Getting started with **ReactJS**

with Daniel Dyrnes,
VP of Engineering @ Ulven Tech

Scope of the course

1. Introduction to ReactJS
2. ReactJS concepts
    a. ES6
    b. JSX
    c. PropTypes
3. Components and structure
4. Routing
5. State Management
    a. React state (local state)
    b. Redux
6. Useful libraries
7. Live programing
8. Project Day

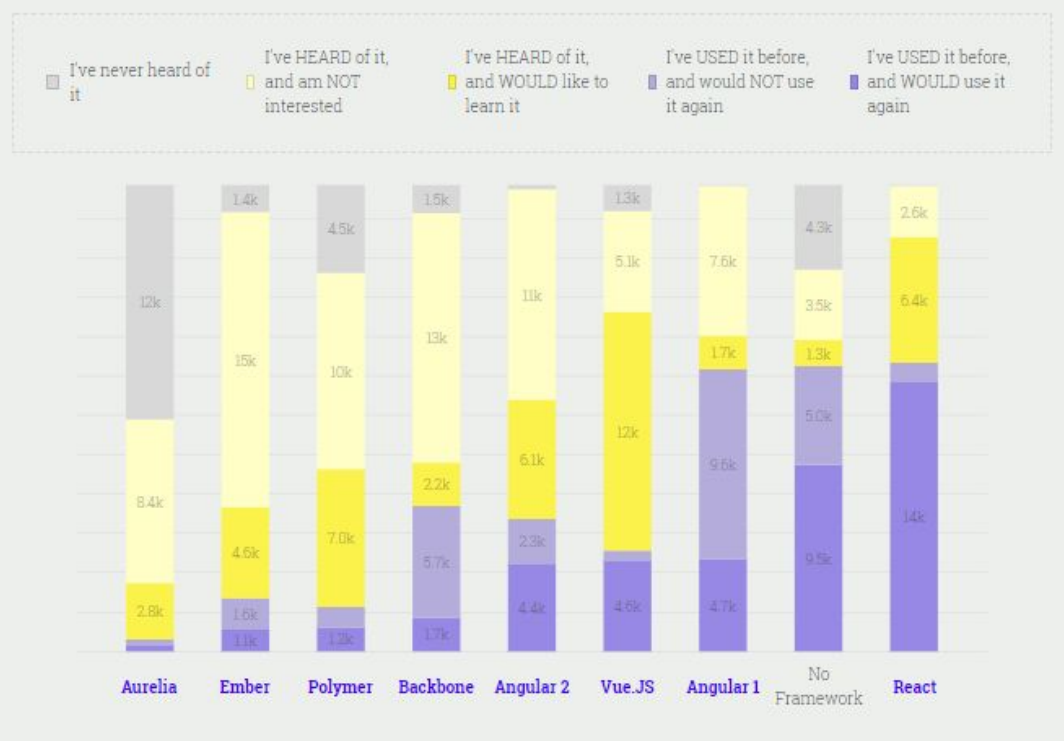
Schedule

Day 1: (1, 2)
Day 2: (3, 4)
Day 3: (5, 6)
Day 4: (7)
Day 5: (8)

So what is ReactJS?

- Library for developing modern SPA's in JavaScript.
- What is SPA?
- Problems with SPA's.
- Why use SPAs.
- Has a cousin called React Native for building cross platform IOS/Android apps.

Is **ReactJS** popular?



Who is using **ReactJS**?

1. Facebook (also the creators of ReactJS)
2. Instagram
3. Netflix
4. New York Times
5. Yahoo! Mail
6. Khan Academy
7. WhatsApp (web)
8. Vivaldi Browser
9. Codecademy
10. Dropbox

Big list of sites using react: https://github.com/facebook/react/wiki/Sites-Using-React

ECMAScript 6 (ES6)

ES6 is a significant update to the JavaScript language, and the first update to the language since ES5 was standardized in 2009. Implementation of these features in major JavaScript engines is underway now.

ES6 can already be used with a module bundler like Web Pack, this also happens to be the recommended way of using react.

Web Pack will cross-compile the ES6 in to widely supported ES5 and bundle the different npm modules in to 1 big js file that can be imported.

We will be using a cli tool called “create-react-app” by Facebook to configure Web Pack and setting up our boilerplate.

Recommended reading: https://github.com/lukehoban/es6features

ES6: Arrow functions

```
// ES5
function greet(name) {
  return 'Hello ' + name;
}

// ES6
const sayHello = (name) => {
  return `Hello ${name}`;
}

// ES6 Shorthand
const sayHi = name => `Hello ${name}`;
```

ES6: export/import

```
export const FISH = 'マグロ';

// from path needs to be the path
// of the file you are exporting from.
import { FISH } from './index';

console.log(FISH); // Output: マグロ
```

ES6: import/export - default

```
// You can export most things.
export default {
  FISH: 'マグロ',
  sayHi: (name) => `Hi ${name}`,
};

// Notice how we are not using
// brackets this time.
import Stuff from './index';

console.log(Stuff.FISH); // Output: マグロ
console.log(Stuff.sayHi('kim')); // Output: Hi kim
```

What is JSX?

```
const element = <h1>Hello, world!</h1>;
```

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript. JSX produces React “elements”.

- Facebook React Documentation

What is JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called “components” that contain both.

React doesn’t require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of **if** statements and **for** loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes.
Notice how we are using **camelCase**

```
const element = <div tabIndex="0"></div>;
```

You may also use curly braces to embed a JavaScript expression in an attribute

```
const element = <img src={user.avatarUrl}></img>;
```

Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Specifying Children with JSX

If a tag is empty, you may close it immediately with />, like XML

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children.
Note: Every class can only return one child.

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX Prevents Injection Attacks

It is safe to embed user input in JSX.

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

Rendering an Element into the DOM

You attach the React DOM renderer to a divs id. So you will need to have a div with a id, usually “root” in your html.

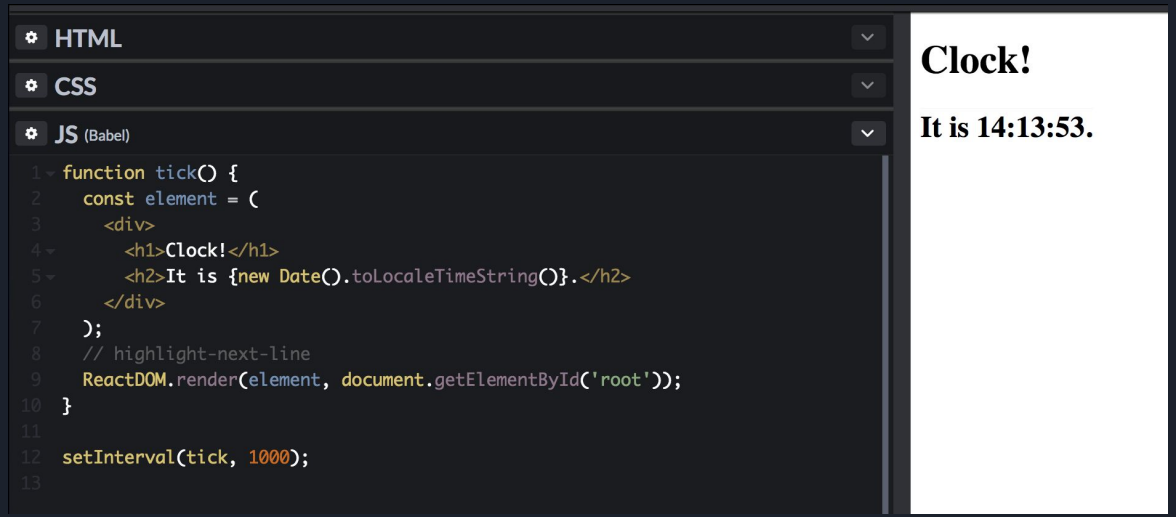
```
1  const element = <h1>Hello, world</h1>;
2
3  ReactDOM.render(element, document.getElementById('root'));
4
```

Hello, world

Updating the Rendered Element

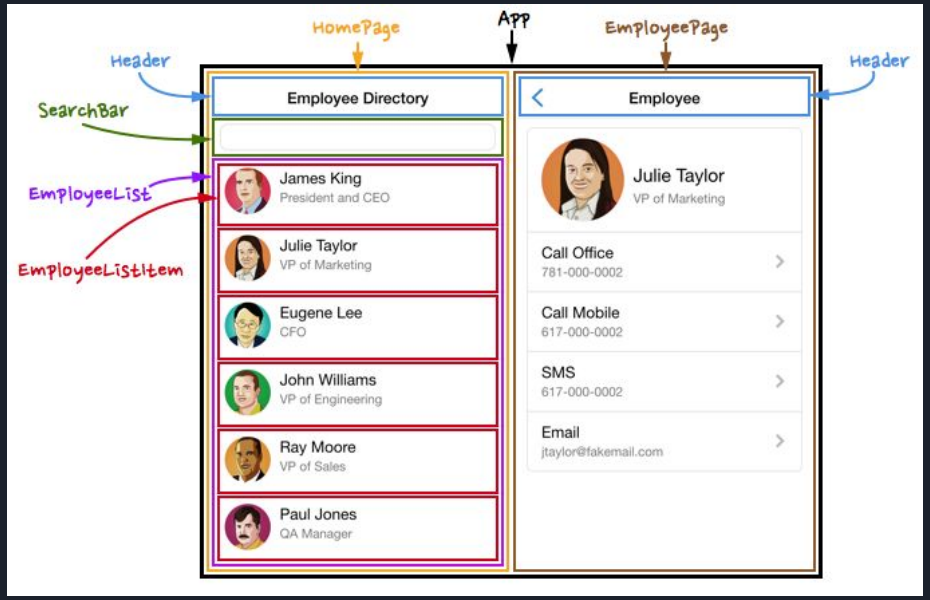
React elements are immutable. Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

In practice, most React apps only call ReactDOM.render() once.



What is a **Component**?

A component in ReactJS is equivalent to a building-block, throughout your project you will be making lots of components that you will assemble and reuse to form your final product.



Why should I use components?

- Components are highly reusable.
- If build right they are stateless and easily unit-testable.
- They have life cycle hooks.
- They encourage you to break your project into smaller pieces of code, hence reducing the likelihood of making a giant wall of complex code.

What does a **component** look like?

index.jsx

```jsx
import React, { Component } from 'react';  8.1K (gzipped: 3.3K)

class Lead extends Component {
  render() {
    return (
      <div>
        <p>Hello World</p>
      </div>
    );
  }
}

export default Lead;
```

Notice how we are using JSX

Components naming convention

Always start component names with a capital letter!

React treats components starting with lowercase letters as DOM tags. For example, represents an HTML div tag, but <Welcome /> represents a component and requires Welcome to be in scope.

Reusable components

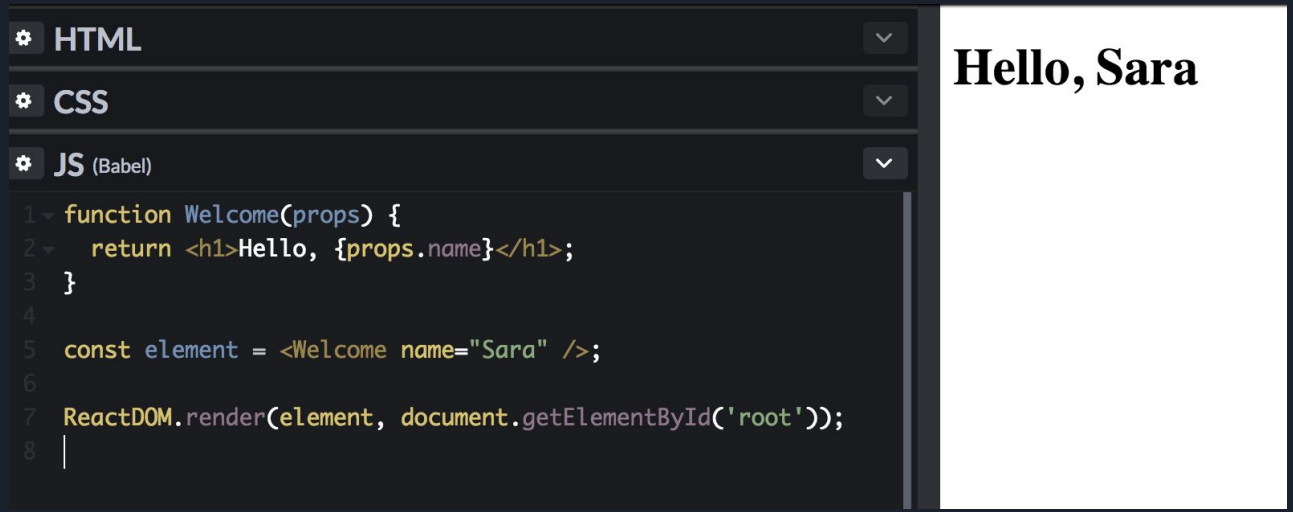
Components can be easily reused by importing them using ES6 “import” statement.
Say we want to import the component we made on the last page, to use it in our application.
We called it **Lead** so we would need to reference the pass then we can render it like any other JSX element.

```
import React, { Component } from 'react';
import Lead from './Lead.jsx';

You, a few seconds ago | 1 author (You)
class Leads extends Component {
  render() {
    return (
      // Notice how we need to wrap with a div
      // because we can only return one element from
      // every render function.
      <div>
        <Lead />
        <Lead />
      </div>
    );
  }
}
```

Passing data to components

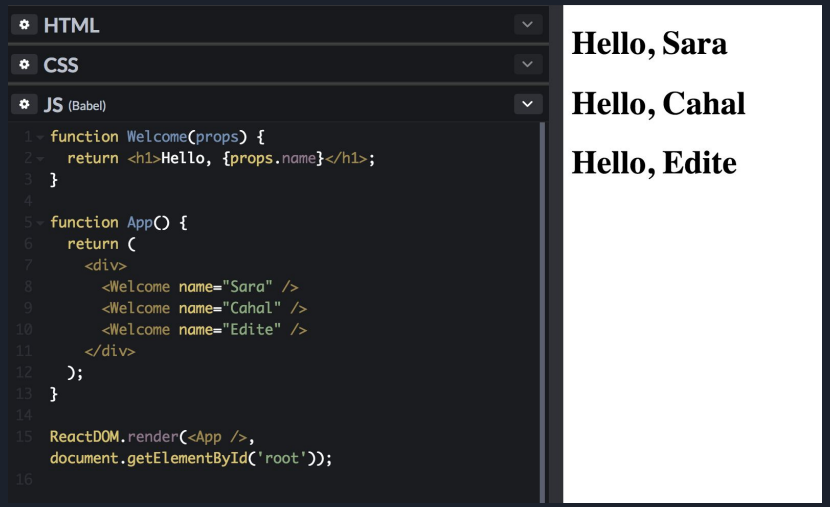
We pass data using **props**. Props are similar HTML attributes but you define them.
In the example below we are passing a prop of **text** to the component then the component will dynamically change the button text to reflect the prop.



Composing Components

Components can refer to other components in their output. This lets us use the same component abstraction for any level of detail. A button, a form, a dialog, a screen: in React apps, all those are commonly expressed as components.

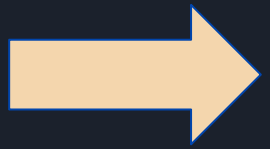
For example, we can create an App component that renders Welcome many times.



Extracting Components

Don’t be afraid to split components into smaller components.
For example, consider this Comment component.

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}
        />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```



```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}
    />
  );
}
```

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

Props are Read-Only

Whether you declare a component as a function or a class, it must never modify its own props. Consider this sum function.

```
function sum(a, b) {
  return a + b;
}
```

Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.
In contrast, this function is impure because it changes its own input.

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

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

PropTypes

PropTypes is how we define required and optional parameters for React components.

You can use PropTypes to specify type such as.

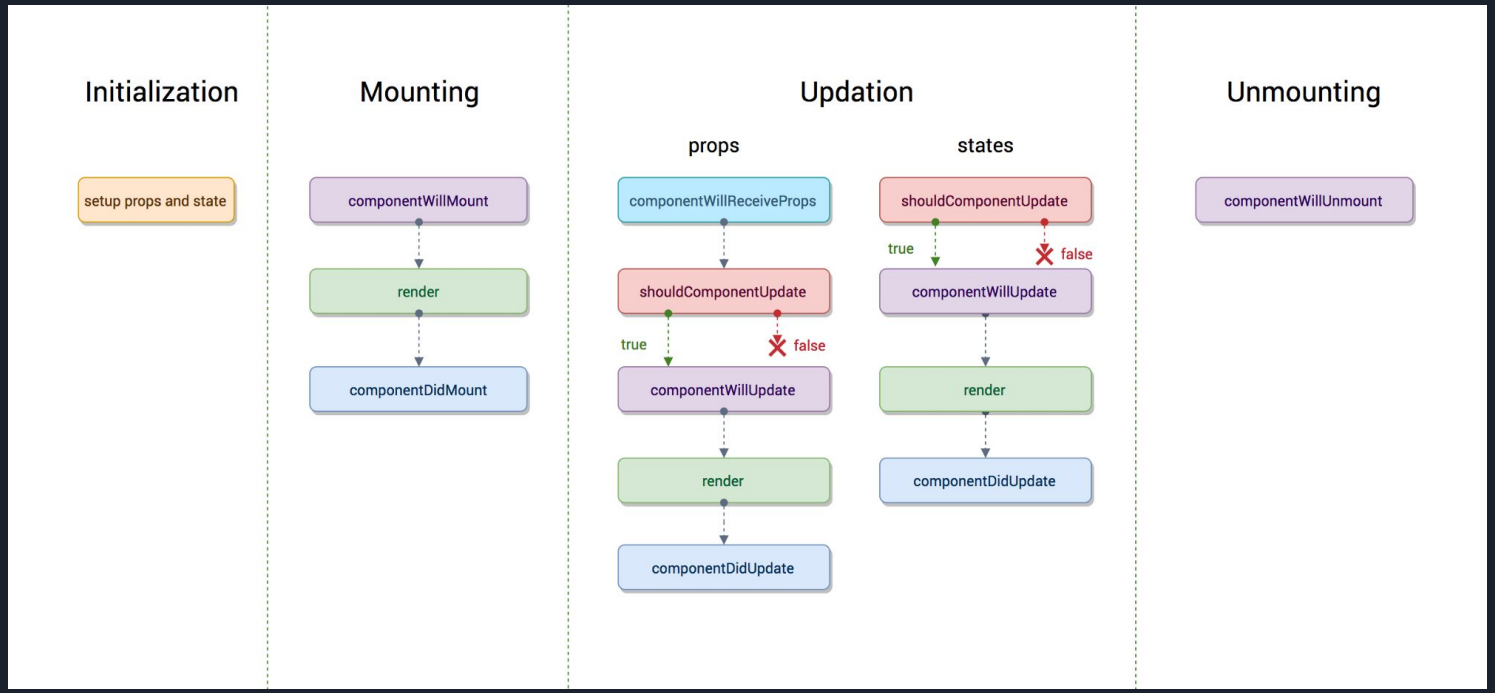
- String
- Number
- Shape (Object)
- Array
- Bool
- Node

PropTypes also support setting default values if no value is passed to a optional prop, this will help you to write more stabile code and catch more errors.

How to use PropTypes

```
You, a few seconds ago | 1 author (You)
1  import React, { Component } from 'react';  8.1K (gzipped: 3.3K)
2  import PropTypes from 'prop-types';  1.6K (gzipped: 838)
3
   You, a few seconds ago | 1 author (You)
4  class Container extends Component {
5    static propTypes = {
6      children: PropTypes.node.isRequired,
7    }
8
9    render() {
10     return (
11       <div className="container-fluid flex-grow-1 container-p-y">
12         {this.props.children}
13       </div>
14     );
15   }
16 }
17
18 export default Container;
19
```

Component lifecycle hooks



Higher-order component (HOC)

```
import React, { Component, Fragment } from 'react';  8.6K (gzipped: 3.4K)
import { withRouter } from 'react-router-dom';  10.4K (gzipped: 4.1K)
import { ThemeProvider } from 'styled-components';  43.4K (gzipped: 16.2K)
import Navbar from '../../components/Navbar';
import Footer from '../../components/Footer';
import './bootstrap.css';
import './custom.css';

export const THEME = {};

You, a few seconds ago | 1 author (You)
class App extends Component {
  render() {
    return (
      <ThemeProvider theme={THEME}>
        <Fragment>
          <Navbar />
          {this.props.children}
          <Footer />
        </Fragment>
      </ThemeProvider>
    );
  }
}

export default withRouter(App);
```

Error Boundaries

A JavaScript error in a part of the UI shouldn’t break the whole app. To solve this problem for React users, React 16 introduces a new concept of an “error boundary”.

In the past, JavaScript errors inside components used to corrupt React’s internal state and cause it to emit cryptic errors on next renders. These errors were always caused by an earlier error in the application code, but React did not provide a way to handle them gracefully in components, and could not recover from them.

Note

Error boundaries do **not** catch errors for:

- Event handlers (learn more)
- Asynchronous code (e.g. `setTimeout` or `requestAnimationFrame` callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }

  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

How to use the boundary

```
<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

State management in **ReactJS**

What is state and why do we need it?

We use state to keep track of things like if a toggle switch is toggled or not, or text field input.
Historically state was kept by declaring a var in JS and assigning state to it.
This approach quickly becomes unmanageable when you have a lot of state to keep track of and no clear scope for every the states.

React's solution to this is to implement a optional "local state" for every component, scoped within that component.

Note: State in react is **immutable**.

Adding Local State to a Class

Local state is local to the component it's defined within.
See the example below for what localstate looks like.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Changing a local state

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Passing state with props

Managing state with Redux

https://github.com/ulventech/react-101

https://github.com/ulventech/react-101-tasks

https://goo.gl/e7giFX <- Slides

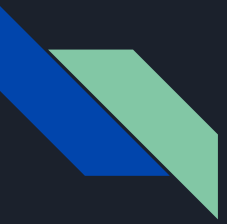
Useful libraries

Axios (Promise based HTTP client for the browser and node.js)

POST: https://us-central1-react-training-101.cloudfunctions.net/api/{uid}/item
GET: https://us-central1-react-training-101.cloudfunctions.net/api/{uid}/item/{id}
PUT: https://us-central1-react-training-101.cloudfunctions.net/api/{uid}/item/{id}
DELETE: https://us-central1-react-training-101.cloudfunctions.net/api/{uid}/item/{id}
GET: https://us-central1-react-training-101.cloudfunctions.net/api/{uid}/items

Redux-saga (An alternative side effect model for Redux apps)

Async state management in Redux with Redux-Saga



Lodash (A modern JavaScript utility library delivering modularity, performance, & extras.)

Project time!

What will be be building?

We will create a simple task application using ReactJS.

We will use local state to manage the task state and we will use reactstrap and styled-components to do some basic layout and styling.

Questions?