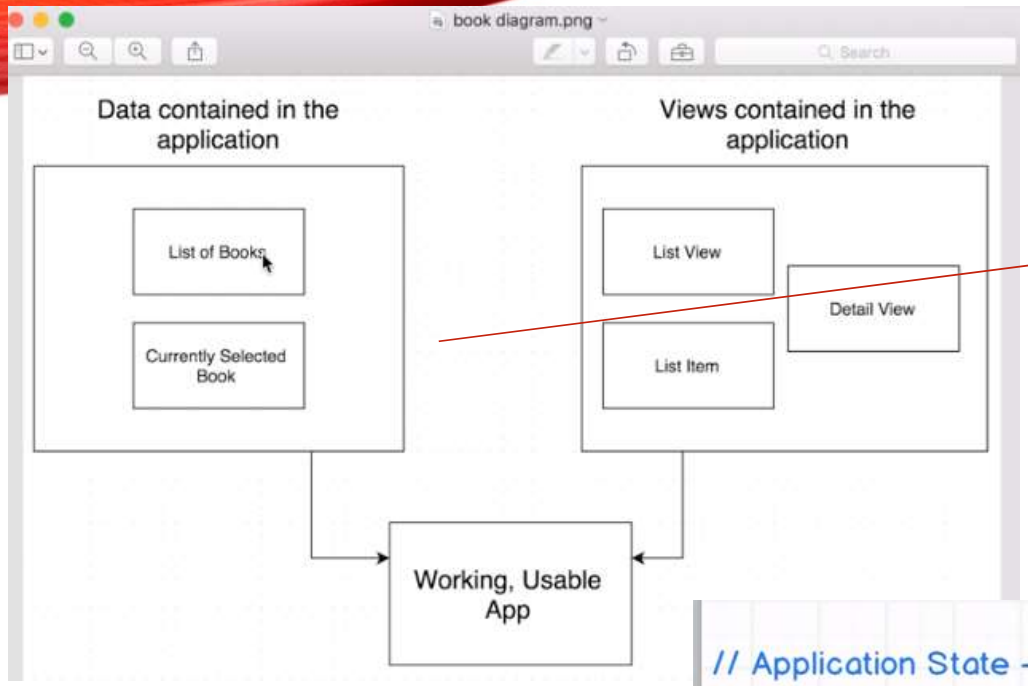
- Redux Official Website

- What is Redux?

- Redux is DATA contained in the application
- React is the Views contained in the application.
- The main difference between redux and other patterns is that in Redux all of the applications data is centralized into a single object referred to as application level state

- A REDUCER is a function that returns a piece of the application state. An application can have many pieces of state we can have many reducers

- Reducres



Here there are two application states, hence there will be two reducers.
Reducers produce the values of the application state.

```
// Application State - Generated by reducers
{
  books: [ { title: 'Harry Potter' }, { title: 'Javascript" } ],

  activeBook: { title: 'Javascript: The Good Parts" }
}
```

Books Reducer

ActiveBook Reducer

- Redux Api


Redux API
createStore(reducer, initialState)
getState()
dispatch()
subscribe()

- Redux API
- n the world of React, Redux is a popular and quality state container.
- it provides a good basis for implementing the MVI architecture.
- Redux usage can be very simple, but it can also scale to sophisticated scenarios.
- CreateStore is the function used to create a new store.
- To create a store, the programmer supplies a reducer function and the initial store state.
- Get state returns the current application state from within the store.
- Dispatch sends an action to the store to be applied to the current state.
- The action is processed by the reducer function which builds a new application state.
- Subscribe registers a call back to be called when the application state held within the store changes.

- Redux – Redux
- React-redux is an extra module that helps with the integration of React and Redux.
- For sophisticated React and Redux users, it helps to make code neater and add some useful features.
- The main service provided by React-redux is to connect React components to the application state.
- React-redux can provide data from the Redux store to components when the component is rendered, and it can provide a way for components to publish actions that can then be used to modify the Redux store.
- Provider is a React component provided by React-redux.
- When it is included in a React application, it enables all React components below it in the component tree, that is its children or children's children, et cetera to connect to the Redux store.
- Connect is a function provided by React-redux that enhances React components connecting them to the Redux store in the ways specified.
- To specify what data from the Redux store should be provided to the React component as props, connect expects a parameter called mapStateToProps.

- Redux – Redux
- mapStateToProps is a function from the Redux store to a set of props for the component.
- mapStateToProps takes care of getting data from the store to the component.
- Another parameter of the connect function, mapDispatchToProps, takes care of specifying how the component can send actions to the Redux store.
- mapDispatchtoProps is a function from Redux's dispatch function to a set of props for the component.
- In practice, this provides a place to map component events to Redux store actions.

React-redux

Provider

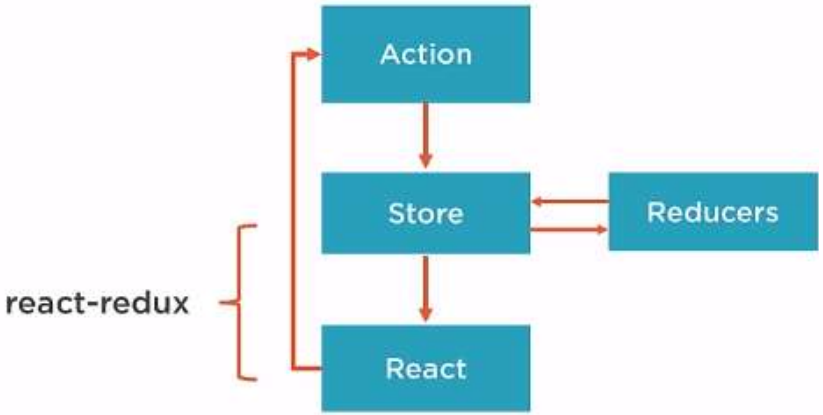
connect

- mapStateToProps
- mapDispatchToProps

Two Component Types

Container	Presentational
Focus on how things work	Focus on how things look
Aware of Redux	Unaware of Redux
Subscribe to Redux State	Read data from props
Dispatch Redux actions	Invoke callbacks on props
Generated by react-redux	Written by hand

React-Redux Provider




```
<Provider store={this.props.store}>
   <App/>
</Provider>
```

Connect

- **Wraps our component so it's connected to the Redux store.**

```
export default connect(
  mapStateToProps,
  mapDispatchToProps
)(AuthorPage);
```

Flux

```
componentWillMount() {
    AuthorStore.addChangeListener(this.onChange);
}

componentWillUnmount() {
    AuthorStore.removeChangeListener(this.onChange);
}

onChange() {
    this.setState({ authors: AuthorStore.getAll() });
}
```

Redux

```
function mapStateToProps(state, ownProps) {
  return {appState: state.authorReducer };
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(AuthorPage);
```

Benefits:

1. No manual unsubscribe
2. No lifecycle methods required
3. Declare what subset of state you want
4. Enhanced performance for free

React-Redux Connect

```
connect(mapStateToProps, mapDispatchToProps)
```

What state should I expose as props?

```
function mapStateToProps(state) {
    return {
        appState: state
    };
}
```

React-Redux Connect

```
connect(mapStateToProps, mapDispatchToProps)



function mapDispatchToProps(dispatch) {
     return {
          actions: bindActionCreators(actions, dispatch)
     };
}
```

Redux function

3 Ways to Handle mapDispatchToProps

```
this.props.dispatch(loadCourses());
```

Ignore it. Use dispatch.

```
function mapDispatchToProps(dispatch) {
  return {
    loadCourses: () => {
      dispatch(loadCourses());
    }
  };
}
```

Manually wrap

```
function mapDispatchToProps(dispatch) {
  return {
    actions:
      bindActionCreators(actions, dispatch)
  };
}
```

Use bindActionCreators

Option 1: Use Dispatch Directly

```
// In component...
this.props.dispatch(loadCourses())
```

Two downsides

1. **Boilerplate**
2. **Redux concerns in child components**

- The first option is to ignore it since mapDispatchToProps is an optional parameter on the connect function.
- When you omit it, then the dispatch function will be attached to your container component.
- This means you can call dispatch manually and pass it an action creator.
- Calling connect on your component automatically adds a dispatch prop to your component. You can use this dispatch prop to call your action creators.
- However, there's a couple of downsides with this approach.
- First, it requires more boilerplate each time you want to fire off an action because you have to explicitly call dispatch and pass it the action you'd like to fire.

- Second, this means your child components need to reference Redux-specific concepts, like the dispatch function, as well as your action creators.
- If you want to keep your child components as simple as possible and avoid tying them to Redux, then this approach is not ideal.

Option 2: Wrap Manually

```
function mapDispatchToProps(dispatch) {
  return {
    loadCourses: () => {
      dispatch(loadCourses());
    },
    createCourse: (course) => {
      dispatch(createCourse(course));
    },
    updateCourse: (course) => {
      dispatch(updateCourse(course));
    }
  };
}

// In component...
this.props.loadCourses()
```

Specifying the actions we want to expose to my component explicitly here.
One by one, we wrap each action creator in a dispatch call, and then this is how the call would look within the component.
When you're getting started, it is recommend using this option because manually wrapping action creators makes it clear what you're doing.
But as we can see, it's quite redundant.
That's why you may prefer to use option number 3, which is bindActionCreators.

Option 3: bindActionCreators

```
function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(actions, dispatch)
  };
}
// In component:
this.props.actions.loadCourses();
```

Wraps action creators in dispatch call for you!

- This function ships with Redux to handle this redundancy for you. With this approach, the bindActionCreators function will wrap all the actions passed to it in a dispatch call for you.
- Of course, the props created by these two examples is slightly different.
- Notice that the prop that will be exposed to the component here is called actions.
- But if we go back to the previous slide, we are exposing this. props. loadCourses, this. props. createCourse, and so on.
- So it's a minor difference in the way that I chose to wire this up.
- The bottom line is approach 2 and 3 both produce the same result.
- They wrap your actions in a dispatch call so that they're easy to pass down to child components

- . But there's a notable advantage to option 2 and 3 over option 1, and that is that with options 2 and 3 your child components don't have to know anything about Redux.
- Child components can simply call the actions that are passed down to them via props.
- Remember, with option 1, we had to import action creators into our child components so that we could call Redux's dispatch directly.

A Chat With Redux

React	Hey CourseAction, someone clicked this "Save Course" button.
Action	Thanks React! I will dispatch an action so reducers that care can update state.
Reducer	Ah, thanks action. I see you passed me the current state and the action to perform. I'll make a new copy of the state and return it.
Store	Thanks for updating the state reducer. I'll make sure that all connected components are aware.
React-Redux	Woah, thanks for the new data Mr. Store. I'll now intelligently determine if I should tell React about this change so that it only has to bother with updating the UI when necessary.
React	Ooo! Shiny new data has been passed down via props from the store! I'll update the UI to reflect this!

- Async Redux

```
export function deleteAuthor(authorId) {
  return dispatch => {
    return AuthorApi.deleteAuthor(authorId).then(() => {
      dispatch(deletedAuthor(authorId));
    }).catch(handleError);
  };
}
```

Thunk

- Redux-thunk
- Prev slide has an example of a thunk for deleting an author.
- As you can see, a thunk is a function that returns a function.
- Thunk is actually a computer science term.
- A thunk is a function that wraps an expression in order to delay its evaluation.
- So in this case, the deleteAuthor function is wrapping the dispatch function so that dispatch can run later.
- Depending on your programming background, returning functions from functions may feel strange, but it's a common and powerful technique in functional programming.
- On the third line, we are calling a regular action creator called deletedAuthor.
- But note that you don't have to call a separate action creator function.
- You can simply inline the action within the thunk if you prefer.
- This action creator's only going to be used in this one spot, and that is often the case when you're working with thunks.

```
https://unpkg.com/react@16.0.0/umd/react.development.js
```

```
https://unpkg.com/react-dom@16.0.0/umd/react-dom.development.js
```

React vs. Angular

React	Angular
Renders UI and handles events	A complete UI framework
Uses JavaScript for view logic	Custom “template expression” syntax
JavaScript	TypeScript