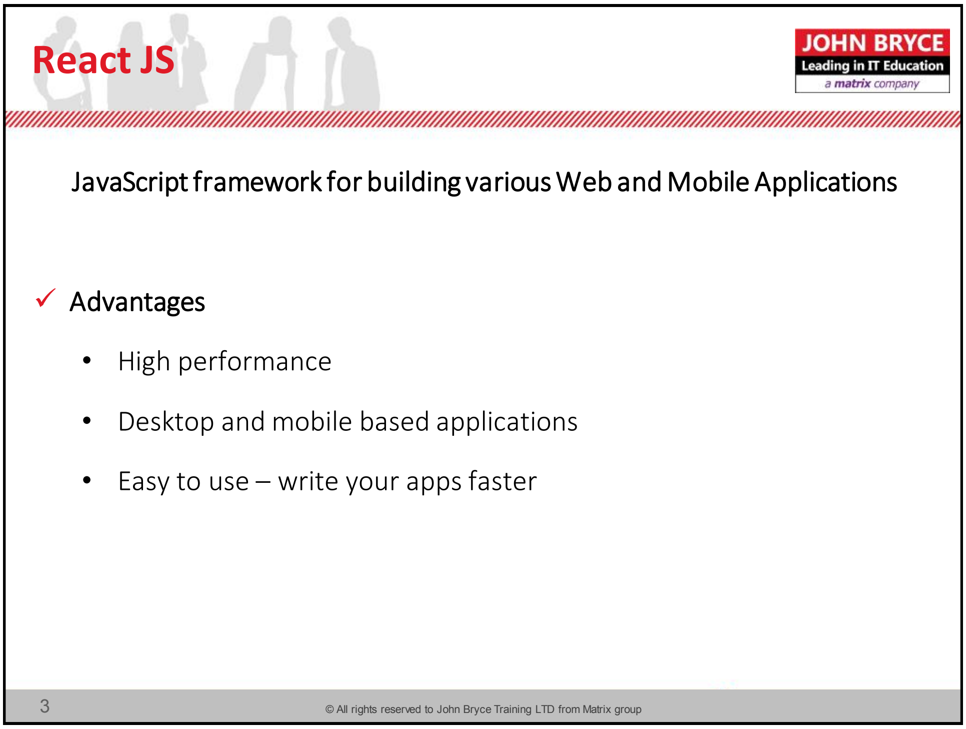
REACT.JS

JOHN BRYCE
Leading in IT Education
a matrix company

JOHN BRYCE
Leading in IT Education
a matrix company

INTRO TO REACT

A VERY HIGH LEVEL ONE

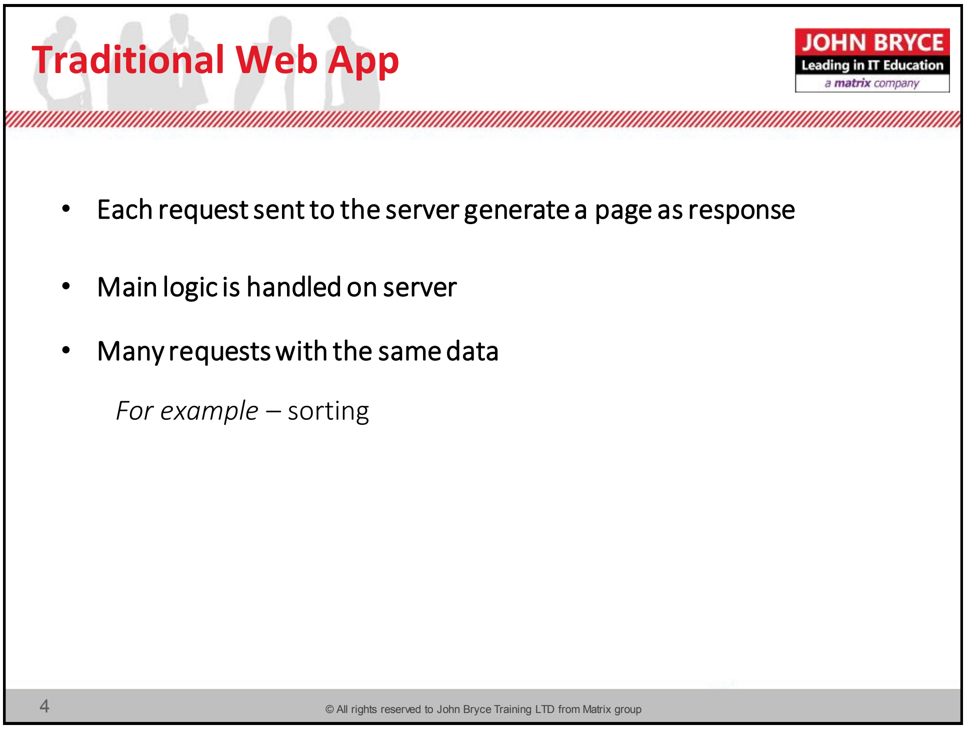


React JS

JavaScript framework for building various Web and Mobile Applications

✓ Advantages

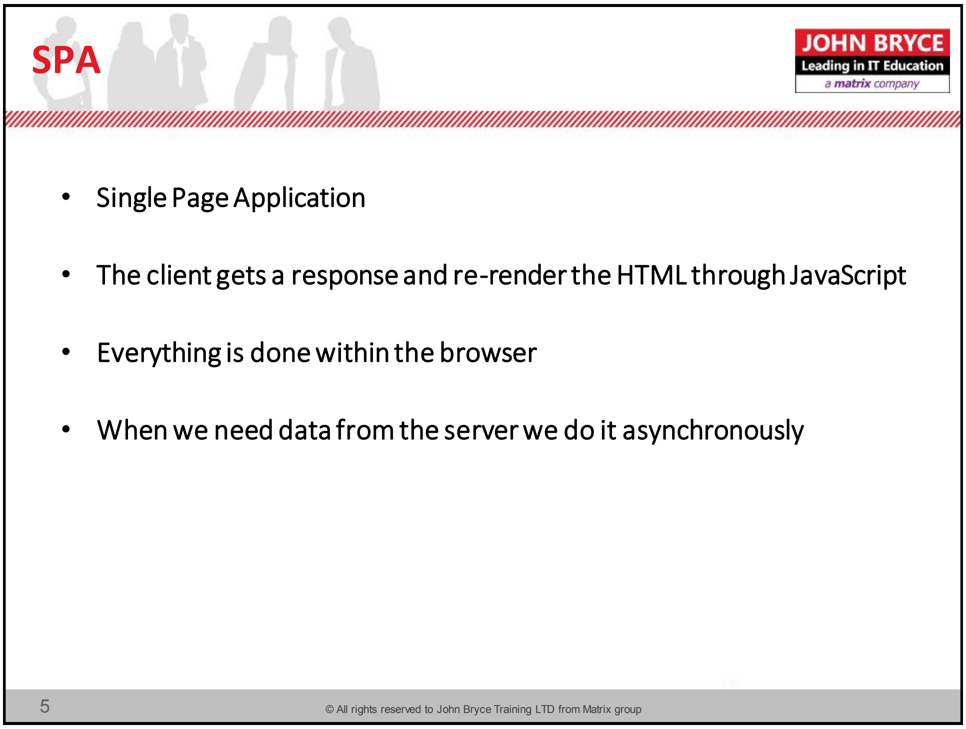
- High performance
- Desktop and mobile based applications
- Easy to use – write your apps faster



Traditional Web App

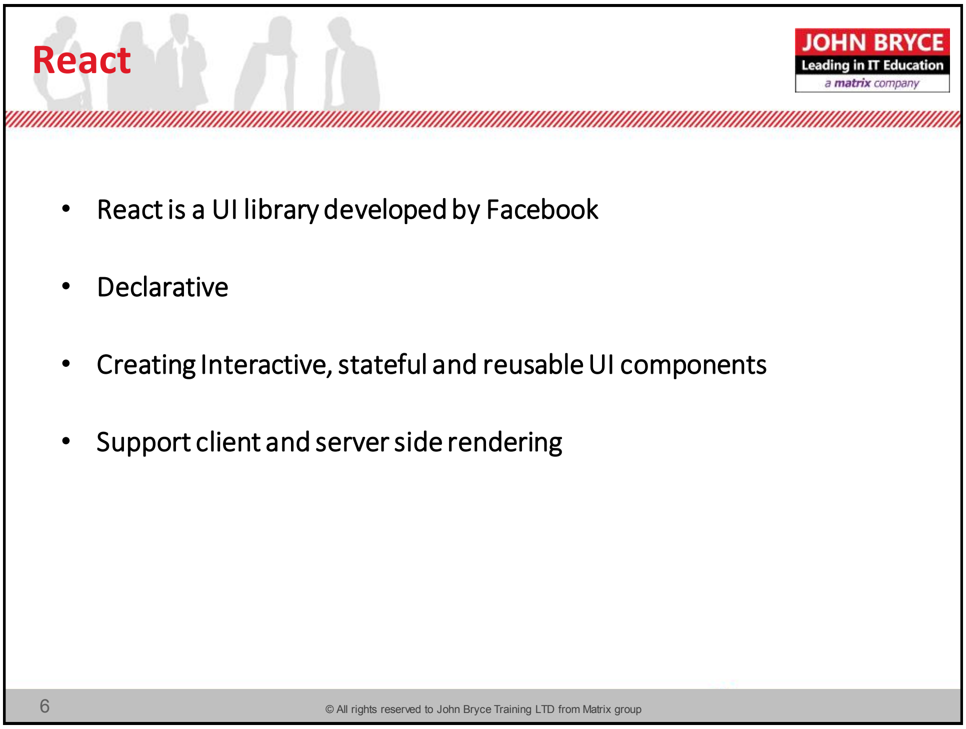
- Each request sent to the server generate a page as response
- Main logic is handled on server
- Many requests with the same data

  For example – sorting



SPA

- Single Page Application
- The client gets a response and re-render the HTML through JavaScript
- Everything is done within the browser
- When we need data from the server we do it asynchronously



React

- React is a UI library developed by Facebook
- Declarative
- Creating Interactive, stateful and reusable UI components
- Support client and server side rendering

Components in other SPA libraries

JOHN BRYCE
Leading in IT Education
a matrix company

jQuery
write less, do more.
Bootstrap
Ext JS
Kendo UI
THE ART OF WEB DEVELOPMENT
ember
wijmo
UI for the web
jQuery
user interface

Why React?

JOHN BRYCE
Leading in IT Education
a matrix company

- ✓ Fast

  Apps made in React can handle complex updates and still feel quick and responsive

- ✓ Modular

  Instead of writing large, dense files of code, you can write many smaller, reusable files

- ✓ Scalable

  Large programs that display a lot of changing data are where React performs best

- ✓ Flexible

  You can use React for interesting projects that have nothing to do with making a web app

Who is using react?

create-react-app

- Automatic tool to build development environment

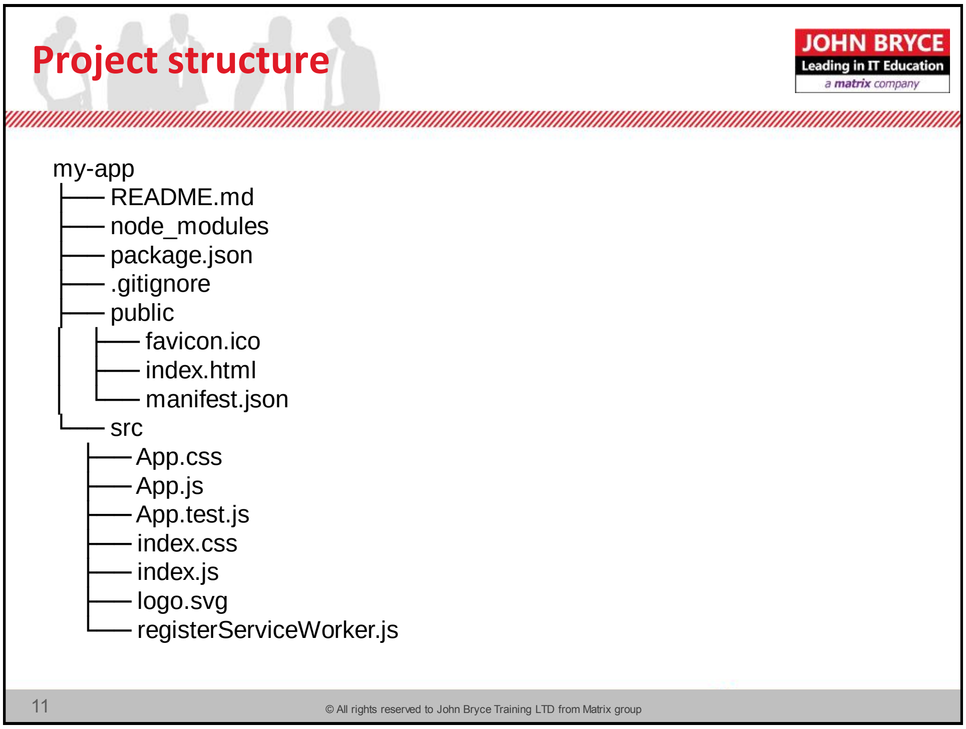
https://github.com/facebook/create-react-app

To create a new react app project:

```
# npx create-react-app my-app

# cd my-app/

# npm start
```

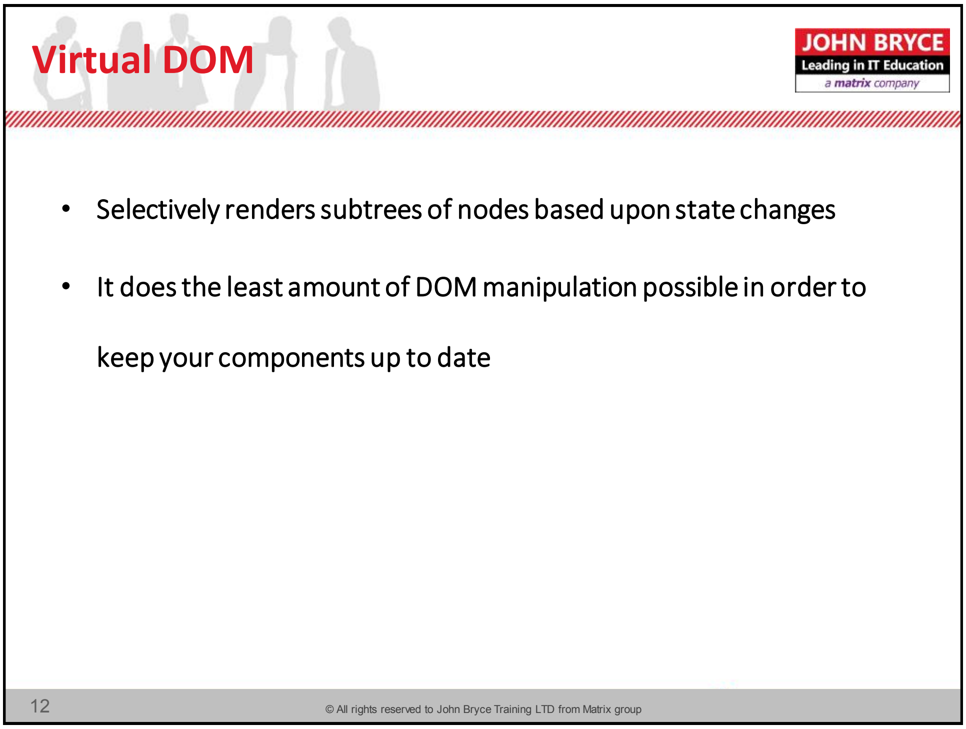
- To deploy:

```
# npm run build

# npm run eject
```



Project structure

```
my-app
├── README.md
├── node_modules
├── package.json
├── .gitignore
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
└── src
    ├── App.css
    ├── App.js
    ├── App.test.js
    ├── index.css
    ├── index.js
    ├── logo.svg
    └── registerServiceWorker.js
```



Virtual DOM

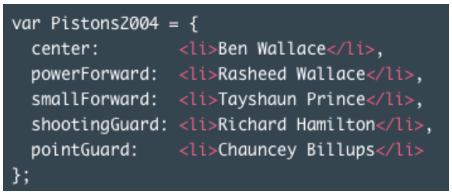
- Selectively renders subtrees of nodes based upon state changes
- It does the least amount of DOM manipulation possible in order to keep your components up to date

DOM Rendering

- ReactDOM.render makes changes by leaving the current DOM in place and simply updating the DOM elements that need to be updated.
- This smart DOM rendering is necessary for React to work in a reasonable amount of time because our application state changes a lot.
- Every time we change that state, we are going to rely on ReactDOM.render to efficiently re-render the UI.

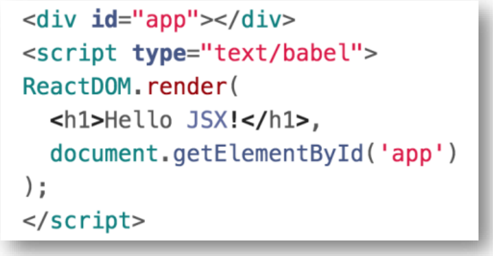
JSX

- Syntax extension for JavaScript
- It was written to be used with React
- JSX code looks a lot like HTML
  - var h1 = <h1>Hello world</h1>;
- JSX is not valid JavaScript
  - Web browsers can't read it!
  - Translation is needed



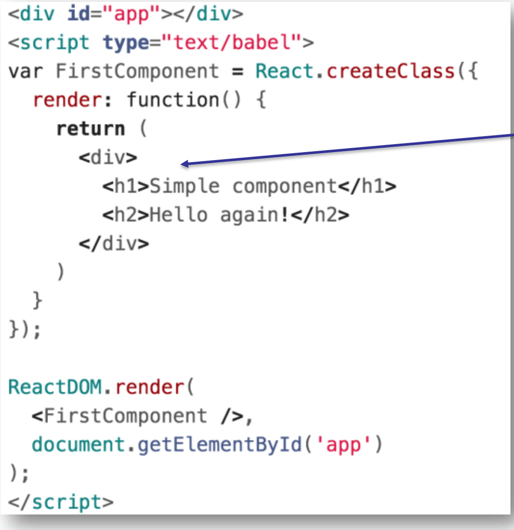
```
var Pistons2004 = {
  center:        <li>Ben Wallace</li>,
  powerForward:  <li>Rasheed Wallace</li>,
  smallForward:  <li>Tayshaun Prince</li>,
  shootingGuard: <li>Richard Hamilton</li>,
  pointGuard:    <li>Chauncey Billups</li>
};
```

Simple Example - JSX



```
<div id="app"></div>
<script type="text/babel">
ReactDOM.render(
  <h1>Hello JSX!</h1>,
  document.getElementById('app')
);
</script>
```

Simple Class – ES5

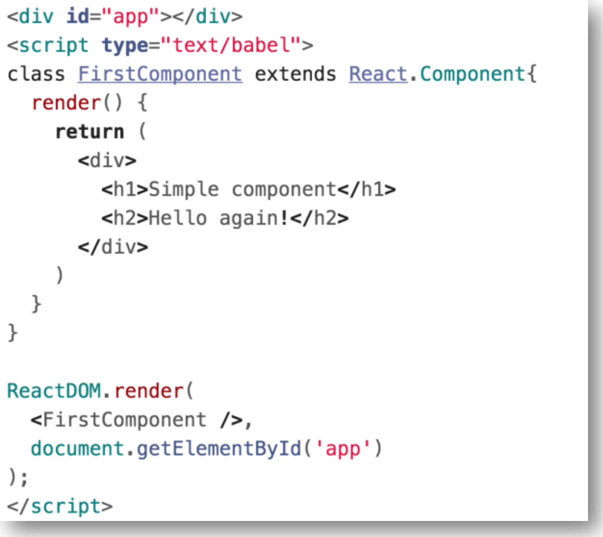


```
<div id="app"></div>
<script type="text/babel">
var FirstComponent = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple component</h1>
        <h2>Hello again!</h2>
      </div>
    )
  }
});

ReactDOM.render(
  <FirstComponent />,
  document.getElementById('app')
);
</script>
```

Note: One root element required

Simple Class – ES6

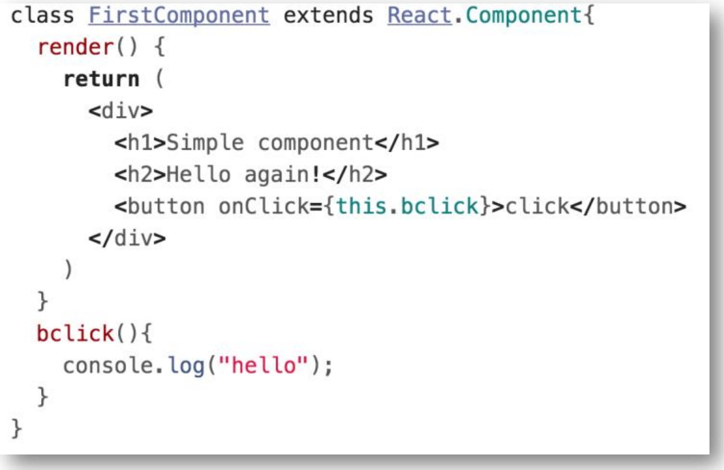
JOHN BRYCE
Leading in IT Education
a matrix company



```
<div id="app"></div>
<script type="text/babel">
class FirstComponent extends React.Component{
  render() {
    return (
      <div>
        <h1>Simple component</h1>
        <h2>Hello again!</h2>
      </div>
    )
  }
}

ReactDOM.render(
  <FirstComponent />,
  document.getElementById('app')
);
</script>
```

Events

JOHN BRYCE
Leading in IT Education
a matrix company



```
class FirstComponent extends React.Component{
  render() {
    return (
      <div>
        <h1>Simple component</h1>
        <h2>Hello again!</h2>
        <button onClick={this.bclick}>click</button>
      </div>
    )
  }
  bclick(){
    console.log("hello");
  }
}
```

React Events

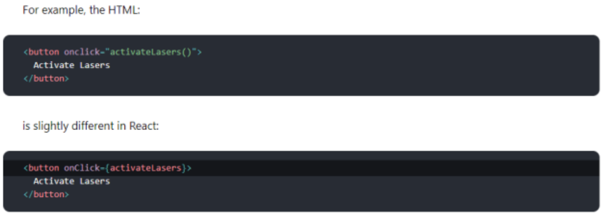
JOHN BRYCE
Leading in IT Education
a matrix company

Handling events with React elements is very similar to handling events on DOM elements.

There are some syntactic differences:

- React events are named using camelCase, rather than lowercase.

-With JSX you pass a function as the event handler, rather than a string.



For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

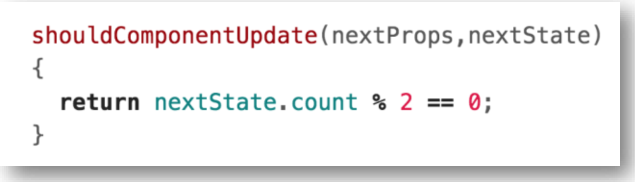
is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Updating The Component

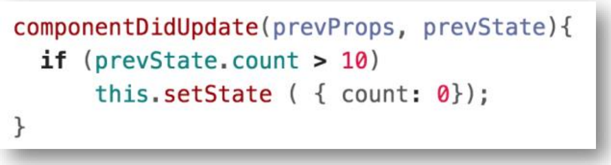
JOHN BRYCE
Leading in IT Education
a matrix company

We can decide to call the render method after state change



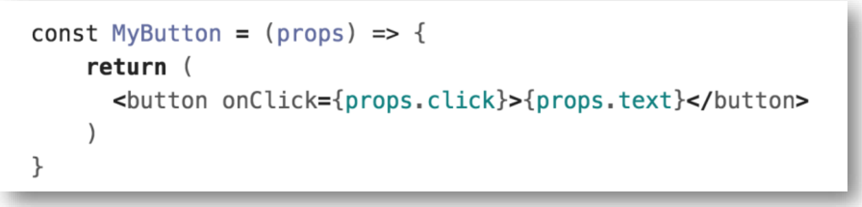
```
shouldComponentUpdate(nextProps,nextState)
{
   return nextState.count % 2 == 0;
}
```

We can check the state changes

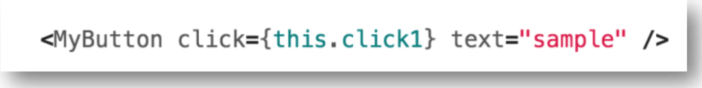


```
componentDidUpdate(prevProps, prevState){
  if (prevState.count > 10)
      this.setState ( { count: 0});
}
```

Stateless Components

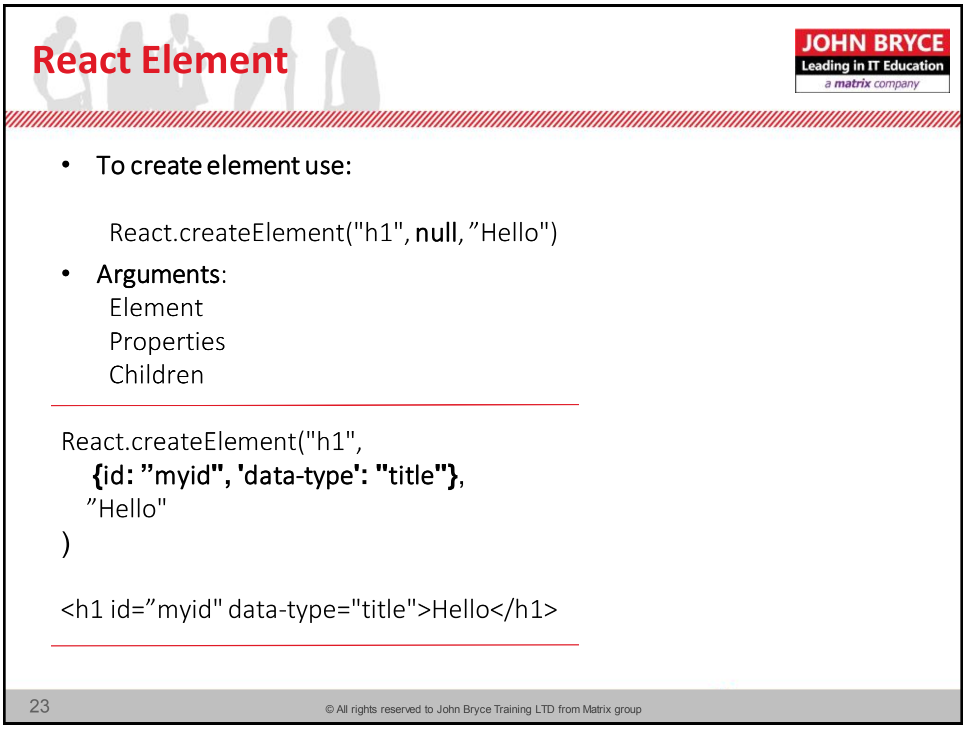


```
const MyButton = (props) => {
    return (
      <button onClick={props.click}>{props.text}</button>
    )
}
```



```
<MyButton click={this.click1} text="sample" />
```

COMPONENTS



React Element

- To create element use:

  React.createElement("h1", **null**, ”Hello")

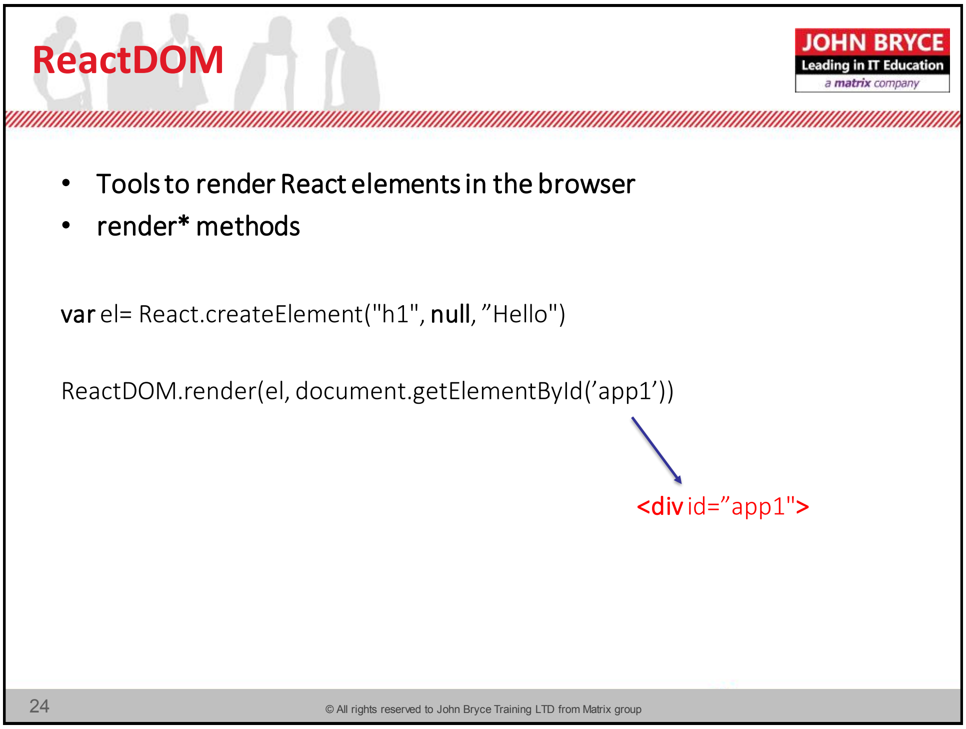
- Arguments:

  Element

  Properties

  Children

```
React.createElement("h1",
    {id: ”myid", 'data-type': "title"},
   ”Hello"
)

<h1 id=”myid" data-type="title">Hello</h1>
```



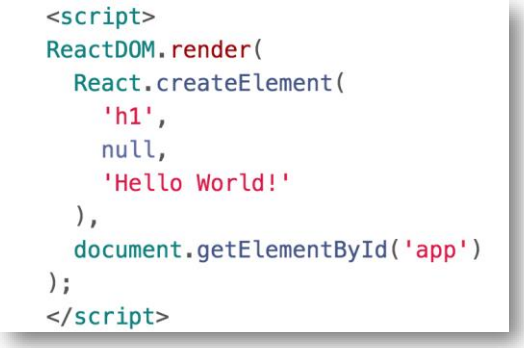
ReactDOM

- Tools to render React elements in the browser
- render* methods

```
var el= React.createElement("h1", null, ”Hello")

ReactDOM.render(el, document.getElementById(’app1’))
```

<div id=”app1">

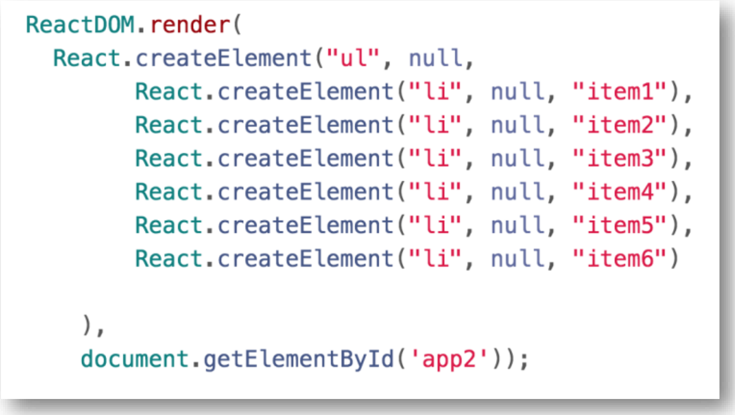
Simple Example



```
<script>
ReactDOM.render(
  React.createElement(
    'h1',
    null,
    'Hello World!'
  ),
  document.getElementById('app')
);
</script>
```

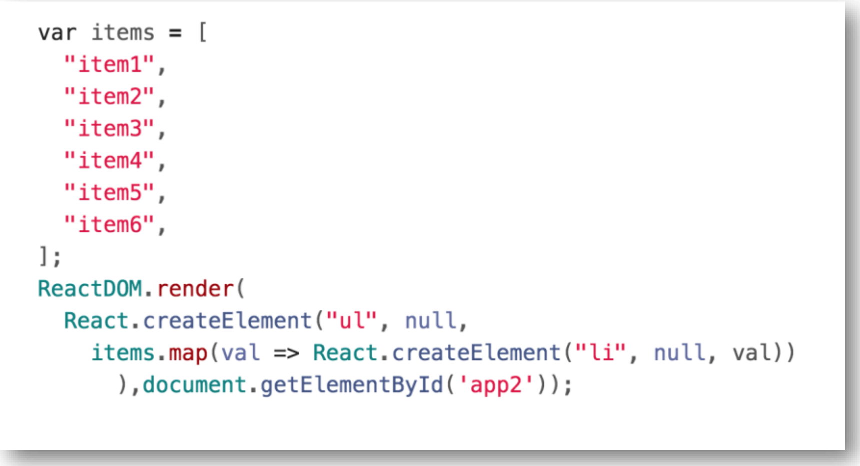
Plain JavaScript

Add More Elements



```
ReactDOM.render(
  React.createElement("ul", null,
        React.createElement("li", null, "item1"),
        React.createElement("li", null, "item2"),
        React.createElement("li", null, "item3"),
        React.createElement("li", null, "item4"),
        React.createElement("li", null, "item5"),
        React.createElement("li", null, "item6")

    ),
    document.getElementById('app2'));
```

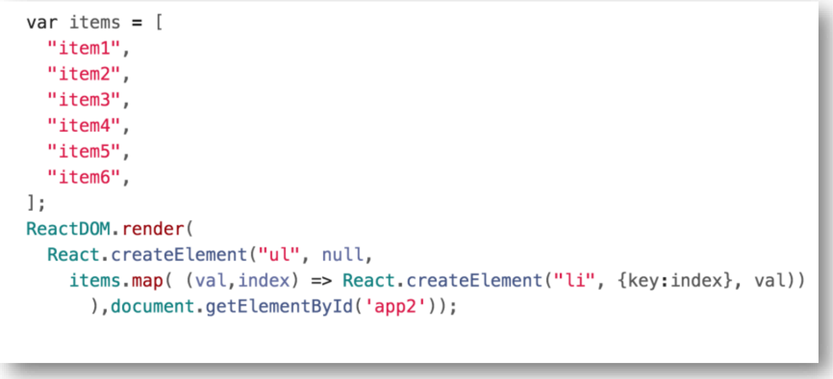
With Code



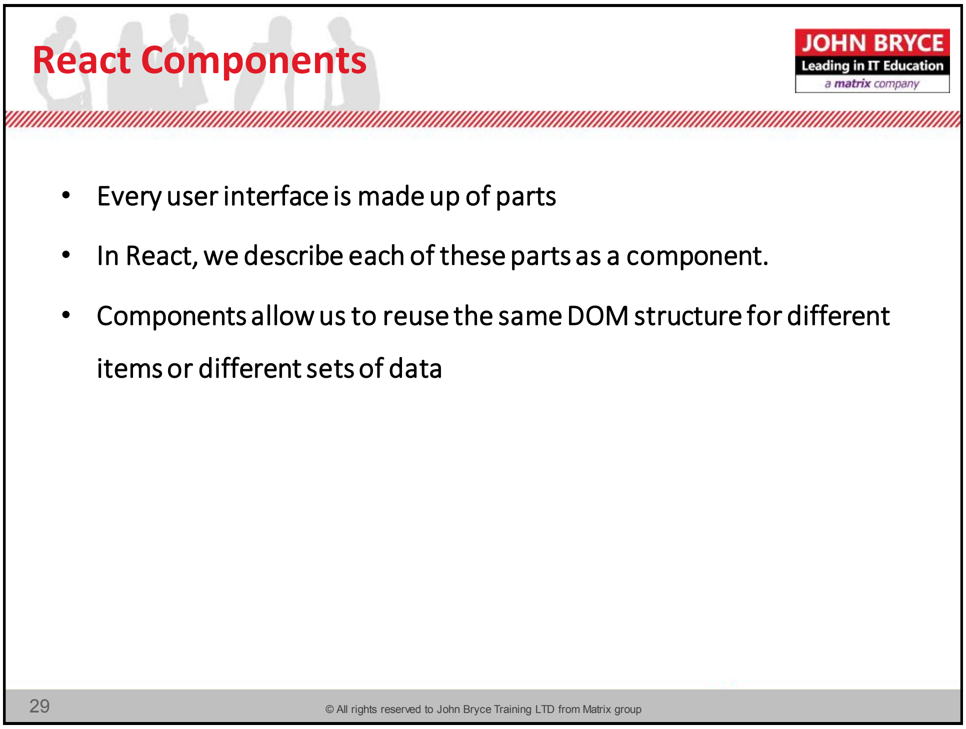
```
var items = [
  "item1",
  "item2",
  "item3",
  "item4",
  "item5",
  "item6",
];
ReactDOM.render(
  React.createElement("ul", null,
    items.map(val => React.createElement("li", null, val))
      ),document.getElementById('app2'));
```

key property

Helps to manipulate the DOM



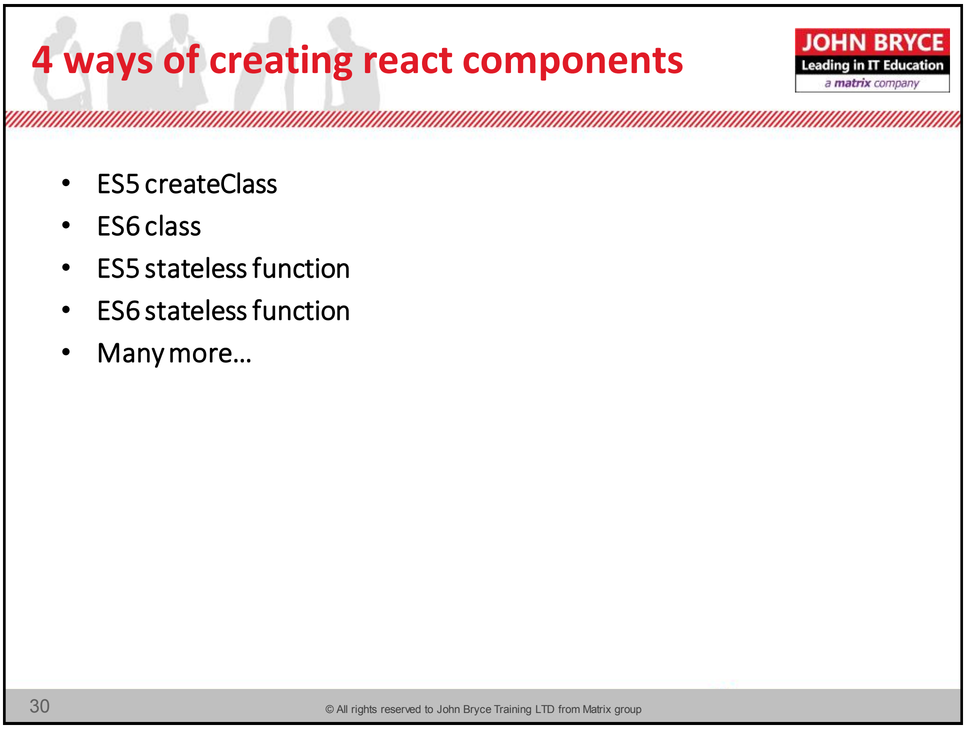
```
var items = [
  "item1",
  "item2",
  "item3",
  "item4",
  "item5",
  "item6",
];
ReactDOM.render(
  React.createElement("ul", null,
    items.map( (val,index) => React.createElement("li", {key:index}, val))
      ),document.getElementById('app2'));
```



React Components

JOHN BRYCE
Leading in IT Education
a matrix company

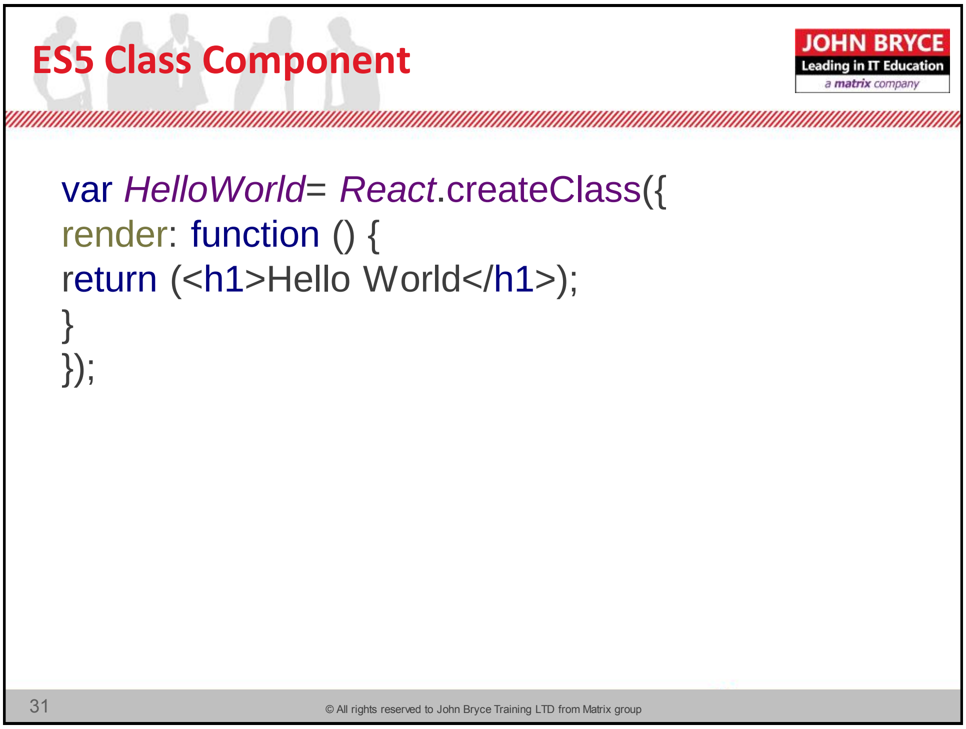
- Every user interface is made up of parts
- In React, we describe each of these parts as a component.
- Components allow us to reuse the same DOM structure for different items or different sets of data



4 ways of creating react components

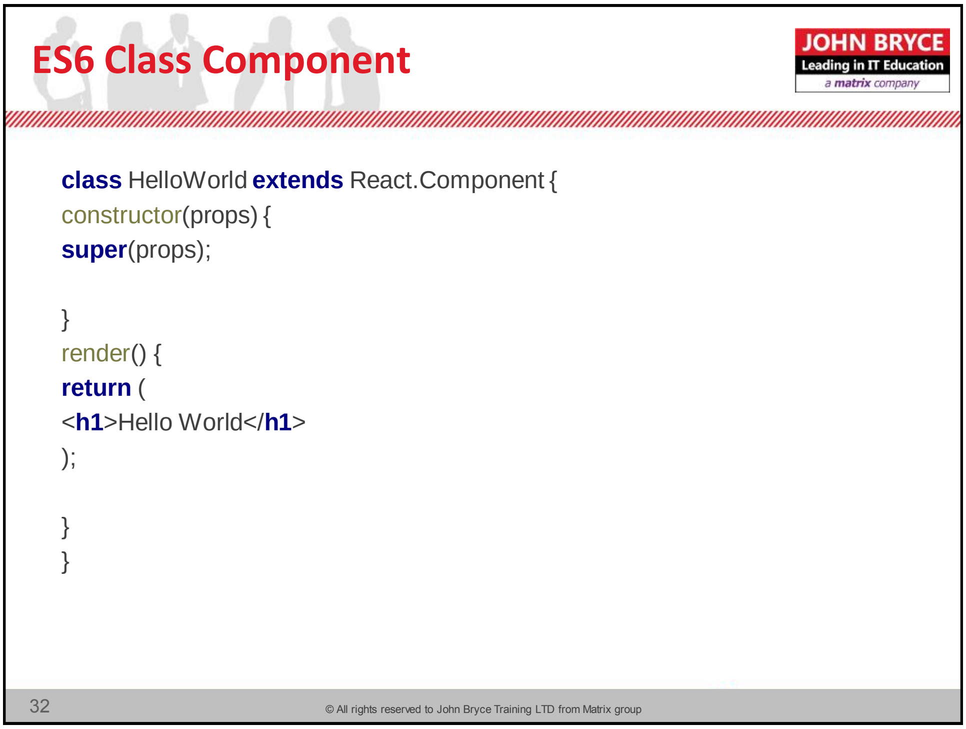
JOHN BRYCE
Leading in IT Education
a matrix company

- ES5 createClass
- ES6 class
- ES5 stateless function
- ES6 stateless function
- Many more…



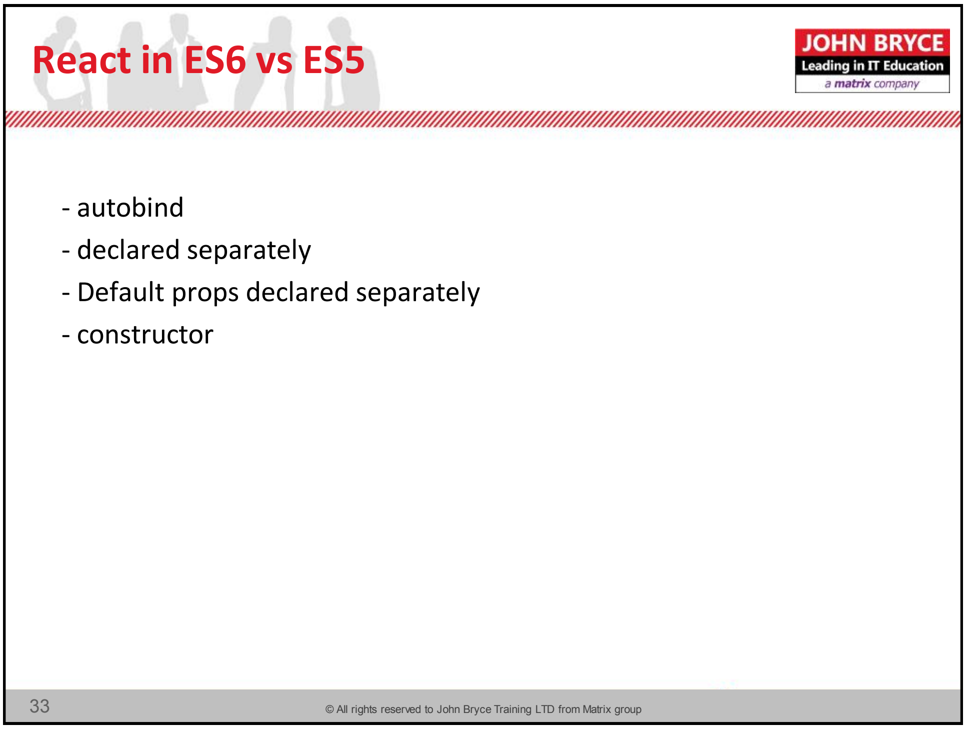
ES5 Class Component

```
var HelloWorld= React.createClass({
render: function () {
return (<h1>Hello World</h1>);
}
});
```



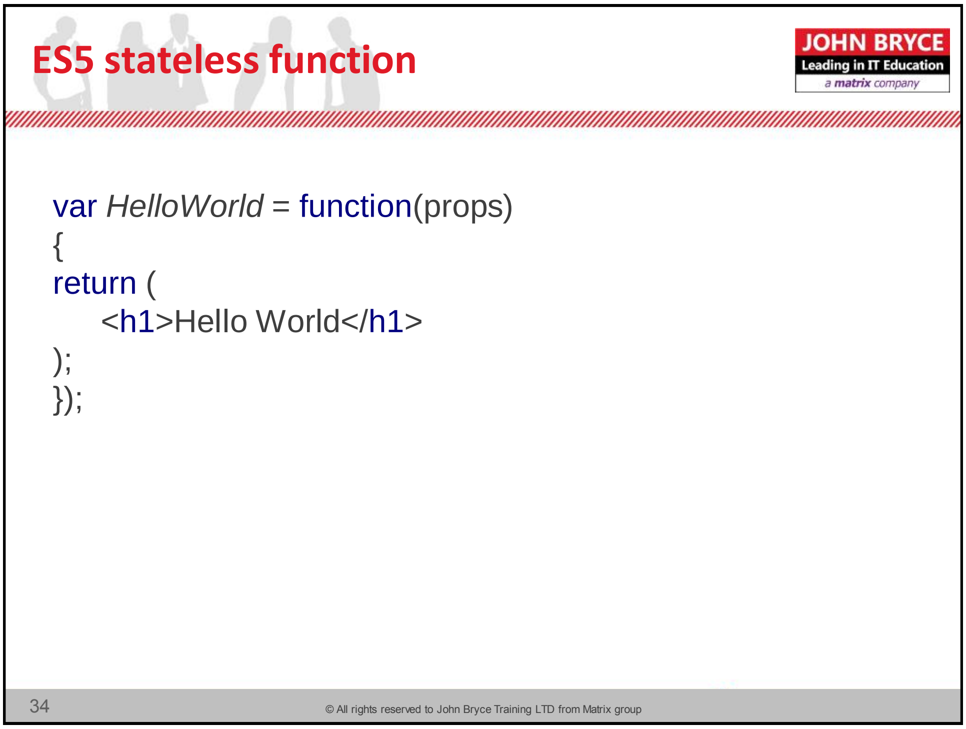
ES6 Class Component

```
class HelloWorld extends React.Component {
constructor(props) {
super(props);

}
render() {
return (
<h1>Hello World</h1>
);

}
}
```



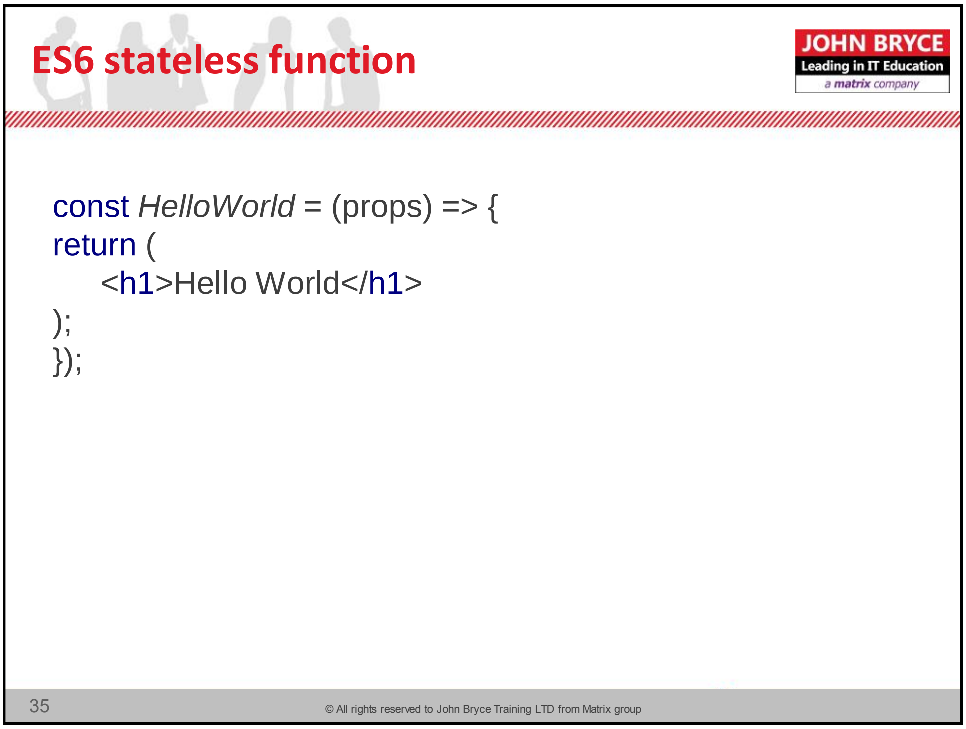
React in ES6 vs ES5

- autobind
- declared separately
- Default props declared separately
- constructor



ES5 stateless function

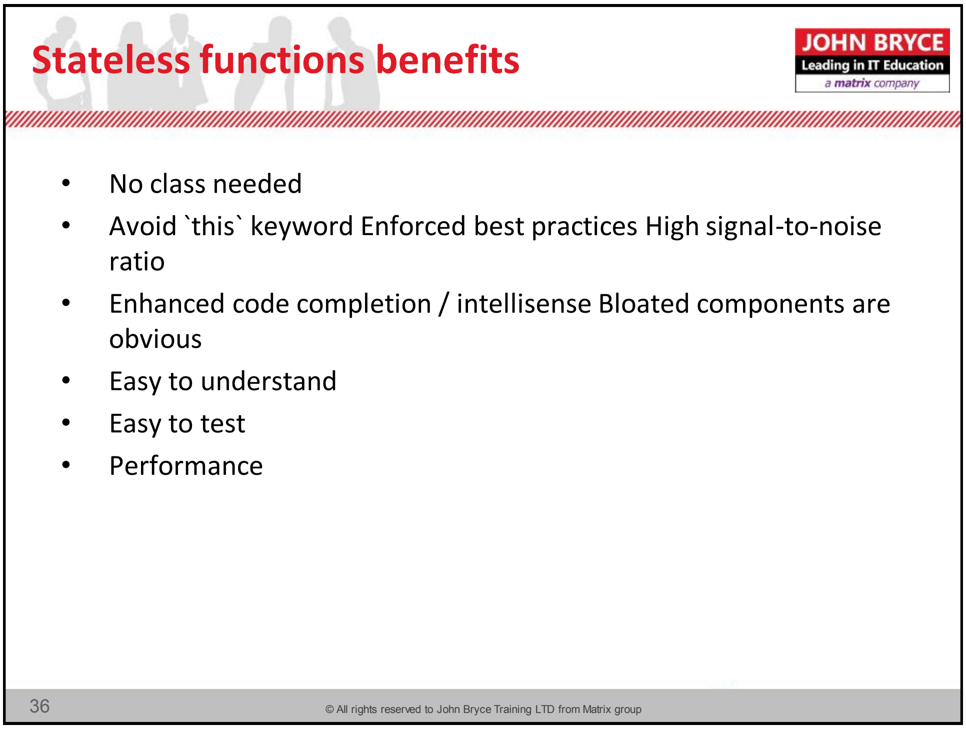
```
var HelloWorld = function(props)
{
return (
    <h1>Hello World</h1>
);
});
```



ES6 stateless function

```
const HelloWorld = (props) => {
return (
    <h1>Hello World</h1>
);
});
```

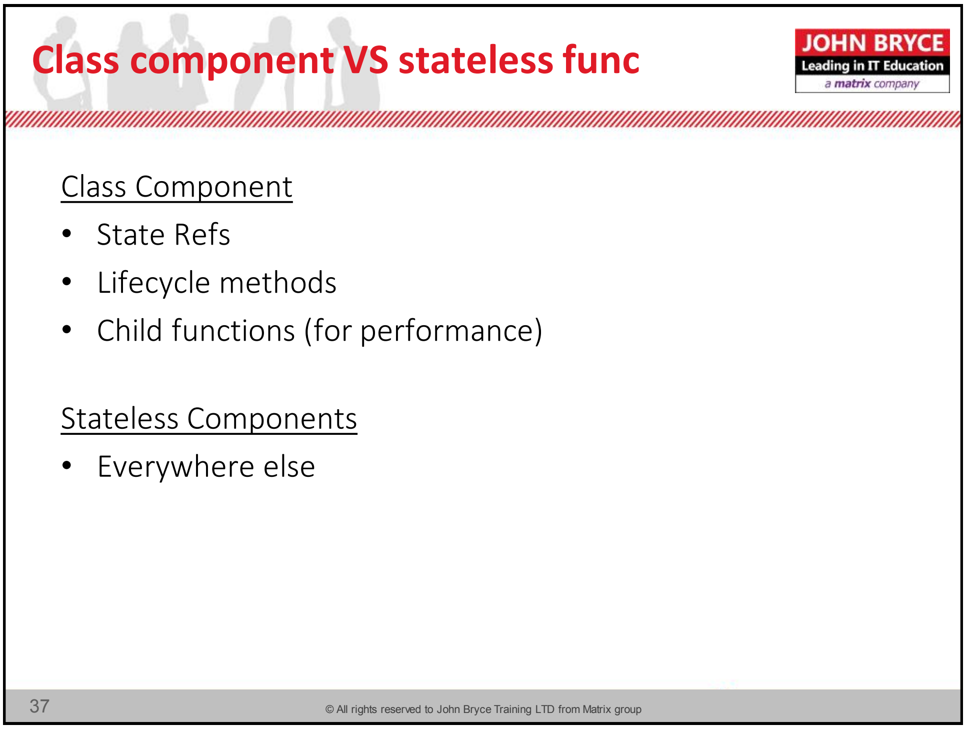
35



Stateless functions benefits

- No class needed
- Avoid `this` keyword Enforced best practices High signal-to-noise ratio
- Enhanced code completion / intellisense Bloated components are obvious
- Easy to understand
- Easy to test
- Performance

36



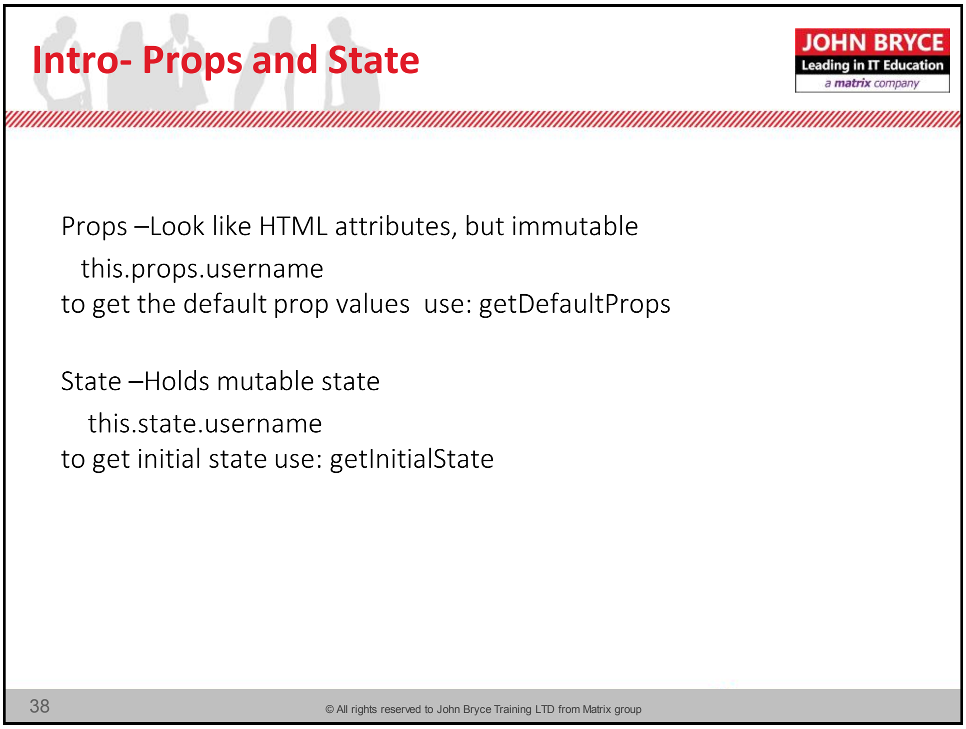
Class component VS stateless func

Class Component

- State Refs
- Lifecycle methods
- Child functions (for performance)

Stateless Components

- Everywhere else



Intro- Props and State

Props –Look like HTML attributes, but immutable

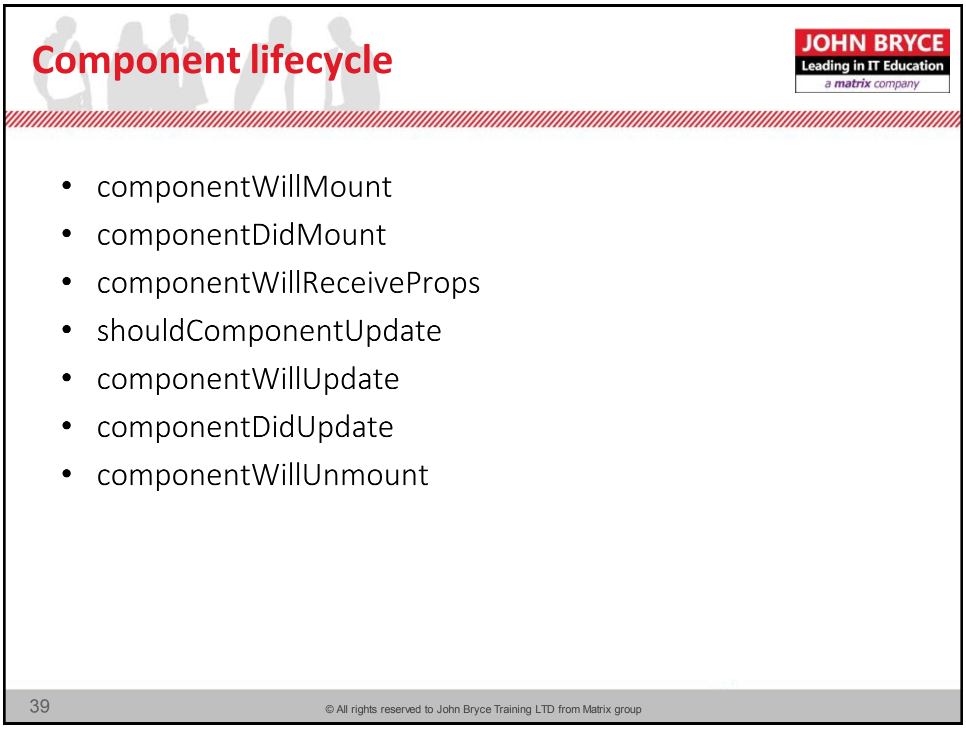
this.props.username

to get the default prop values use: getDefaultProps

State –Holds mutable state

this.state.username

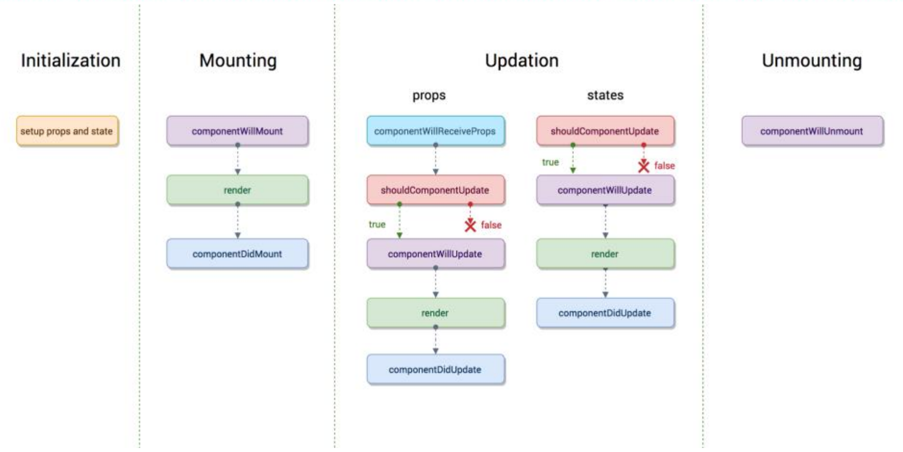
to get initial state use: getInitialState

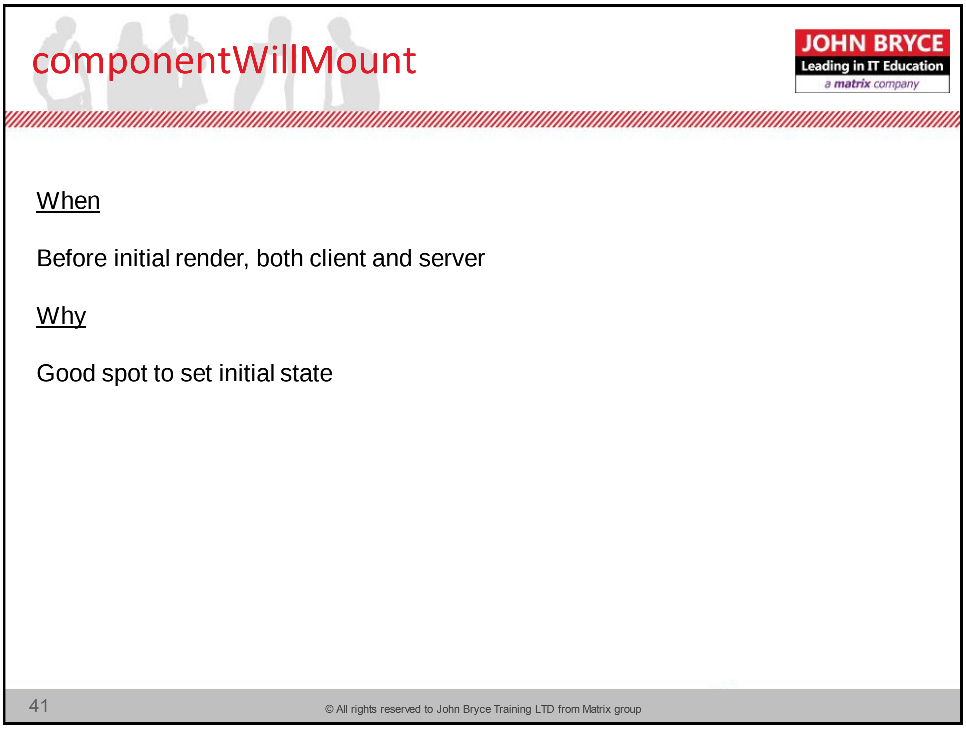


Component lifecycle

- componentWillMount
- componentDidMount
- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- componentDidUpdate
- componentWillUnmount

Component lifecycle





componentWillMount

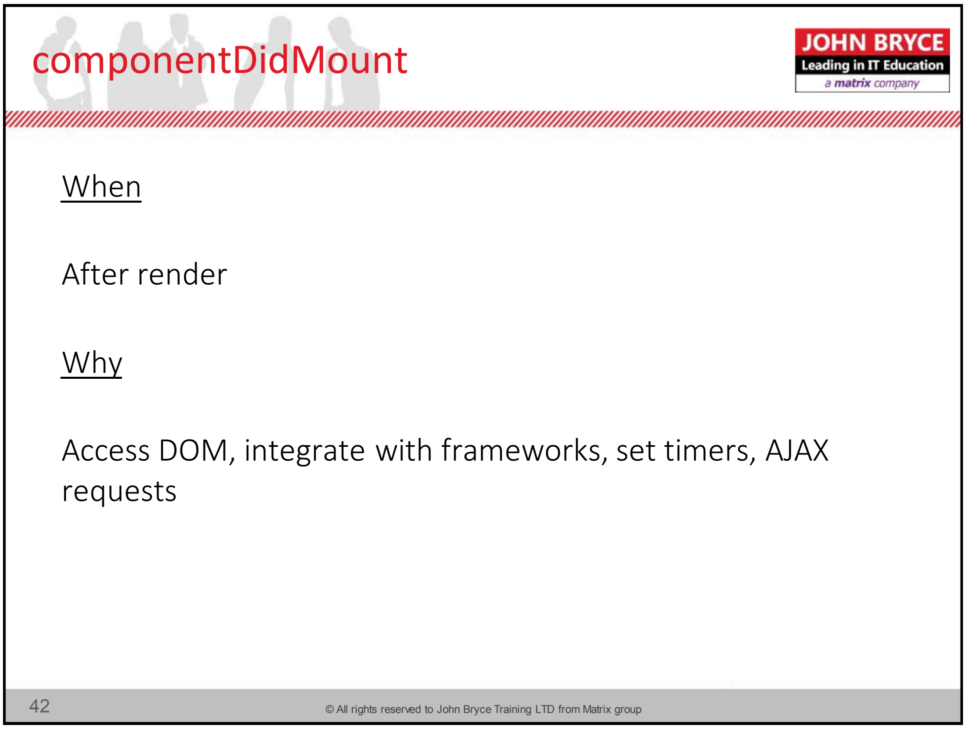
JOHN BRYCE
Leading in IT Education
a matrix company

When

Before initial render, both client and server

Why

Good spot to set initial state



componentDidMount

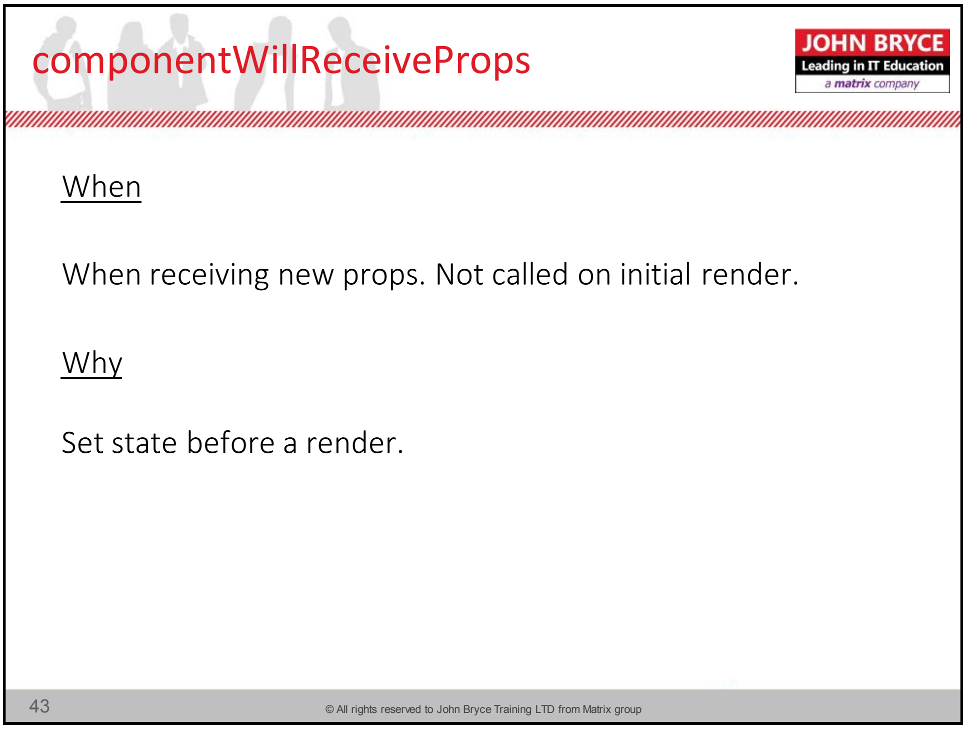
JOHN BRYCE
Leading in IT Education
a matrix company

When

After render

Why

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



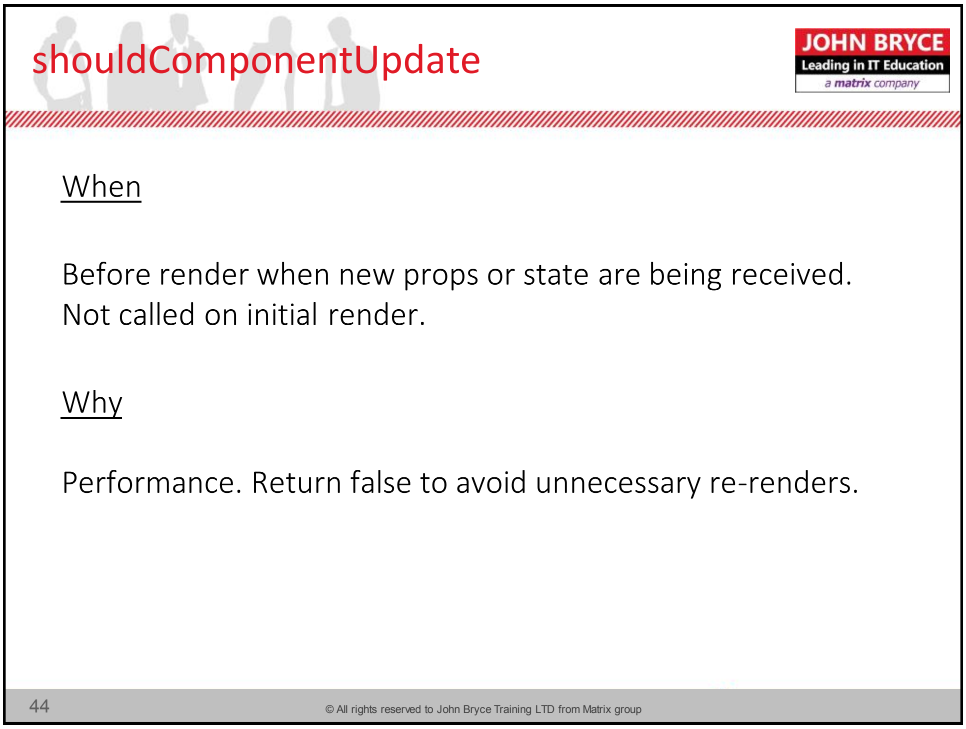
componentWillReceiveProps

When

When receiving new props. Not called on initial render.

Why

Set state before a render.



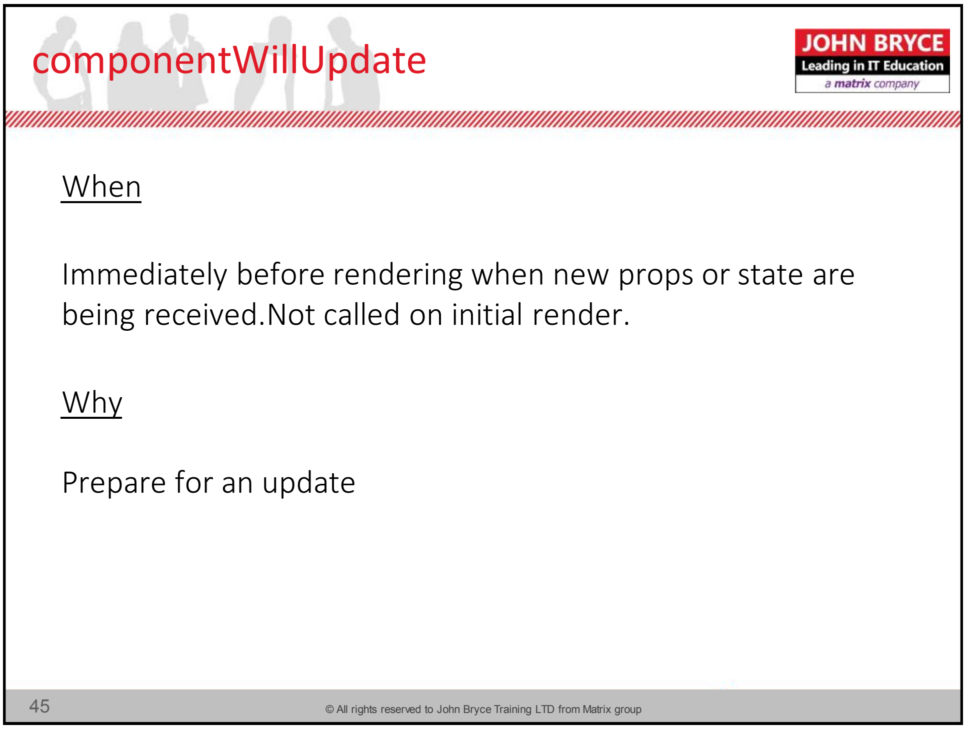
shouldComponentUpdate

When

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

Why

Performance. Return false to avoid unnecessary re-renders.



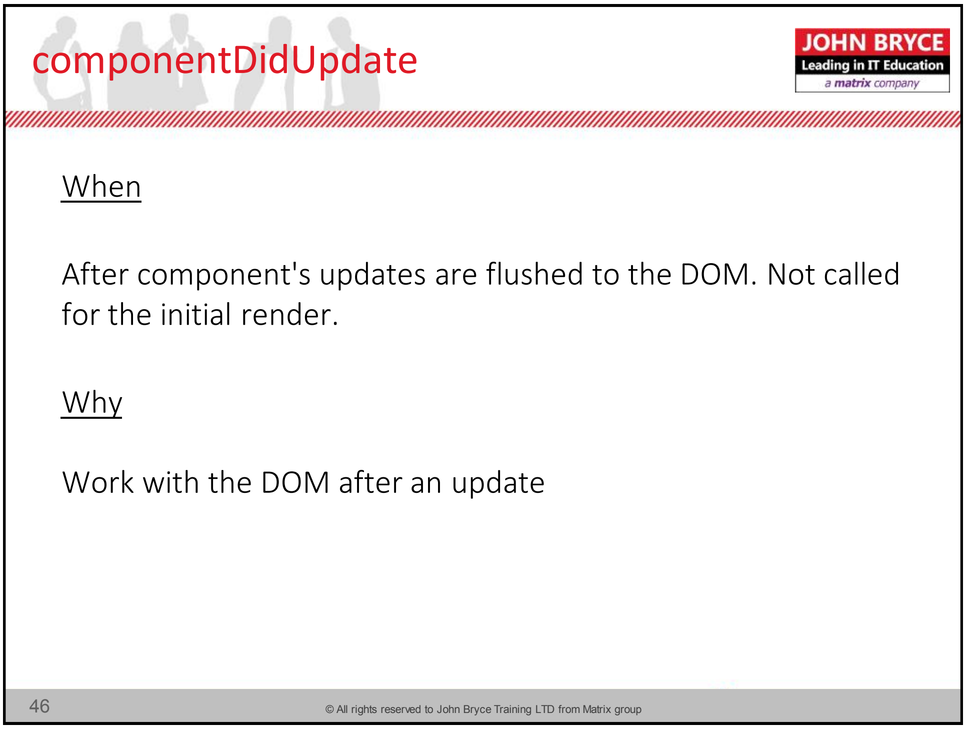
componentWillUpdate

<u>When</u>

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

<u>Why</u>

Prepare for an update



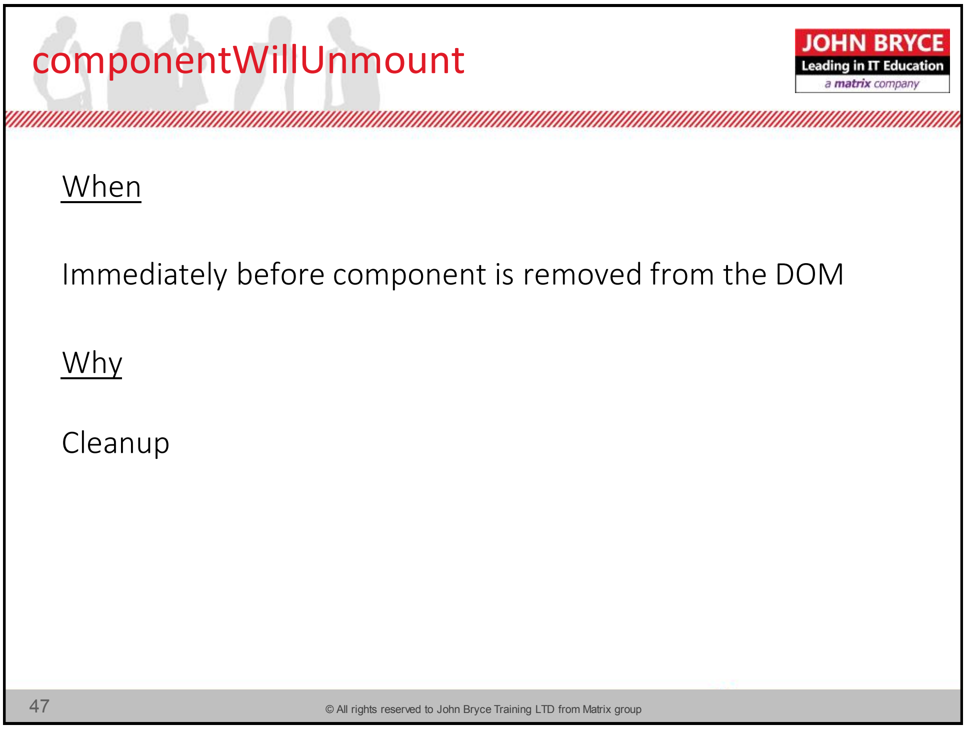
componentDidUpdate

<u>When</u>

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

<u>Why</u>

Work with the DOM after an update



componentWillUnmount

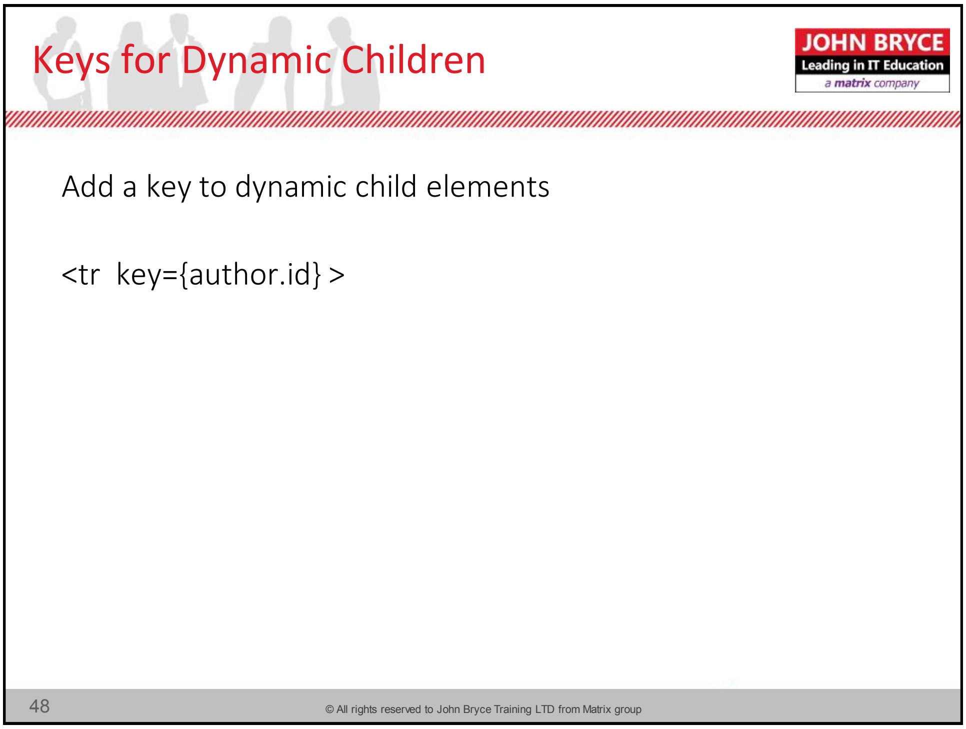
JOHN BRYCE
Leading in IT Education
a matrix company

When

Immediately before component is removed from the DOM

Why

Cleanup



Keys for Dynamic Children

JOHN BRYCE
Leading in IT Education
a matrix company

Add a key to dynamic child elements

```
<tr  key={author.id} >
```

Lifecycle- Summary

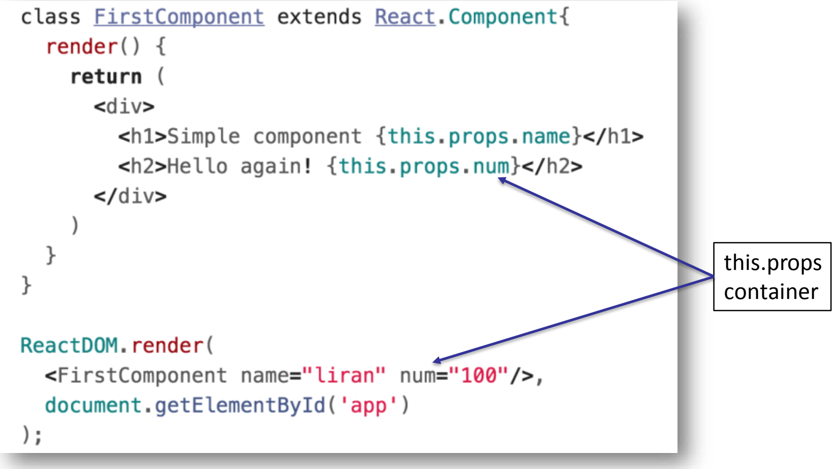
Props - Pass data to child components

State - Data in controller view

Lifecycle - Handle bootstrapping and third party integrations

Properties

One way data binding

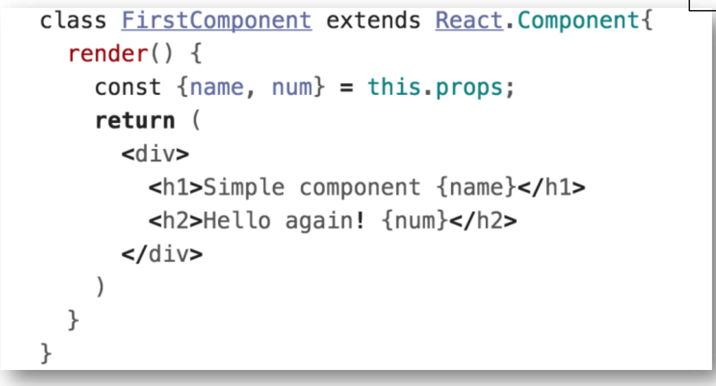


```
class FirstComponent extends React.Component{
  render() {
    return (
      <div>
        <h1>Simple component {this.props.name}</h1>
        <h2>Hello again! {this.props.num}</h2>
      </div>
    )
  }
}

ReactDOM.render(
  <FirstComponent name="liran" num="100"/>,
  document.getElementById('app')
);
```

this.props container

Shorter way

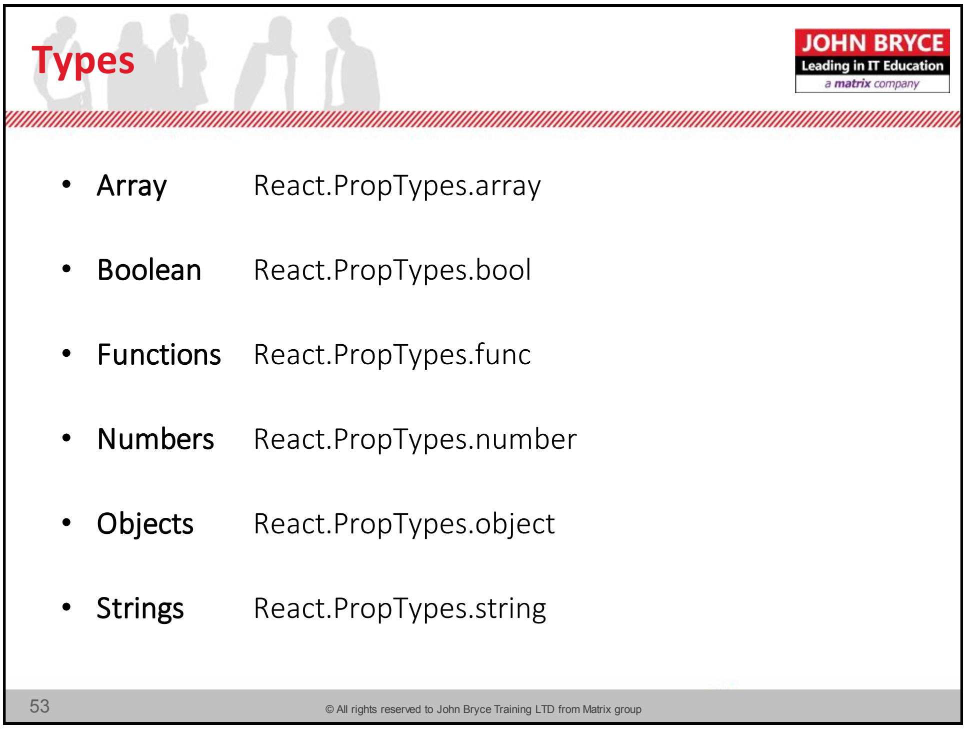
ES6 Destructuring



```
class FirstComponent extends React.Component{
  render() {
    const {name, num} = this.props;
    return (
      <div>
        <h1>Simple component {name}</h1>
        <h2>Hello again! {num}</h2>
      </div>
    )
  }
}
```

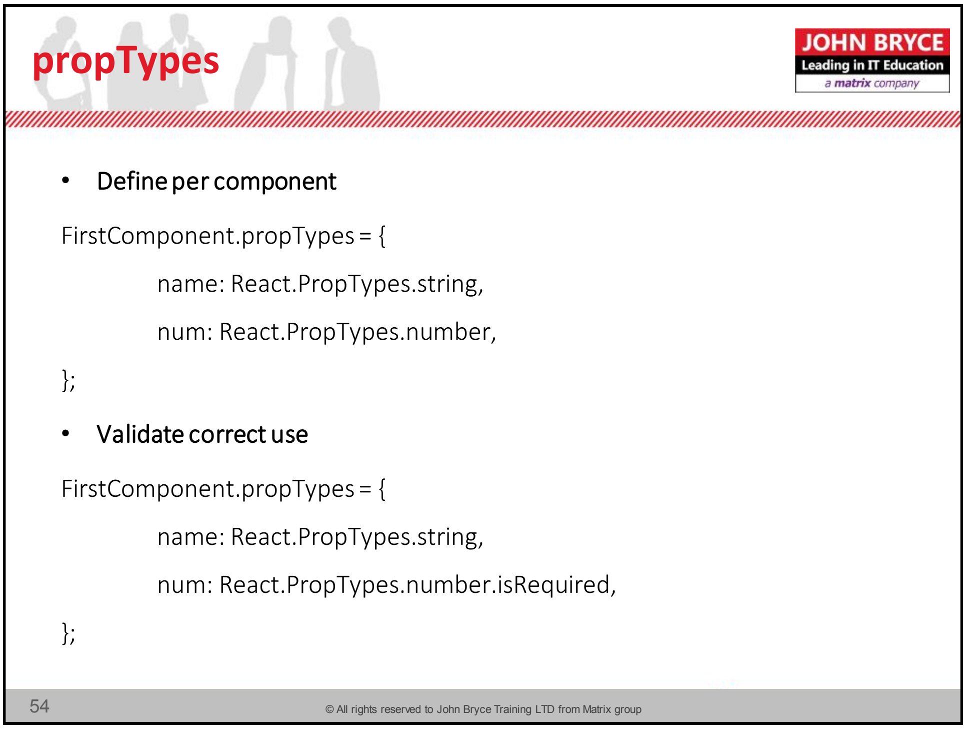
Properties - Types

- React components provide a way to specify and validate property types.
- Using these features will greatly reduce the amount of time spent debugging applications.
- Supplying incorrect property types triggers warnings that can help us find bugs that may have otherwise slipped through the cracks



Types

Array	React.PropTypes.array
Boolean	React.PropTypes.bool
Functions	React.PropTypes.func
Numbers	React.PropTypes.number
Objects	React.PropTypes.object
Strings	React.PropTypes.string



propTypes

- Define per component

```
FirstComponent.propTypes = {
        name: React.PropTypes.string,
        num: React.PropTypes.number,
};
```

- Validate correct use

```
FirstComponent.propTypes = {
        name: React.PropTypes.string,
        num: React.PropTypes.number.isRequired,
};
```

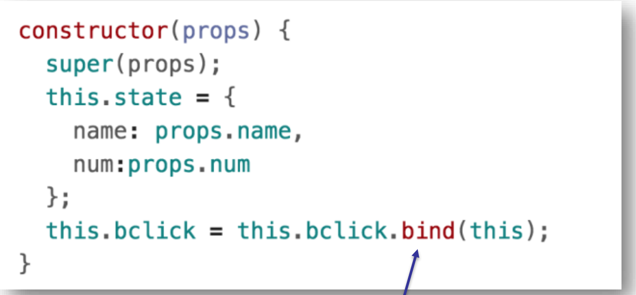
Default Values

You can define the properties default value in case the user didn't supply it

```
FirstComponent.defaultProps = {
        name: 'John',
        num: '20',
};
```

State Management

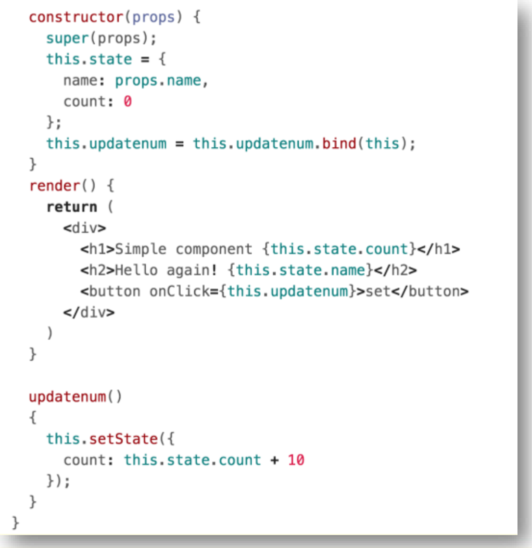
The simple way to create the state object is to use the constructor:



```
constructor(props) {
  super(props);
  this.state = {
    name: props.name,
    num:props.num
  };
  this.bclick = this.bclick.bind(this);
}
```

Important – need to bind the methods to the object

Update The State

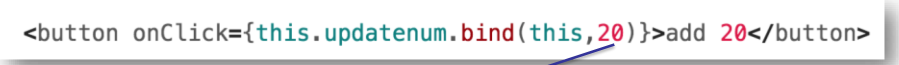


```
  constructor(props) {
    super(props);
    this.state = {
      name: props.name,
      count: 0
    };
    this.updatenum = this.updatenum.bind(this);
  }
  render() {
    return (
      <div>
        <h1>Simple component {this.state.count}</h1>
        <h2>Hello again! {this.state.name}</h2>
        <button onClick={this.updatenum}>set</button>
      </div>
    )
  }

  updatenum()
  {
    this.setState({
      count: this.state.count + 10
    });
  }
}
```

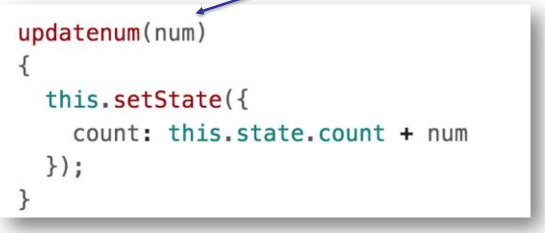
To update the state use the setState method with a new object

Adding Parameter

To add a parameter to event handler use bind:

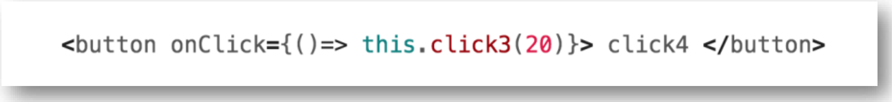


```
<button onClick={this.updatenum.bind(this,20)}>add 20</button>
```

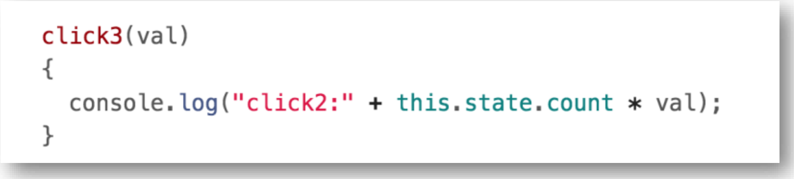


```
updatenum(num)
{
  this.setState({
    count: this.state.count + num
  });
}
```

Using Arrow functions



```
<button onClick={()=> this.click3(20)}> click4 </button>
```



```
click3(val)
{
  console.log("click2:" + this.state.count * val);
}
```

No need to use bind

props vs state

Common ground

Before separating props and state, let's also identify where they overlap.

- Both props and state are plain JS objects
- Both props and state changes trigger a render update
- Both props and state are deterministic. If your Component generates different outputs for the same combination of props and state then you're doing something wrong.

Props VS State

JOHN BRYCE
Leading in IT Education
a matrix company

props

props (short for *properties*) are a Component's **configuration,** its *options* if you may. They are received from above and **immutable** as far as the Component receiving them is concerned.

A Component cannot change its *props,* but it is responsible for putting together the *props* of its child Components.

state

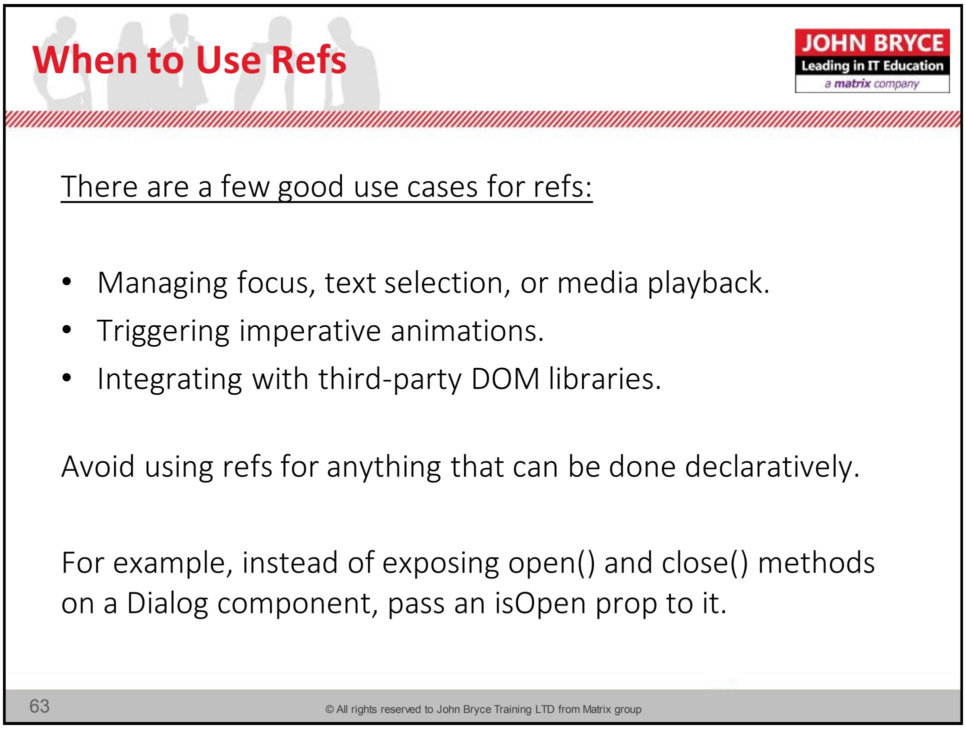
The *state* starts with a default value when a Component mounts and then **suffers from mutations in time (mostly generated from user events).** It's a serializable* representation of one point in time—a snapshot.

References

JOHN BRYCE
Leading in IT Education
a matrix company

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, props are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.



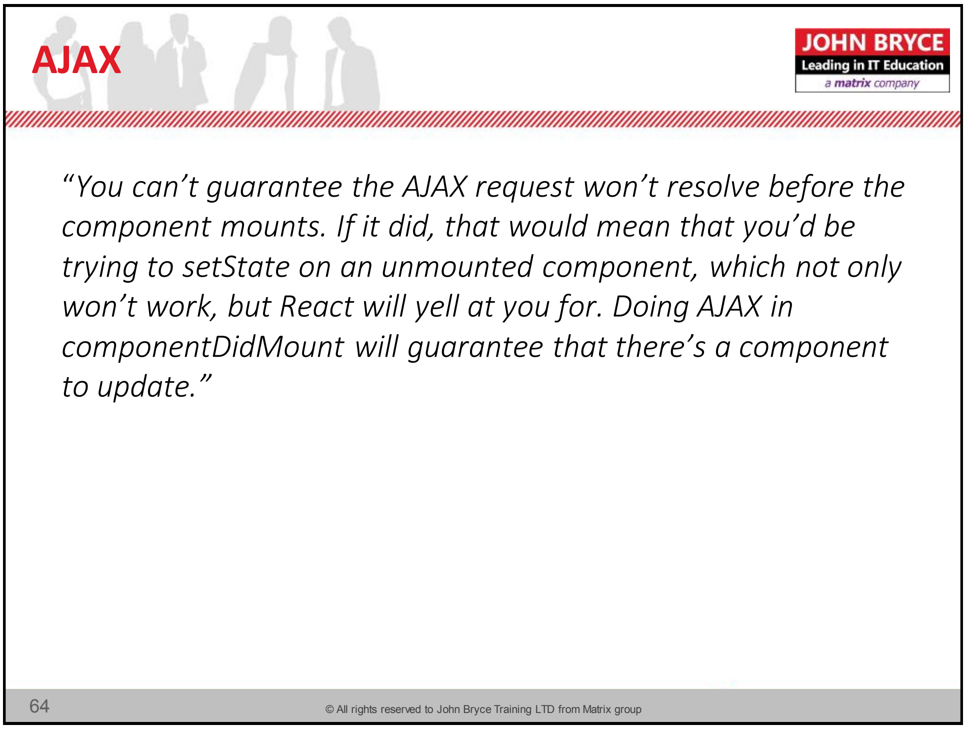
When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

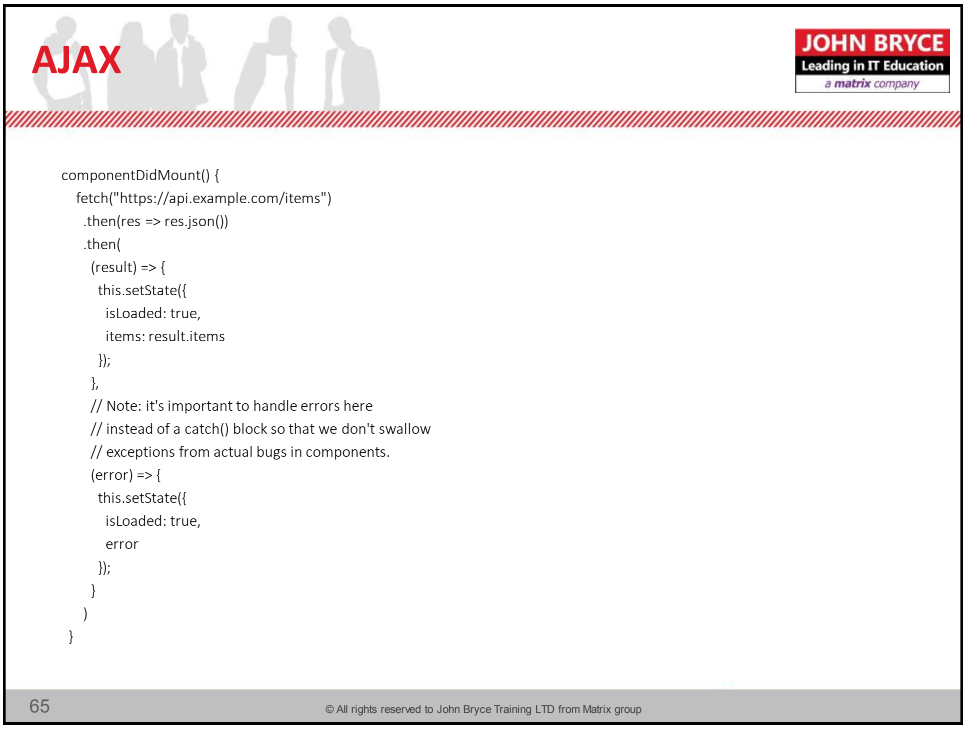
Avoid using refs for anything that can be done declaratively.

For example, instead of exposing open() and close() methods on a Dialog component, pass an isOpen prop to it.



AJAX

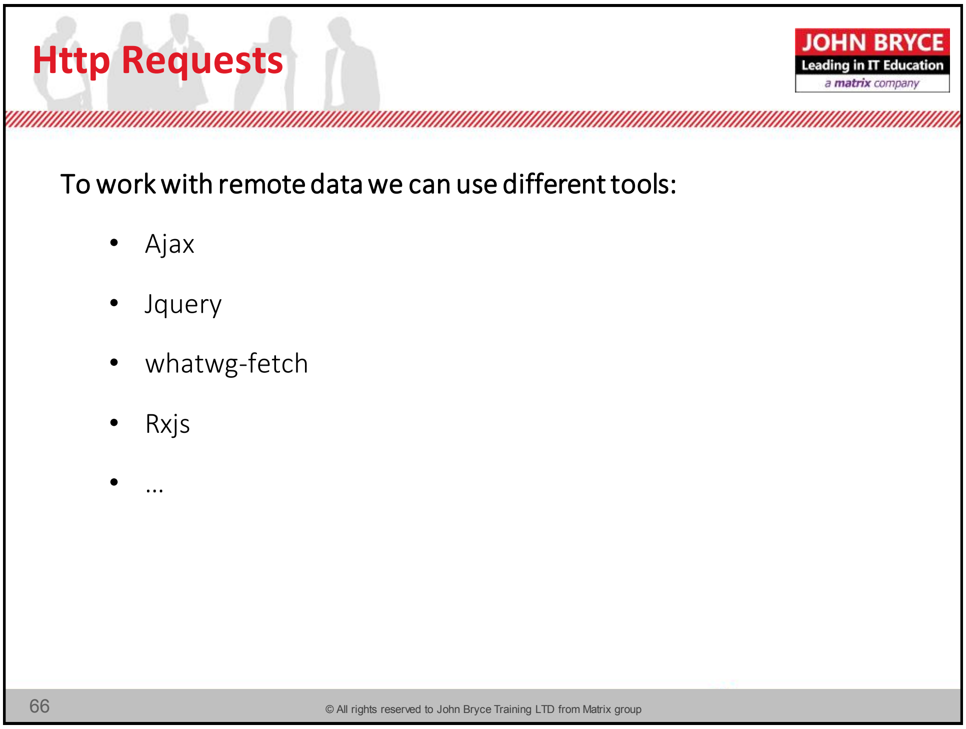
“You can’t guarantee the AJAX request won’t resolve before the component mounts. If it did, that would mean that you’d be trying to setState on an unmounted component, which not only won’t work, but React will yell at you for. Doing AJAX in componentDidMount will guarantee that there’s a component to update.”



AJAX

JOHN BRYCE
Leading in IT Education
a matrix company

```
componentDidMount() {
  fetch("https://api.example.com/items")
   .then(res => res.json())
   .then(
    (result) => {
     this.setState({
      isLoaded: true,
      items: result.items
     });
    },
    // Note: it's important to handle errors here
    // instead of a catch() block so that we don't swallow
    // exceptions from actual bugs in components.
    (error) => {
     this.setState({
      isLoaded: true,
      error
     });
    }
   )
 }
```



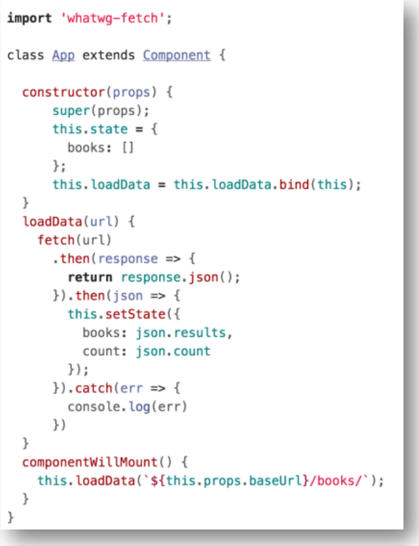
Http Requests

JOHN BRYCE
Leading in IT Education
a matrix company

To work with remote data we can use different tools:

- Ajax
- Jquery
- whatwg-fetch
- Rxjs
- …

Using whatwg-fetch



```
import 'whatwg-fetch';

class App extends Component {

  constructor(props) {
      super(props);
      this.state = {
        books: []
      };
      this.loadData = this.loadData.bind(this);
  }
  loadData(url) {
    fetch(url)
      .then(response => {
        return response.json();
      }).then(json => {
        this.setState({
          books: json.results,
          count: json.count
        });
      }).catch(err => {
        console.log(err)
      })
  }
  componentWillMount() {
    this.loadData(`${this.props.baseUrl}/books/`);
  }
}
```

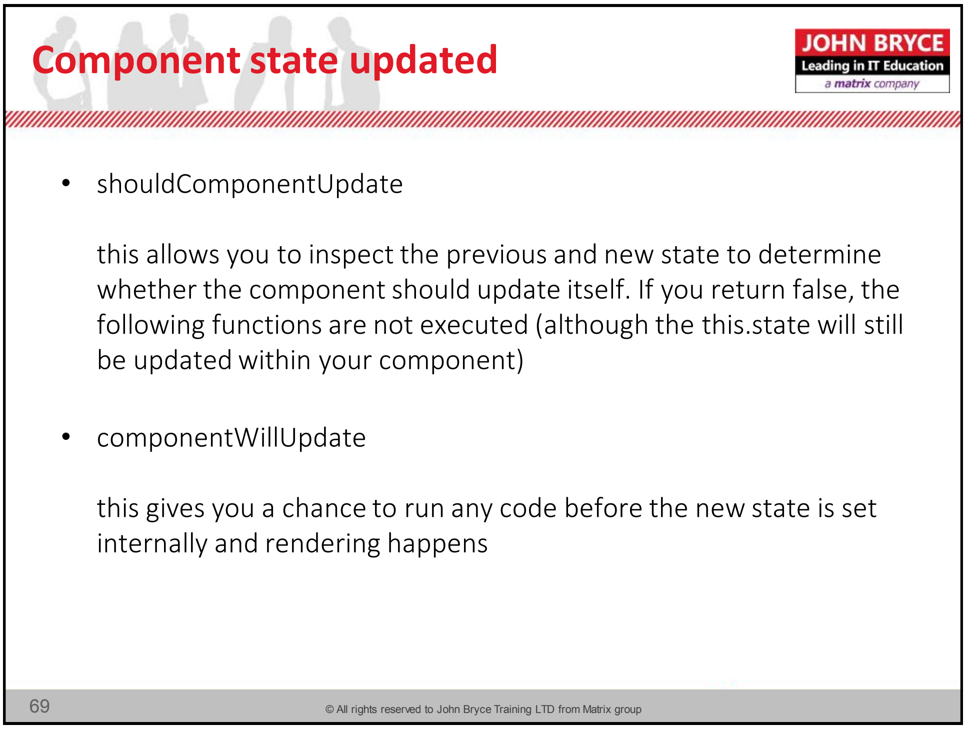
- Create the state objects to store the data
- Use promise to retrieve the data asynchronously
- Use the map method to bind the array to the component

Component state updated

setState is asynchronous

```
this.setState({page: page}, function stateUpdateComplete() {
   console.log(this.state.page)
   this.findByName();
}.bind(this));
```

After we call setState, three functions are called
https://facebook.github.io/react/docs/component-specs.html



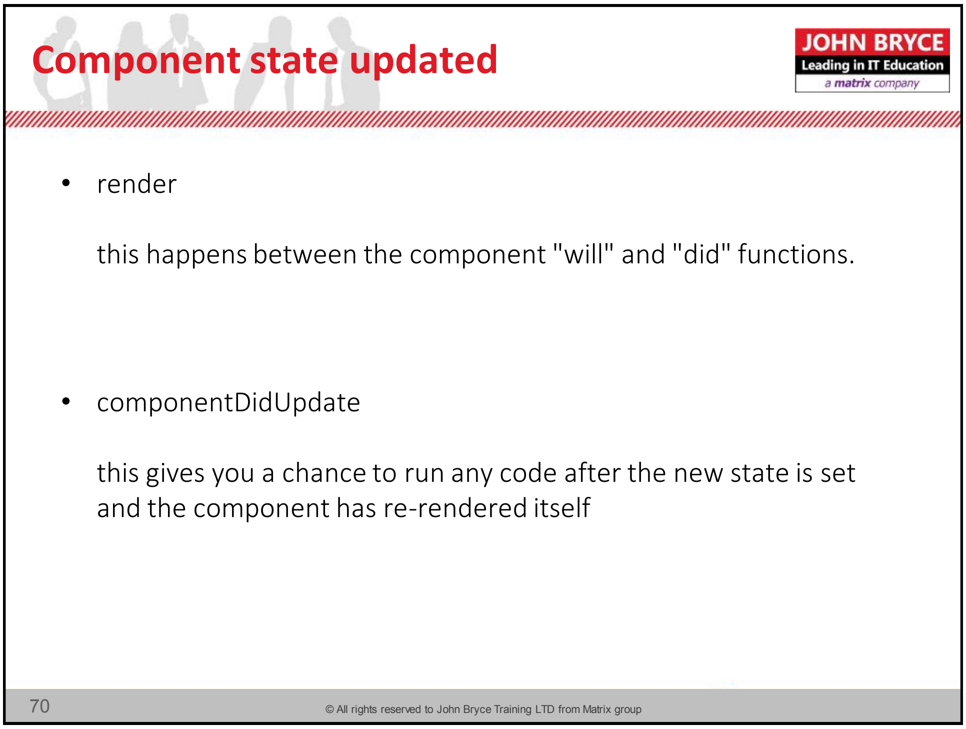
Component state updated

- shouldComponentUpdate

  this allows you to inspect the previous and new state to determine whether the component should update itself. If you return false, the following functions are not executed (although the this.state will still be updated within your component)

- componentWillUpdate

  this gives you a chance to run any code before the new state is set internally and rendering happens



Component state updated

- render

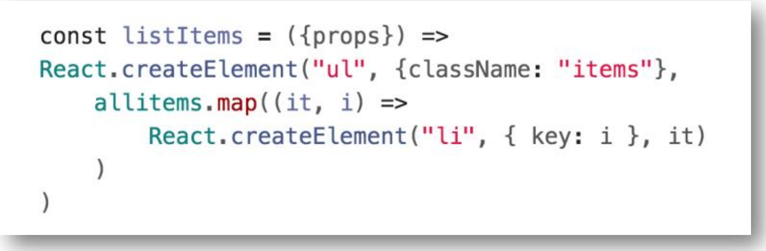
  this happens between the component "will" and "did" functions.

- componentDidUpdate

  this gives you a chance to run any code after the new state is set and the component has re-rendered itself

Stateless Components

- Arrow functions are not objects – no this
- Can be used with simple components if no state is needed



```
const listItems = ({props}) =>
React.createElement("ul", {className: "items"},
    allitems.map((it, i) =>
        React.createElement("li", { key: i }, it)
    )
)
```

Stateless vs stateful components

Stateful Components

Stateful components are always class components. As previously mentioned, stateful components have a state that gets initialized in the constructor.

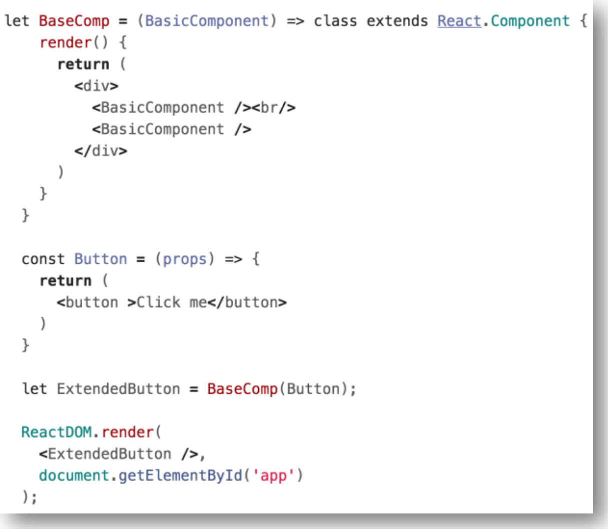
Stateless vs stateful components

Stateless Components

You can use either a function or a class for creating stateless components. But unless you need to use a lifecycle hook in your components, you should go for stateless functional components. There are a lot of benefits if you decide to use stateless functional components here; they are easy to write, understand, and test, and you can avoid the this keyword altogether. However, as of React v16, there are no performance benefits from using stateless functional components over class components.

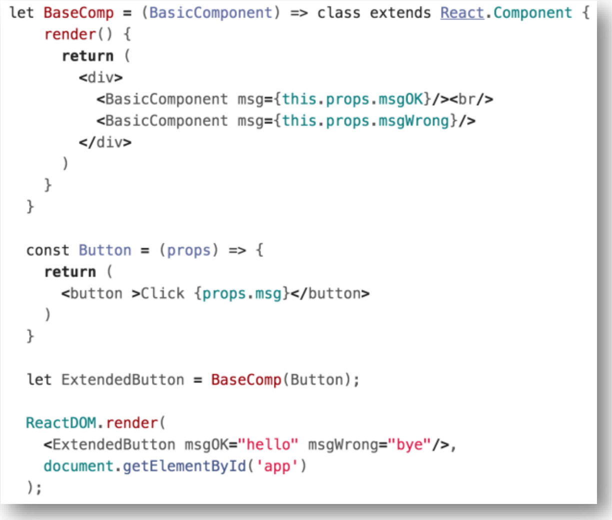
The downside is that you can't have lifecycle hooks. The lifecycle method ShouldComponentUpdate() is often used to optimize performance and to manually control what gets rerendered. You can't use that with functional components yet. Refs are also not supported.

Extending Components



```
let BaseComp = (BasicComponent) => class extends React.Component {
    render() {
      return (
        <div>
          <BasicComponent /><br/>
          <BasicComponent />
        </div>
      )
    }
  }

  const Button = (props) => {
    return (
      <button >Click me</button>
    )
  }

  let ExtendedButton = BaseComp(Button);

  ReactDOM.render(
    <ExtendedButton />,
    document.getElementById('app')
  );
```

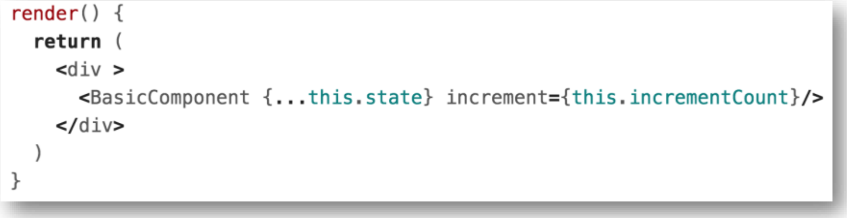
Using props



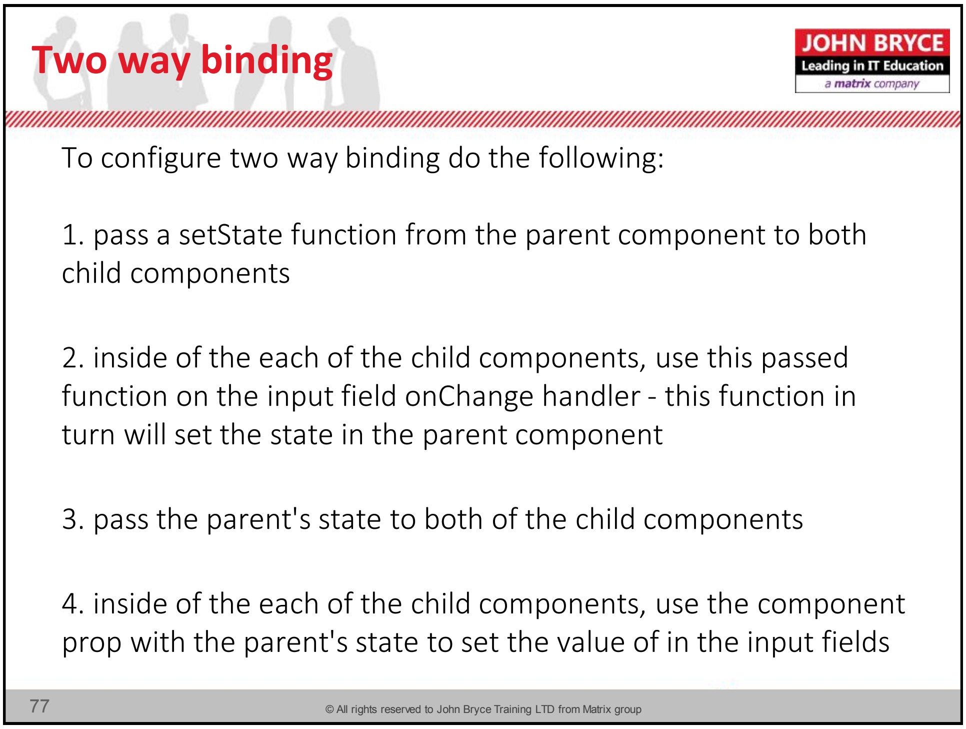
```
let BaseComp = (BasicComponent) => class extends React.Component {
    render() {
      return (
        <div>
          <BasicComponent msg={this.props.msgOK}/><br/>
          <BasicComponent msg={this.props.msgWrong}/>
        </div>
      )
    }
  }

  const Button = (props) => {
    return (
      <button >Click {props.msg}</button>
    )
  }

  let ExtendedButton = BaseComp(Button);

  ReactDOM.render(
    <ExtendedButton msgOK="hello" msgWrong="bye"/>,
    document.getElementById('app')
  );
```

Using state

Convert the state to properties for the child component:



```
render() {
  return (
    <div >
      <BasicComponent {...this.state} increment={this.incrementCount}/>
    </div>
  )
}
```



Two way binding

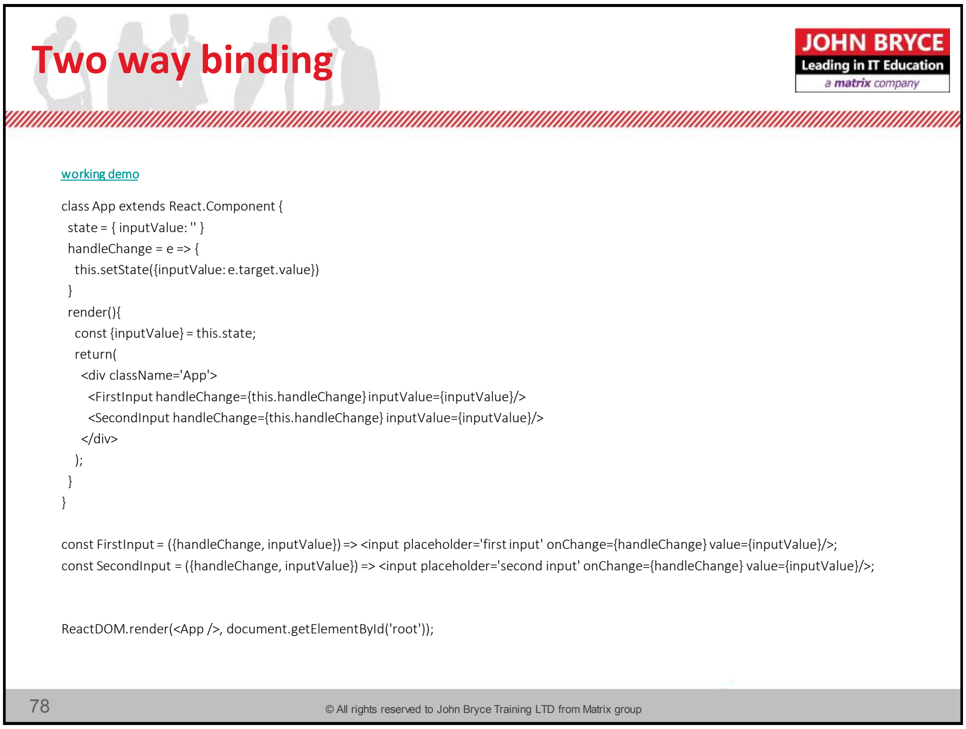
To configure two way binding do the following:

1. pass a setState function from the parent component to both child components

2. inside of the each of the child components, use this passed function on the input field onChange handler - this function in turn will set the state in the parent component

3. pass the parent's state to both of the child components

4. inside of the each of the child components, use the component prop with the parent's state to set the value of in the input fields



Two way binding

working demo

```
class App extends React.Component {
  state = { inputValue: '' }
  handleChange = e => {
    this.setState({inputValue: e.target.value})
  }
  render(){
    const {inputValue} = this.state;
    return(
      <div className='App'>
        <FirstInput handleChange={this.handleChange} inputValue={inputValue}/>
        <SecondInput handleChange={this.handleChange} inputValue={inputValue}/>
      </div>
    );
  }
}

const FirstInput = ({handleChange, inputValue}) => <input placeholder='first input' onChange={handleChange} value={inputValue}/>;
const SecondInput = ({handleChange, inputValue}) => <input placeholder='second input' onChange={handleChange} value={inputValue}/>;


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

ROUTING

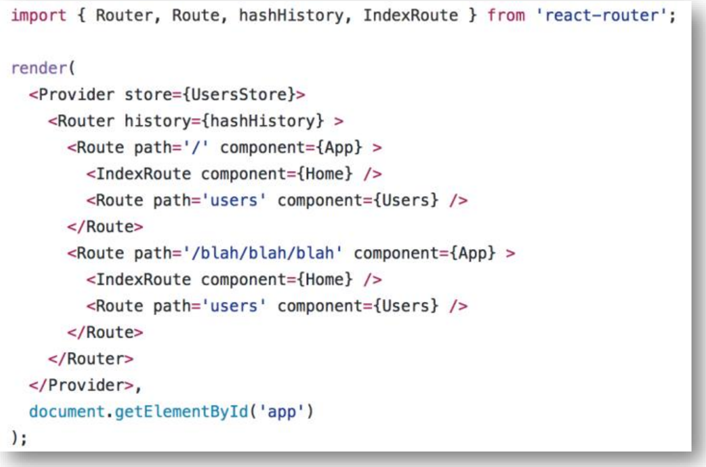
React Router

- Nested views map to nested routes
- Declarative
- Used at Facebook
- Inspired by Ember

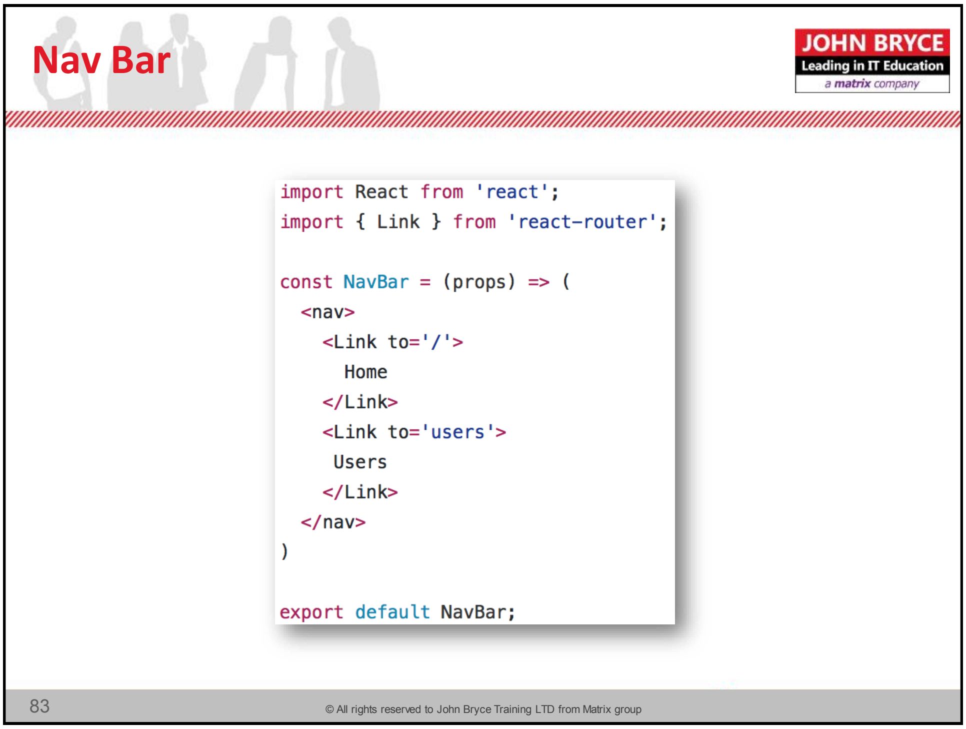
React Route

```
ReactDOM.render((
  <Router>
    <Route path="/" component={Home} />
    <Route path="/users" component={Users} />
    <Route path="/widgets" component={Widgets} />
  </Router>
), document.getElementById('root'));
```

React Router



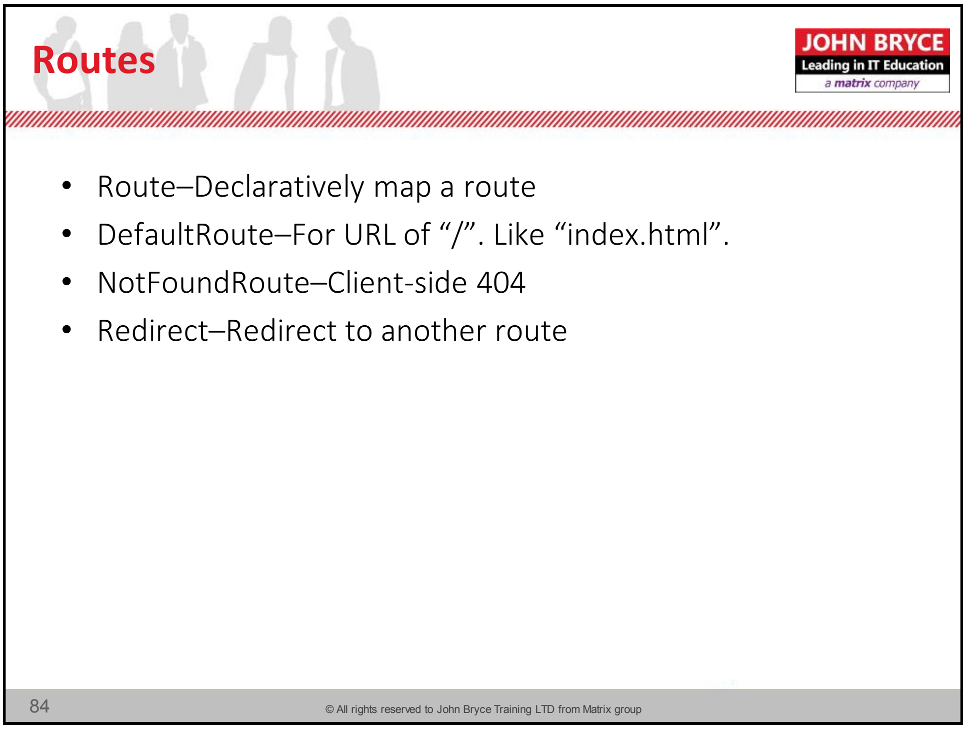
```
import { Router, Route, hashHistory, IndexRoute } from 'react-router';

render(
  <Provider store={UsersStore}>
    <Router history={hashHistory} >
      <Route path='/' component={App} >
        <IndexRoute component={Home} />
        <Route path='users' component={Users} />
      </Route>
      <Route path='/blah/blah/blah' component={App} >
        <IndexRoute component={Home} />
        <Route path='users' component={Users} />
      </Route>
    </Router>
  </Provider>,
  document.getElementById('app')
);
```



Nav Bar

JOHN BRYCE
Leading in IT Education
a matrix company

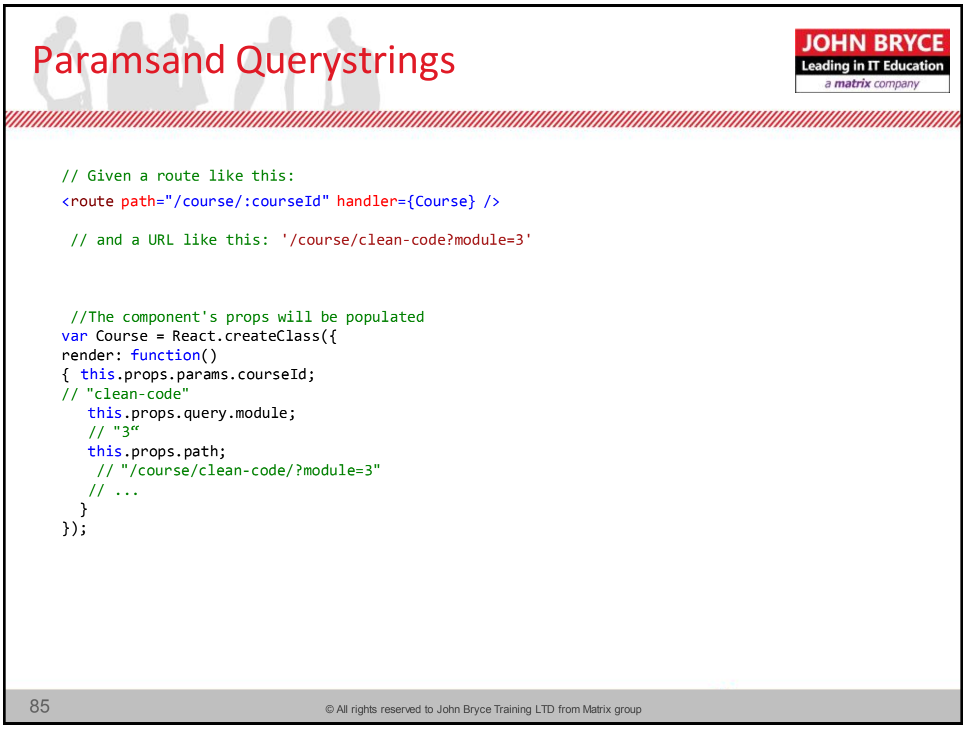
```
import React from 'react';
import { Link } from 'react-router';

const NavBar = (props) => (
  <nav>
    <Link to='/'>
      Home
    </Link>
    <Link to='users'>
     Users
    </Link>
  </nav>
)

export default NavBar;
```



Routes

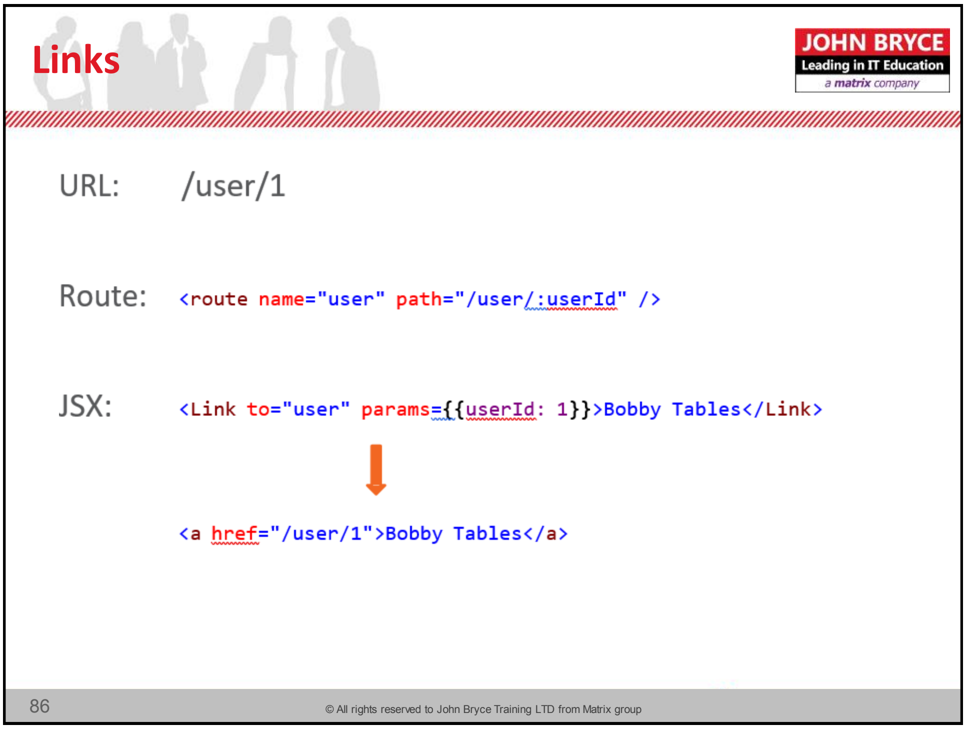
JOHN BRYCE
Leading in IT Education
a matrix company

- Route–Declaratively map a route
- DefaultRoute–For URL of “/”. Like “index.html”.
- NotFoundRoute–Client-side 404
- Redirect–Redirect to another route



Paramsand 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"
   // ...
  }
});
```



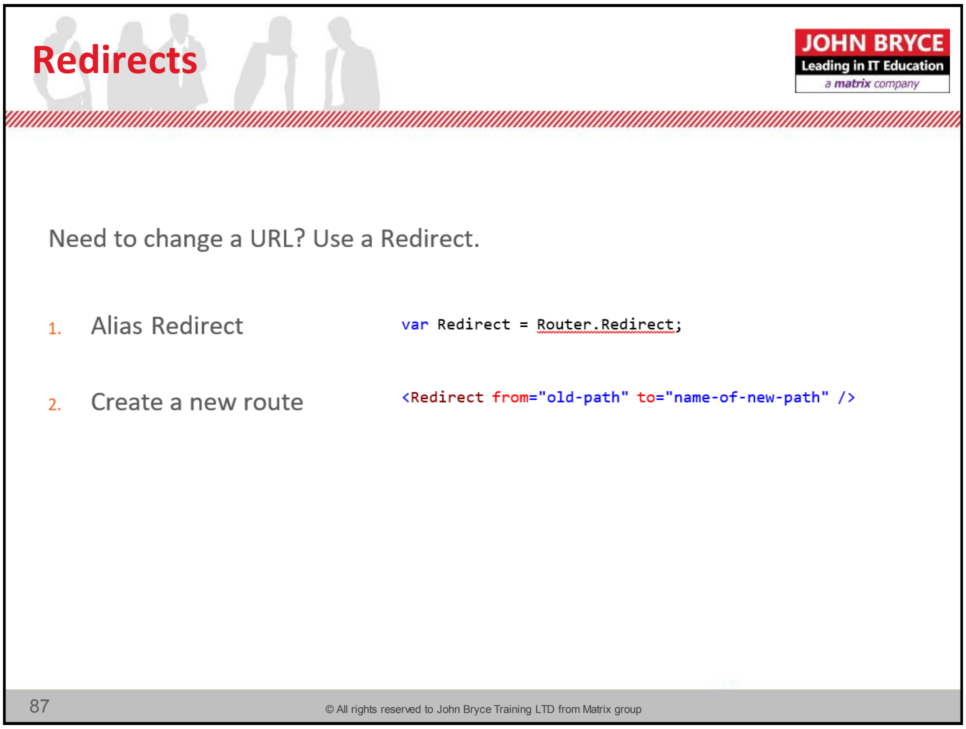
Links

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>`



Redirects

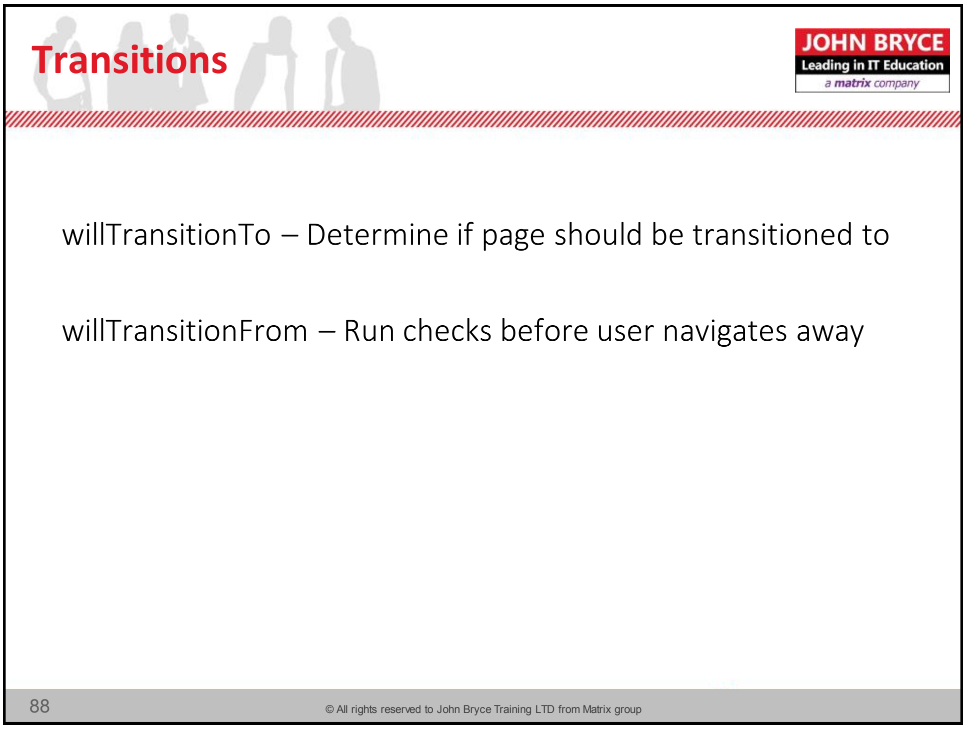
Need to change a URL? Use a Redirect.

1. Alias Redirect

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

2. Create a new route

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

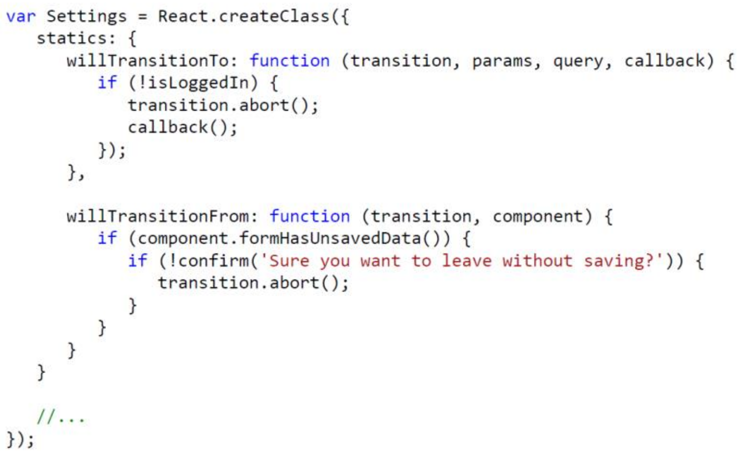


Transitions

willTransitionTo – Determine if page should be transitioned to

willTransitionFrom – Run checks before user navigates away

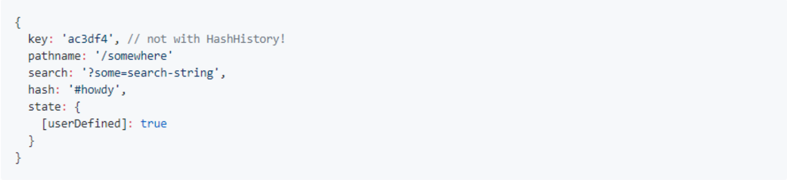
Transitions



```
var Settings = React.createClass({
   statics: {
      willTransitionTo: function (transition, params, query, callback) {
         if (!isLoggedIn) {
            transition.abort();
            callback();
         });
      },

      willTransitionFrom: function (transition, component) {
         if (component.formHasUnsavedData()) {
            if (!confirm('Sure you want to leave without saving?')) {
               transition.abort();
            }
         }
      }
   }

   //...
});
```

Location

- Locations represent where the app is now, where you want it to go, or even where it was. It looks like this:



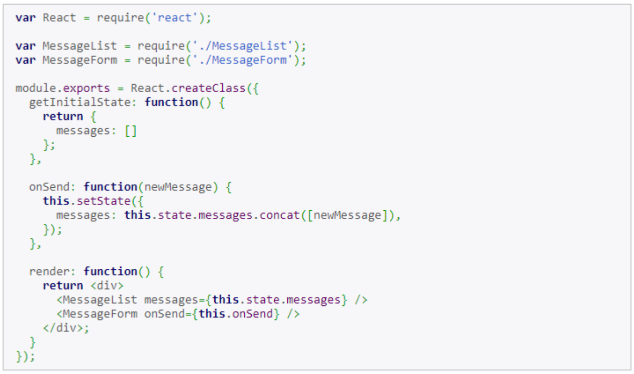
```
{
  key: 'ac3df4', // not with HashHistory!
  pathname: '/somewhere'
  search: '?some=search-string',
  hash: '#howdy',
  state: {
    [userDefined]: true
  }
}
```

The router will provide you with a location object in a few places:
•Route component as this.props.location
•Route render as ({ location }) => ()
•Route children as ({ location }) => ()
•withRouter as this.props.location
It is also found on history.location but you shouldn't use that because its mutable. You can read more about that in the history doc.

FORMS

Controller view

A controller view is a component which acts somewhat similar to controllers in MVC – they contain code to deal with the moving parts and data.



```
var React = require('react');

var MessageList = require('./MessageList');
var MessageForm = require('./MessageForm');

module.exports = React.createClass({
  getInitialState: function() {
    return {
      messages: []
    };
  },

  onSend: function(newMessage) {
    this.setState({
      messages: this.state.messages.concat([newMessage]),
    });
  },

  render: function() {
    return <div>
      <MessageList messages={this.state.messages} />
      <MessageForm onSend={this.onSend} />
    </div>;
  }
});
```

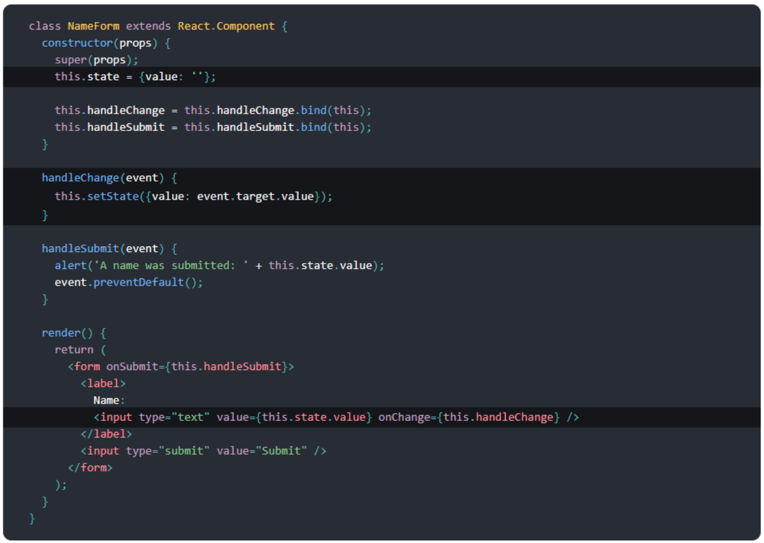
Controlled Component

- Any input with a value is a controlled component

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

Controlled Component



```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Building reusable inputs

We can define a reusable component, which can take in props that get passed down from the parent <form /> component. Three props we'll want to pass into the <Input /> component are:

name

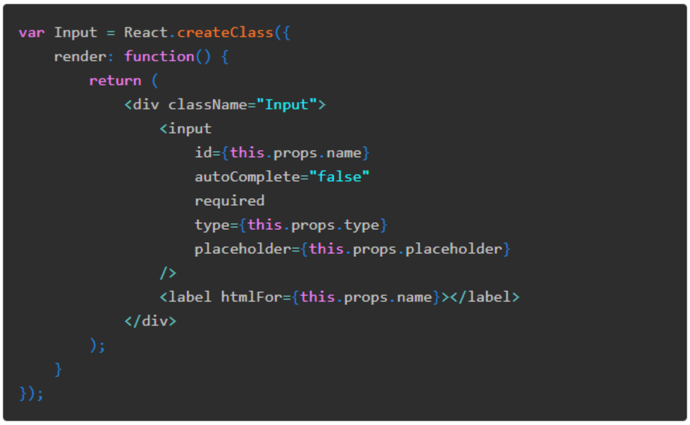
type

placeholder

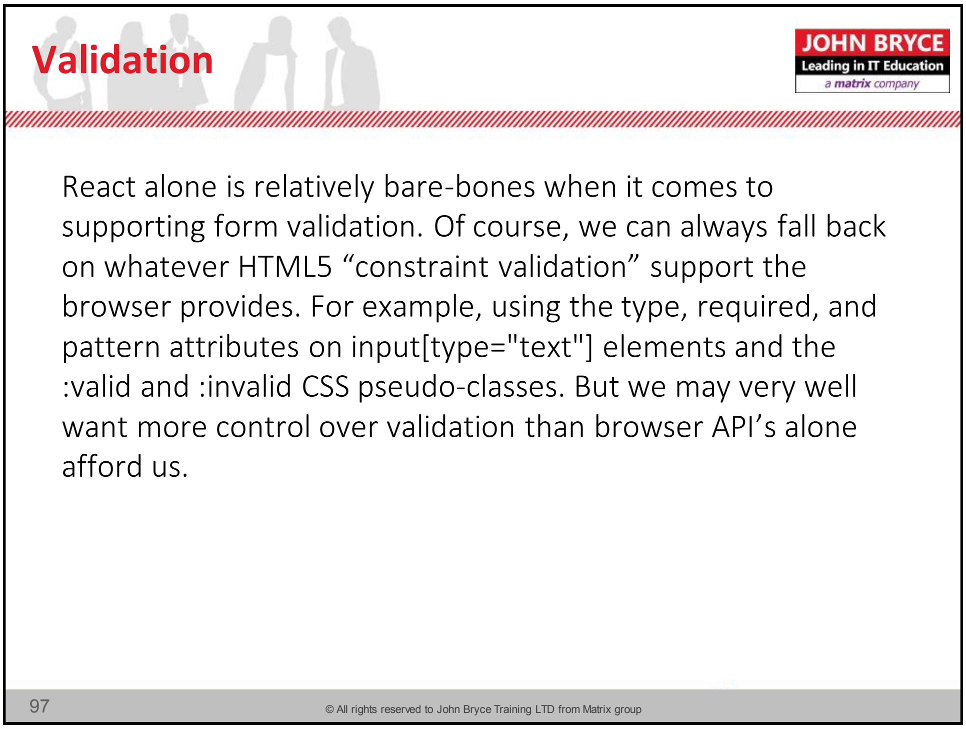
These particular values are JavaScript strings, unlike the onSubmit event handler where we passed a function.

These props allow us to create reusable components since we just have to pass in the type of input (either text, email or password), the name we want to associate with the input element, and the placeholder to a normal input element.

Building reusable inputs



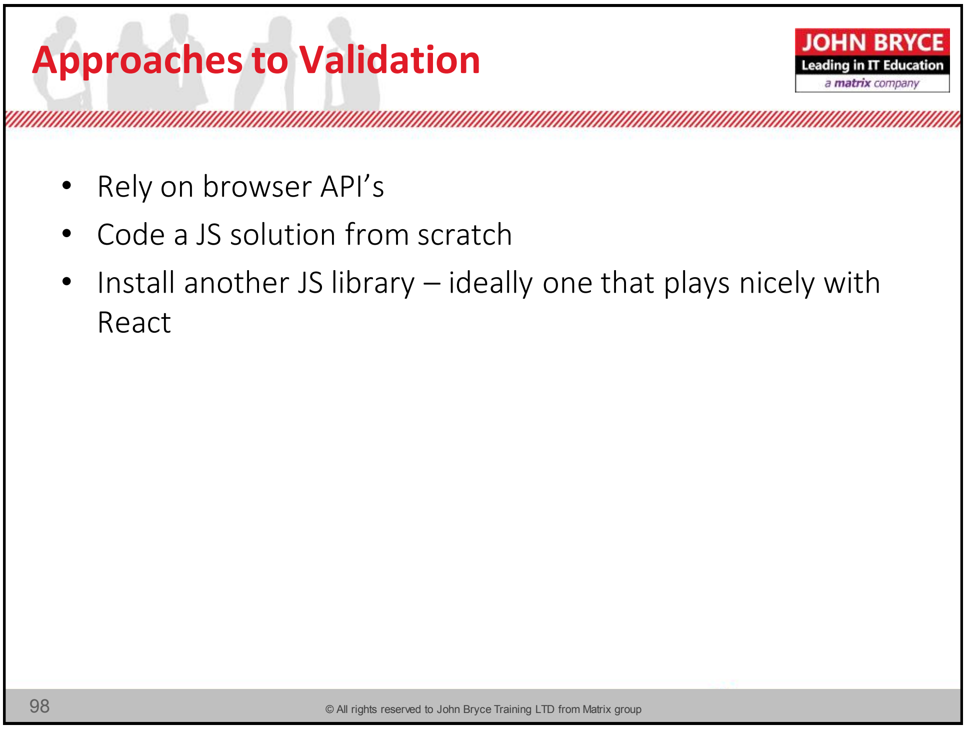
```
var Input = React.createClass({
    render: function() {
        return (
            <div className="Input">
                <input
                    id={this.props.name}
                    autoComplete="false"
                    required
                    type={this.props.type}
                    placeholder={this.props.placeholder}
                />
                <label htmlFor={this.props.name}></label>
            </div>
        );
    }
});
```



Validation

JOHN BRYCE
Leading in IT Education
a matrix company

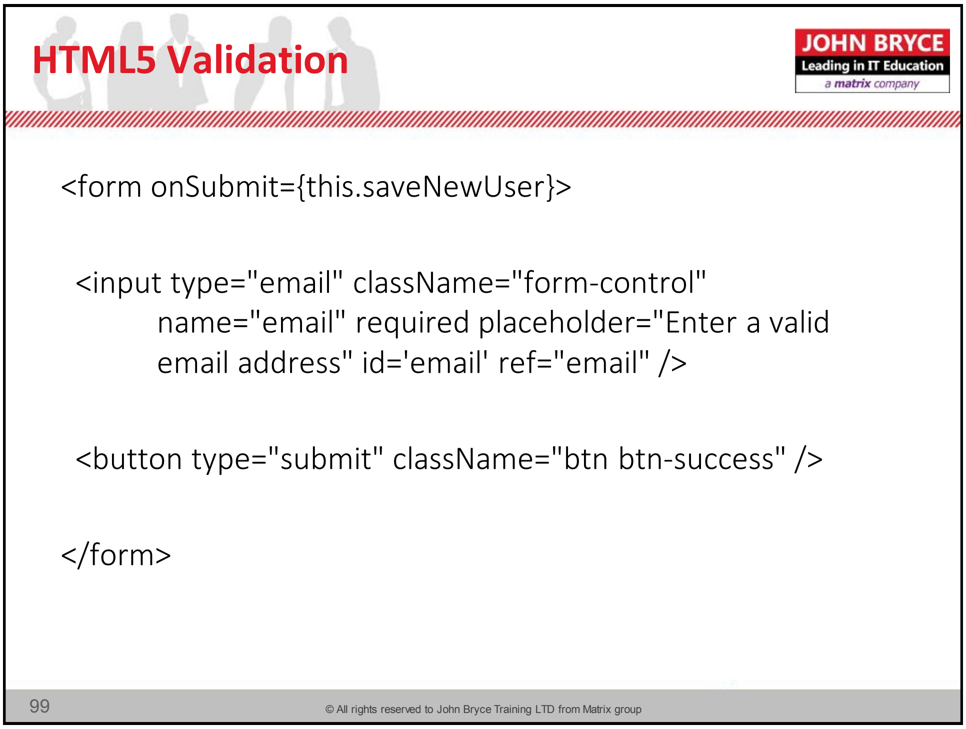
React alone is relatively bare-bones when it comes to supporting form validation. Of course, we can always fall back on whatever HTML5 “constraint validation” support the browser provides. For example, using the type, required, and pattern attributes on input[type="text"] elements and the :valid and :invalid CSS pseudo-classes. But we may very well want more control over validation than browser API’s alone afford us.



Approaches to Validation

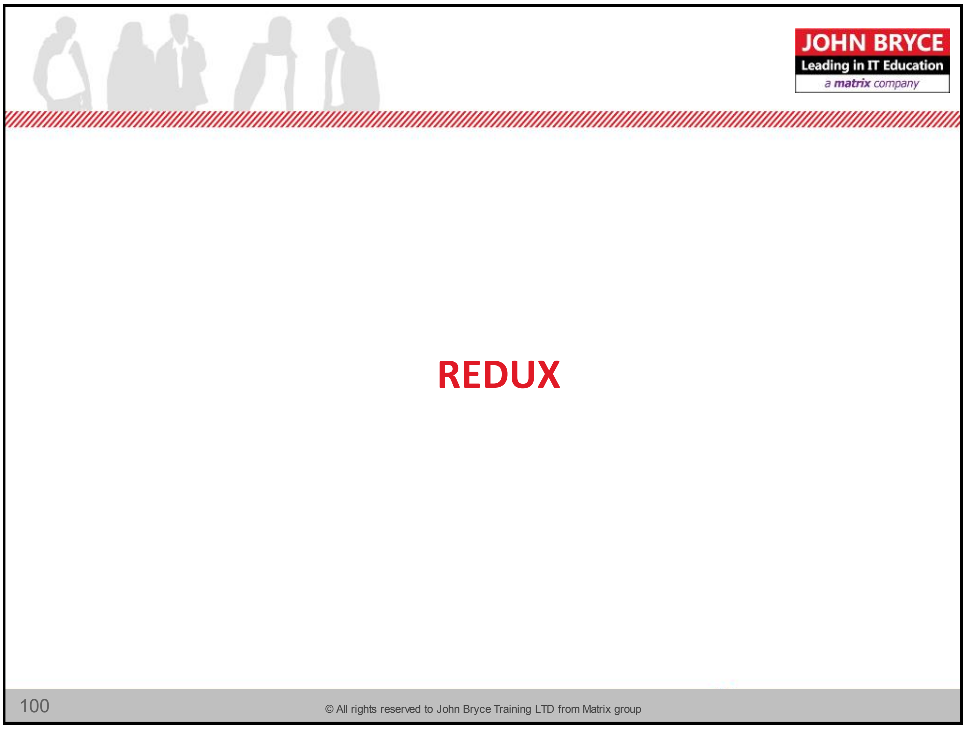
JOHN BRYCE
Leading in IT Education
a matrix company

- Rely on browser API’s
- Code a JS solution from scratch
- Install another JS library – ideally one that plays nicely with React

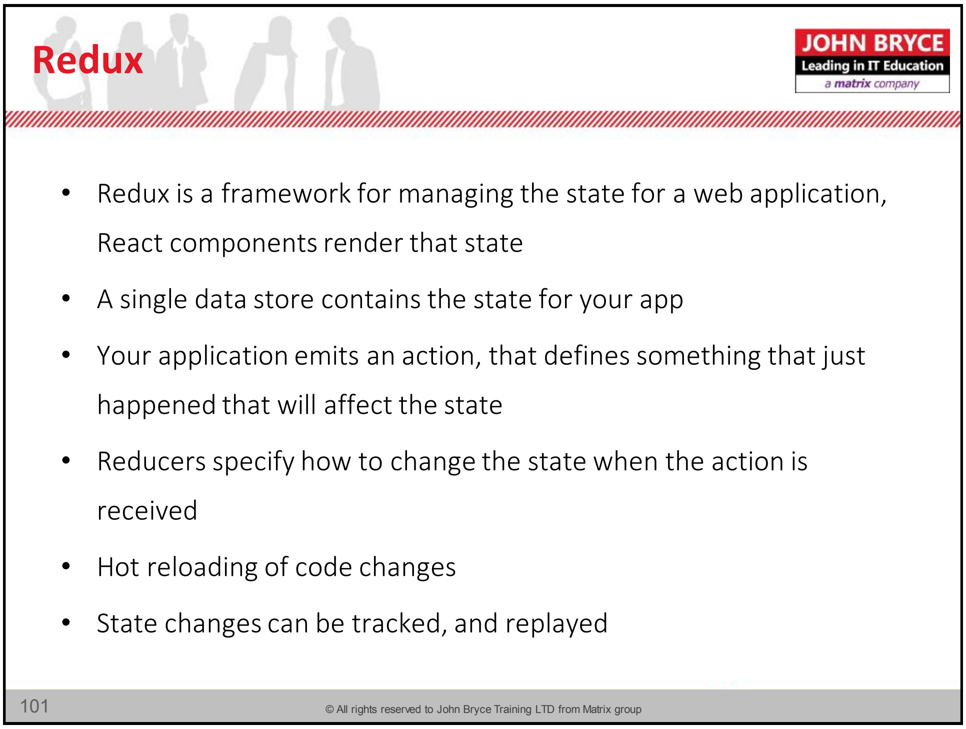


HTML5 Validation

```
<form onSubmit={this.saveNewUser}>

  <input type="email" className="form-control"
        name="email" required placeholder="Enter a valid
        email address" id='email' ref="email" />

  <button type="submit" className="btn btn-success" />

</form>
```



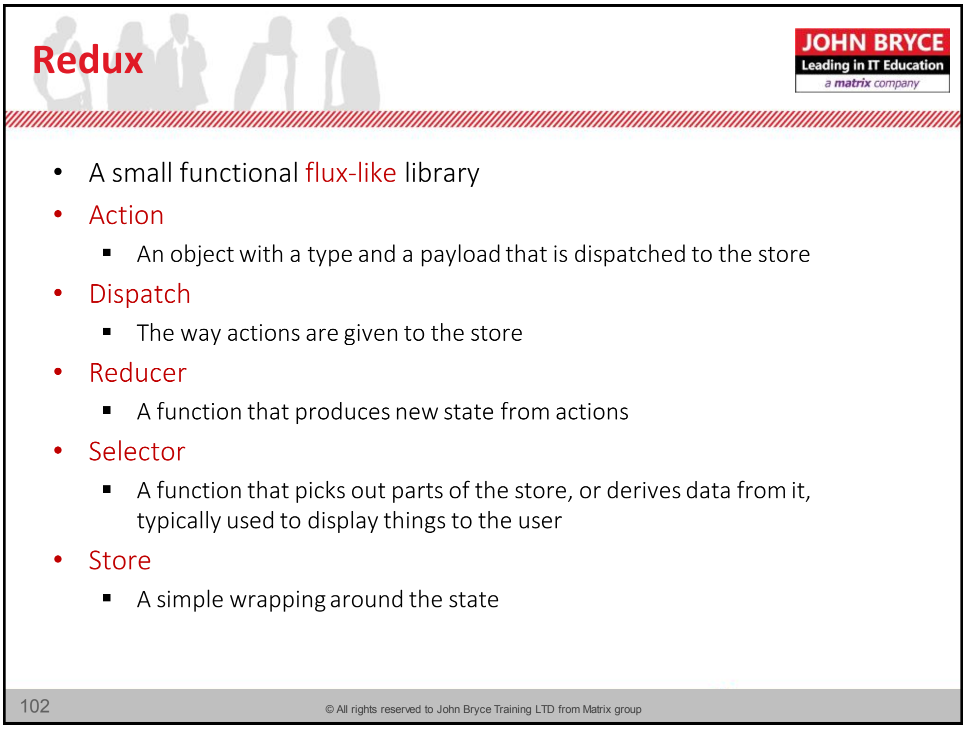
REDUX



Redux

JOHN BRYCE
Leading in IT Education
a matrix company

- Redux is a framework for managing the state for a web application, React components render that state
- A single data store contains the state for your app
- Your application emits an action, that defines something that just happened that will affect the state
- Reducers specify how to change the state when the action is received
- Hot reloading of code changes
- State changes can be tracked, and replayed



Redux

JOHN BRYCE
Leading in IT Education
a matrix company

- A small functional flux-like library
- Action
  - An object with a type and a payload that is dispatched to the store
- Dispatch
  - The way actions are given to the store
- Reducer
  - A function that produces new state from actions
- Selector
  - A function that picks out parts of the store, or derives data from it, typically used to display things to the user
- Store
  - A simple wrapping around the state

Code Organization

JOHN BRYCE
Leading in IT Education
a matrix company

- Components

  'Dumb' pieces of code that get told:

  which properties to use

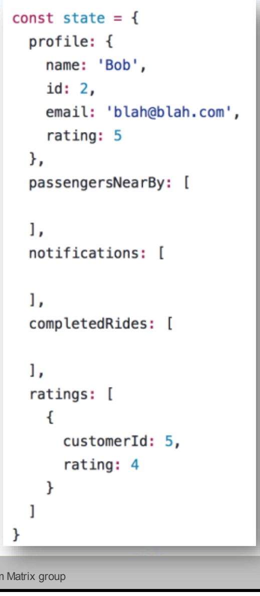
  which functions to call when something happens

- Containers
  - Where components get hooked up to bits of the state & dispatchers
- State/Store
  - The actions, reducers, selectors

Redux

JOHN BRYCE
Leading in IT Education
a matrix company

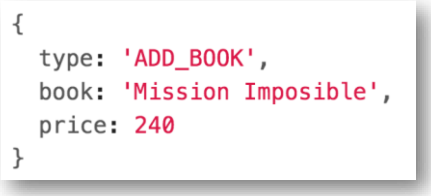
- Single store – a single JavaScript object to represent the application state (state tree)
- The state is an Array of objects



```
const state = {
  profile: {
    name: 'Bob',
    id: 2,
    email: 'blah@blah.com',
    rating: 5
  },
  passengersNearBy: [

  ],
  notifications: [

  ],
  completedRides: [

  ],
  ratings: [
    {
      customerId: 5,
      rating: 4
    }
  ]
}
```

Redux

- The state is read – only
- To change it we use Actions
  - Send data from UI to the Store
  - We get a new state object
- UI never interact with the state directly – only using actions



```
{
  type: 'ADD_BOOK',
  book: 'Mission Imposible',
  price: 240
}
```

Actions

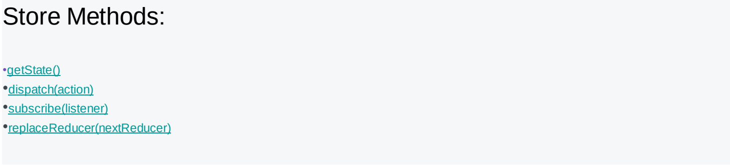
Actions are payloads of information that send data from your application to your store.

They are the only source of information for the store. You send them to the store using store.dispatch().

Store

A store holds the whole state tree of your application.
The only way to change the state inside it is to dispatch an action on it.

A store is not a class. It's just an object with a few methods on it.
To create it, pass your root reducing function to createStore.



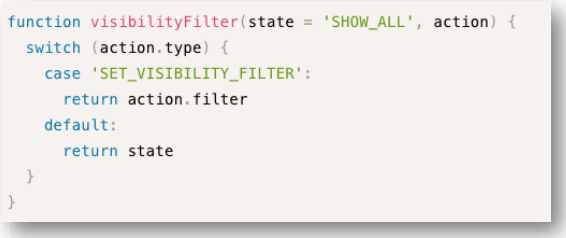
Store Methods:

- getState()
- dispatch(action)
- subscribe(listener)
- replaceReducer(nextReducer)

Pure functions

Changes are made with pure functions

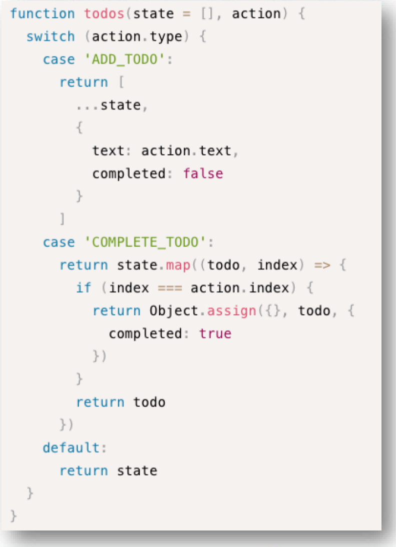
- Return a new state
- Simple implementation with no dependencies



```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}
```

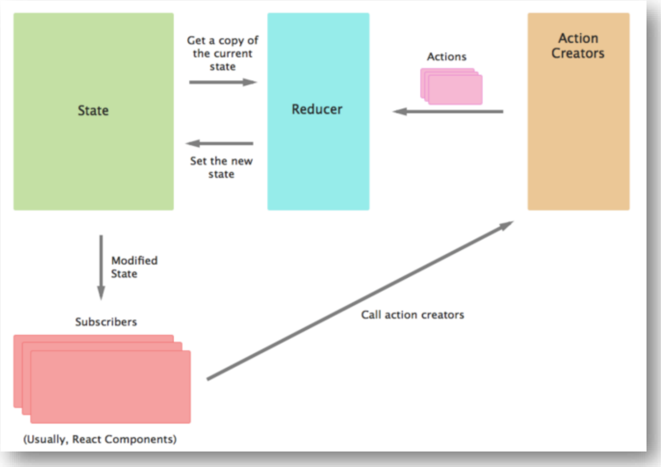
Reducer

- Takes current state and action and returns the next state

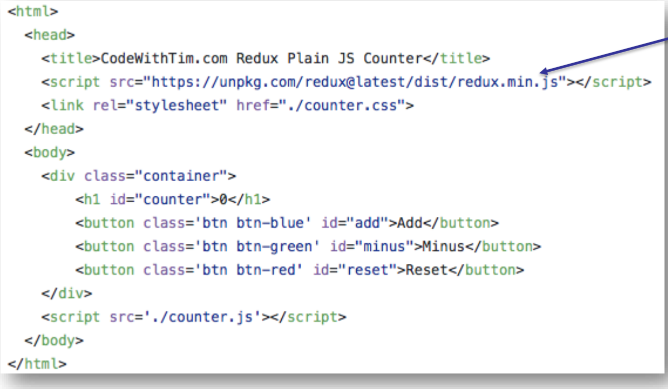


```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}
```

Redux



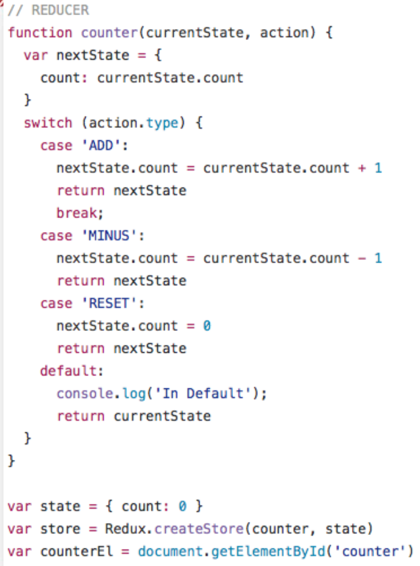
Plain JavaScript Example



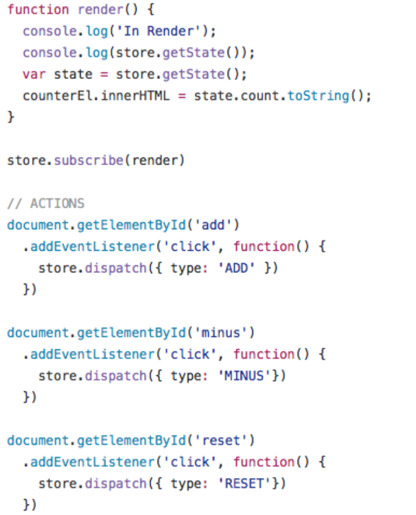
```
<html>
  <head>
    <title>CodeWithTim.com Redux Plain JS Counter</title>
    <script src="https://unpkg.com/redux@latest/dist/redux.min.js"></script>
    <link rel="stylesheet" href="./counter.css">
  </head>
  <body>
    <div class="container">
        <h1 id="counter">0</h1>
        <button class='btn btn-blue' id="add">Add</button>
        <button class='btn btn-green' id="minus">Minus</button>
        <button class='btn btn-red' id="reset">Reset</button>
    </div>
    <script src='./counter.js'></script>
  </body>
</html>
```

Redux library

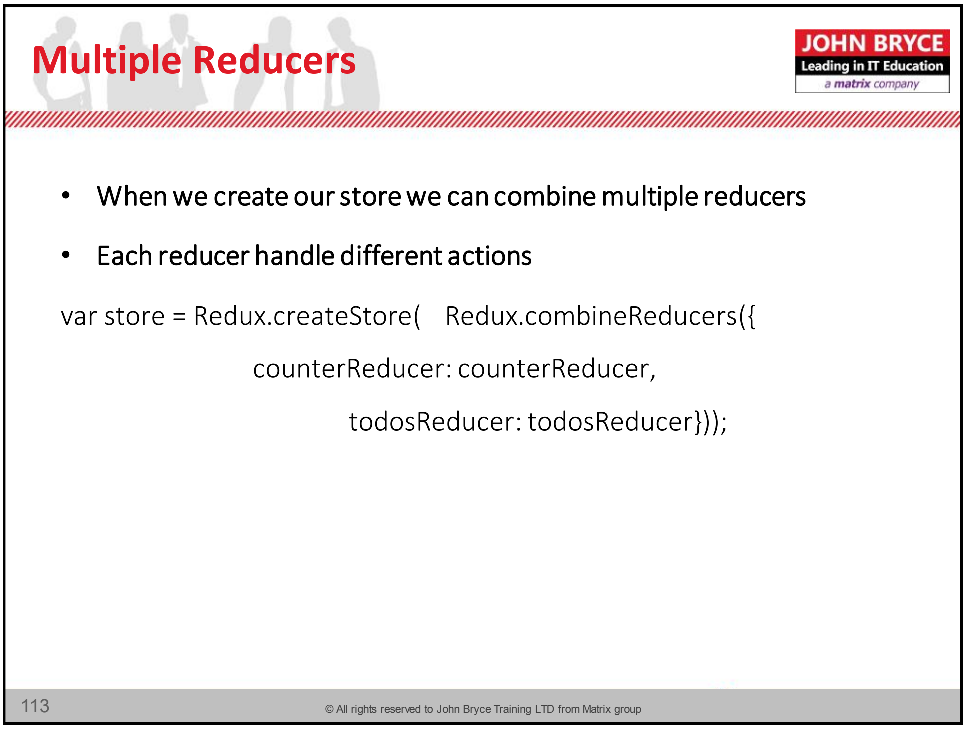
Source: https://github.com/codewithtim/Redux



```
// REDUCER
function counter(currentState, action) {
  var nextState = {
    count: currentState.count
  }
  switch (action.type) {
    case 'ADD':
      nextState.count = currentState.count + 1
      return nextState
      break;
    case 'MINUS':
      nextState.count = currentState.count - 1
      return nextState
    case 'RESET':
      nextState.count = 0
      return nextState
    default:
      console.log('In Default');
      return currentState
  }
}

var state = { count: 0 }
var store = Redux.createStore(counter, state)
var counterEl = document.getElementById('counter')
```



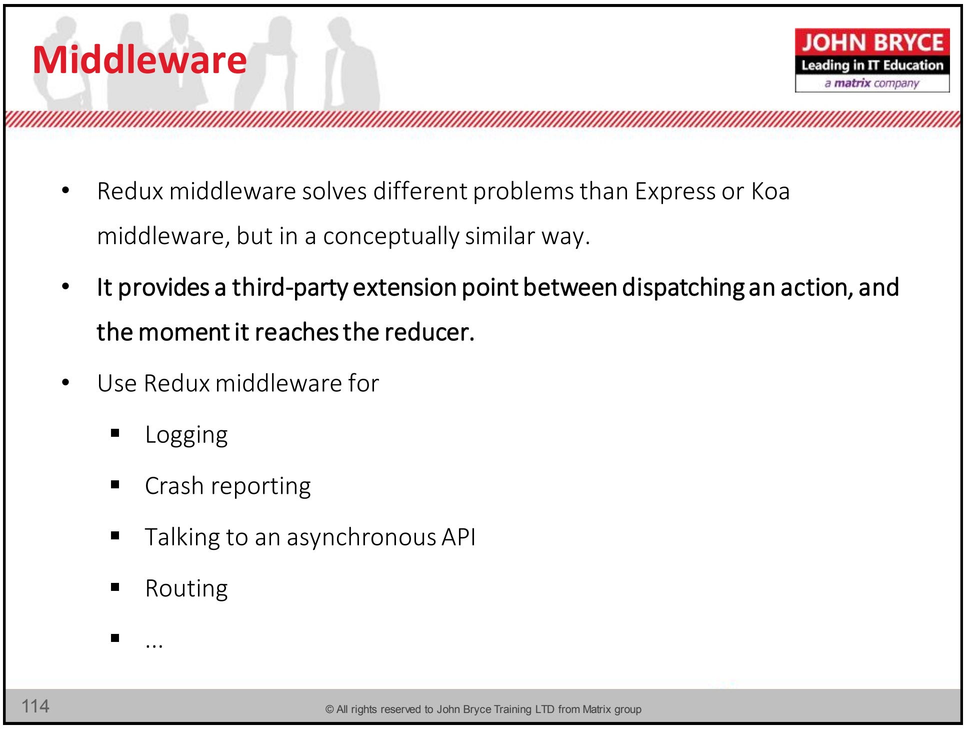
```
function render() {
  console.log('In Render');
  console.log(store.getState());
  var state = store.getState();
  counterEl.innerHTML = state.count.toString();
}

store.subscribe(render)

// ACTIONS
document.getElementById('add')
  .addEventListener('click', function() {
    store.dispatch({ type: 'ADD' })
  })

document.getElementById('minus')
  .addEventListener('click', function() {
    store.dispatch({ type: 'MINUS'})
  })

document.getElementById('reset')
  .addEventListener('click', function() {
    store.dispatch({ type: 'RESET'})
  })
```



Multiple Reducers

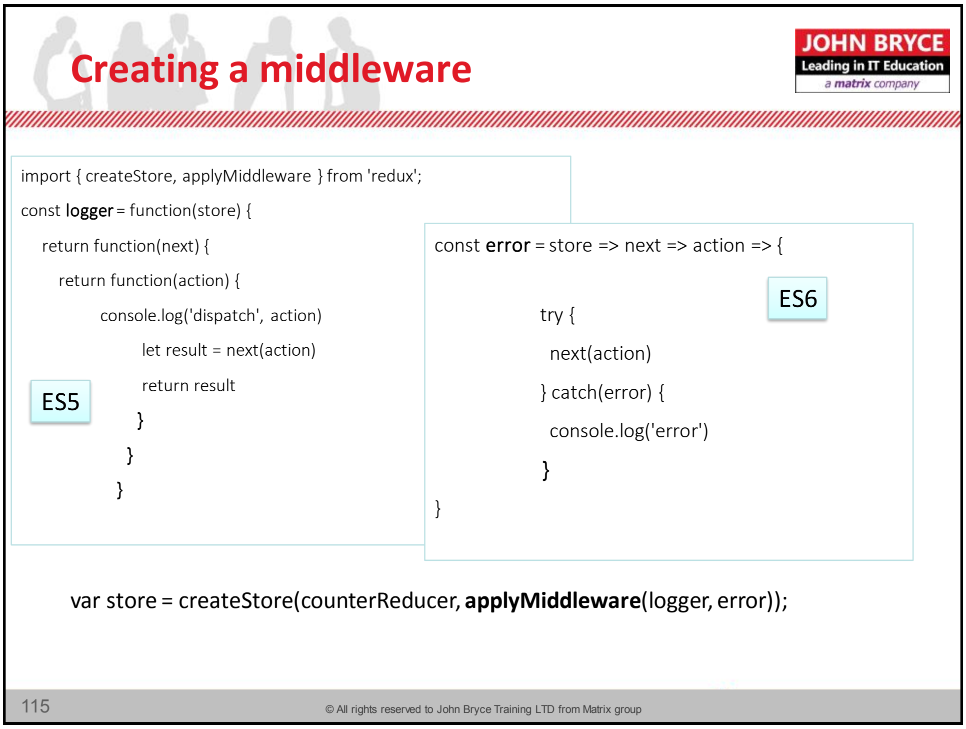
- When we create our store we can combine multiple reducers
- Each reducer handle different actions

```
var store = Redux.createStore(   Redux.combineReducers({
                    counterReducer: counterReducer,
                              todosReducer: todosReducer}));
```



Middleware

- Redux middleware solves different problems than Express or Koa middleware, but in a conceptually similar way.
- It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.
- Use Redux middleware for
  - Logging
  - Crash reporting
  - Talking to an asynchronous API
  - Routing
  - ...



Creating a middleware

JOHN BRYCE Leading in IT Education a matrix company

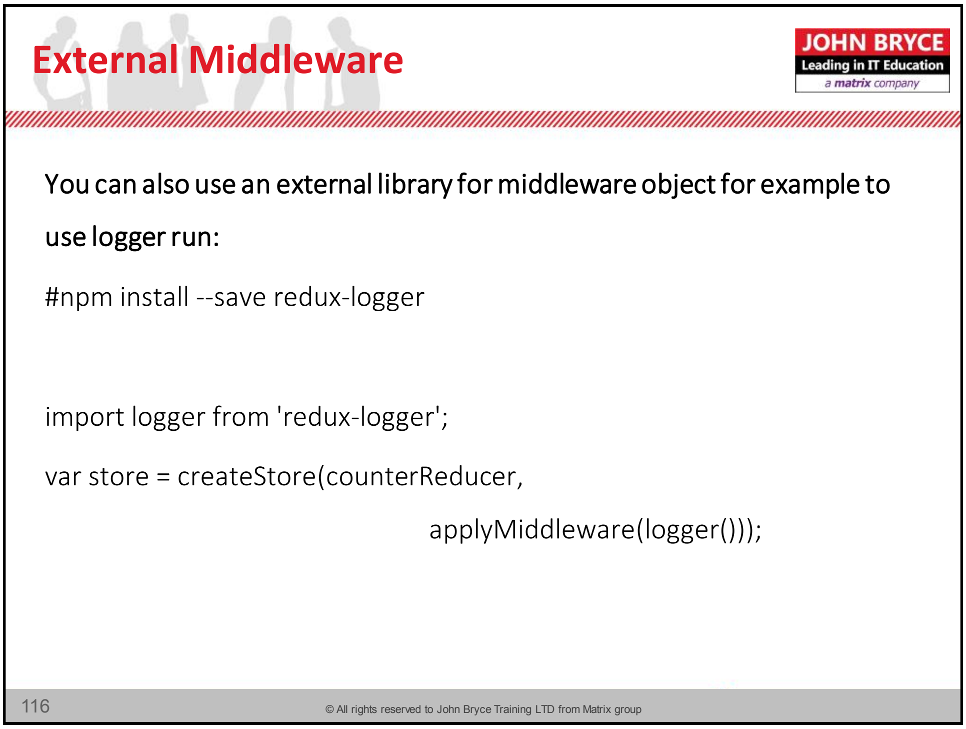
```
import { createStore, applyMiddleware } from 'redux';
const logger = function(store) {
    return function(next) {
      return function(action) {
            console.log('dispatch', action)
                  let result = next(action)
                  return result
                }
              }
            }
```

ES5

```
const error = store => next => action => {
              try {
                next(action)
              } catch(error) {
                console.log('error')
              }
}
```

ES6

```
var store = createStore(counterReducer, applyMiddleware(logger, error));
```



External Middleware

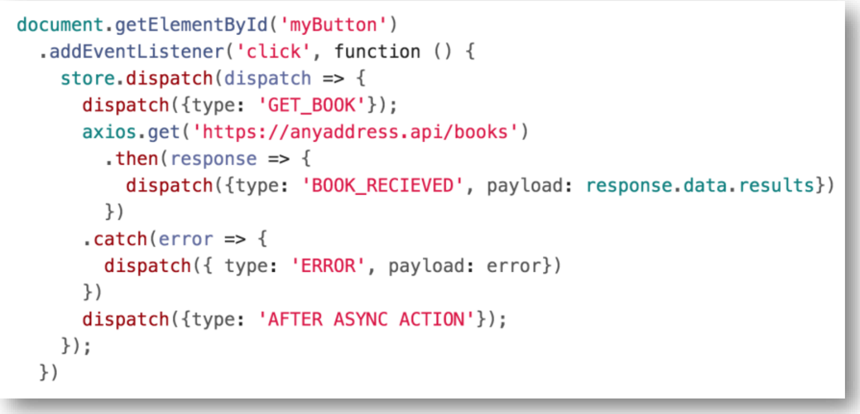
JOHN BRYCE Leading in IT Education a matrix company

You can also use an external library for middleware object for example to use logger run:

```
#npm install --save redux-logger
```

```
import logger from 'redux-logger';
var store = createStore(counterReducer,
                                        applyMiddleware(logger()));
```

Async Actions



```
document.getElementById('myButton')
  .addEventListener('click', function () {
    store.dispatch(dispatch => {
      dispatch({type: 'GET_BOOK'});
      axios.get('https://anyaddress.api/books')
        .then(response => {
          dispatch({type: 'BOOK_RECIEVED', payload: response.data.results})
        })
      .catch(error => {
        dispatch({ type: 'ERROR', payload: error})
      })
      dispatch({type: 'AFTER ASYNC ACTION'});
    });
  })
```

Using Promises

Install:

#npm install --save redux-promise-middleware

import promise from 'redux-promise-middleware';

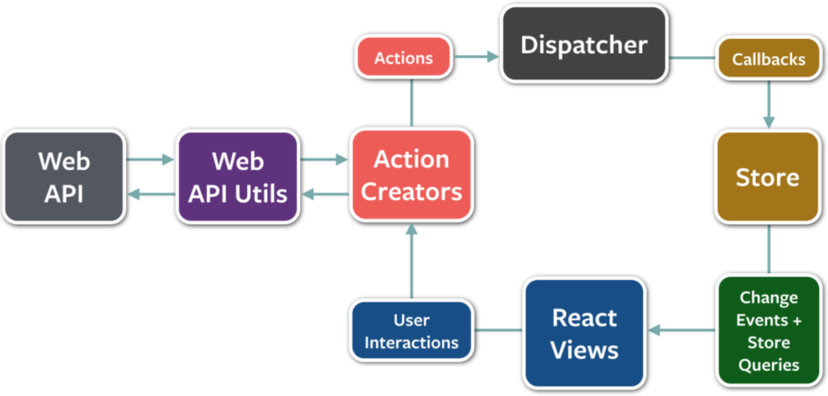
…

const store = createStore(userReducer,
                              applyMiddleware(logger(), promise()));

React + Redux

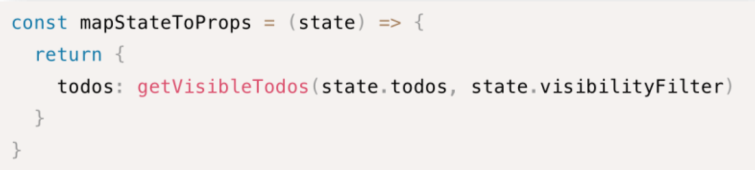
- The props for React components come from the Redux store that tracks the state.
- React components react to user input and emit actions, either directly or indirectly.
- Redux handles the action by running the appropriate reducers which transform the current state into a new state.
- React components react to the new state and update the DOM.
- React components themselves are stateless (most of the time), all of the state is kept in the Redux store, one common place, for simplicity.

React + Redux



mapStateToProp

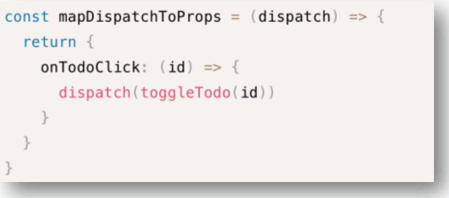
- mapStateToProps gets the Store state as an argument (by react-redux::connect) and its used to link the component with certain part of the store state.
- The object returned by mapStateToProps will be provided at construction time as props and any subsequent change will be available through componentWillReceiveProps.
- Observer design pattern



```
const mapStateToProps = (state) => {
  return {
    todos: getVisibleTodos(state.todos, state.visibilityFilter)
  }
}
```

mapDispatchToProps

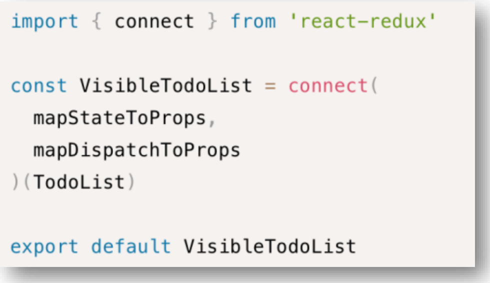
- In addition to reading the state, container components can dispatch actions.
- In a similar fashion, you can define a function called mapDispatchToProps() that receives the dispatch() method and returns callback props that you want to inject into the presentational component.



```
const mapDispatchToProps = (dispatch) => {
  return {
    onTodoClick: (id) => {
      dispatch(toggleTodo(id))
    }
  }
}
```

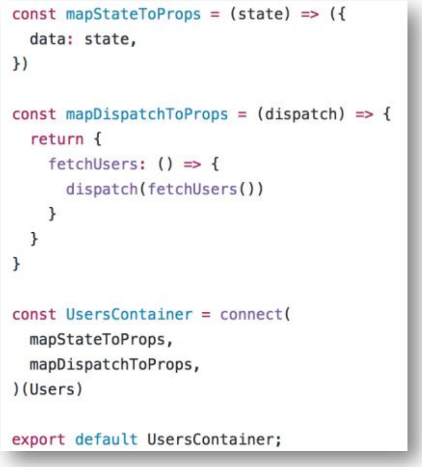
Connect

- Generates a container component
- Use the map* functions



```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

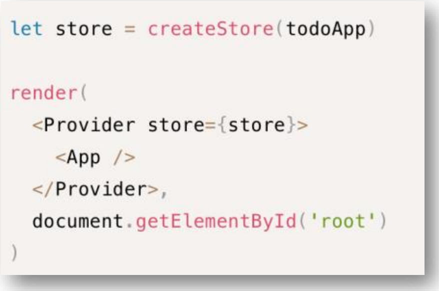
Container Example



```
const mapStateToProps = (state) => ({
  data: state,
})

const mapDispatchToProps = (dispatch) => {
  return {
    fetchUsers: () => {
      dispatch(fetchUsers())
    }
  }
}

const UsersContainer = connect(
  mapStateToProps,
  mapDispatchToProps,
)(Users)

export default UsersContainer;
```

Provider

Parent component that can be used to pass the store properties to its children components



```
let store = createStore(todoApp)

render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
)
```

Async Redux

Each of these two moments usually require a change in the application state; to do that, you need to dispatch normal actions that will be processed by reducers synchronously. Usually, for any API request you'll want to dispatch at least three different kinds of actions:

An action informing the reducers that the request began.

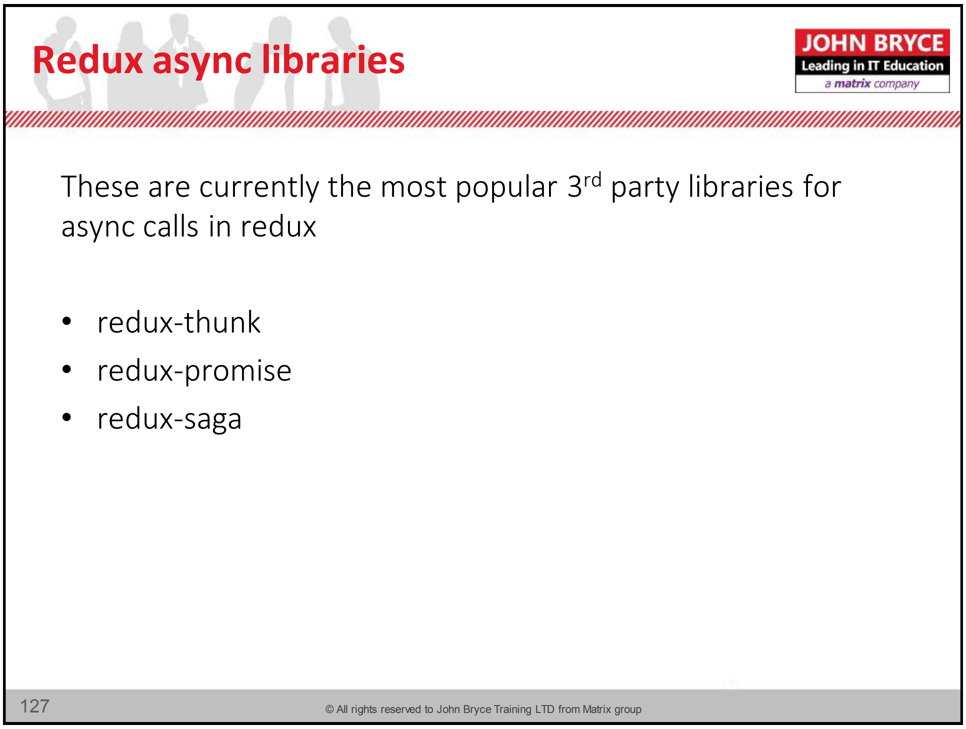
The reducers may handle this action by toggling an isFetching flag in the state. This way the UI knows it's time to show a spinner.

An action informing the reducers that the request finished successfully.

The reducers may handle this action by merging the new data into the state they manage and resetting isFetching. The UI would hide the spinner, and display the fetched data.

An action informing the reducers that the request failed.

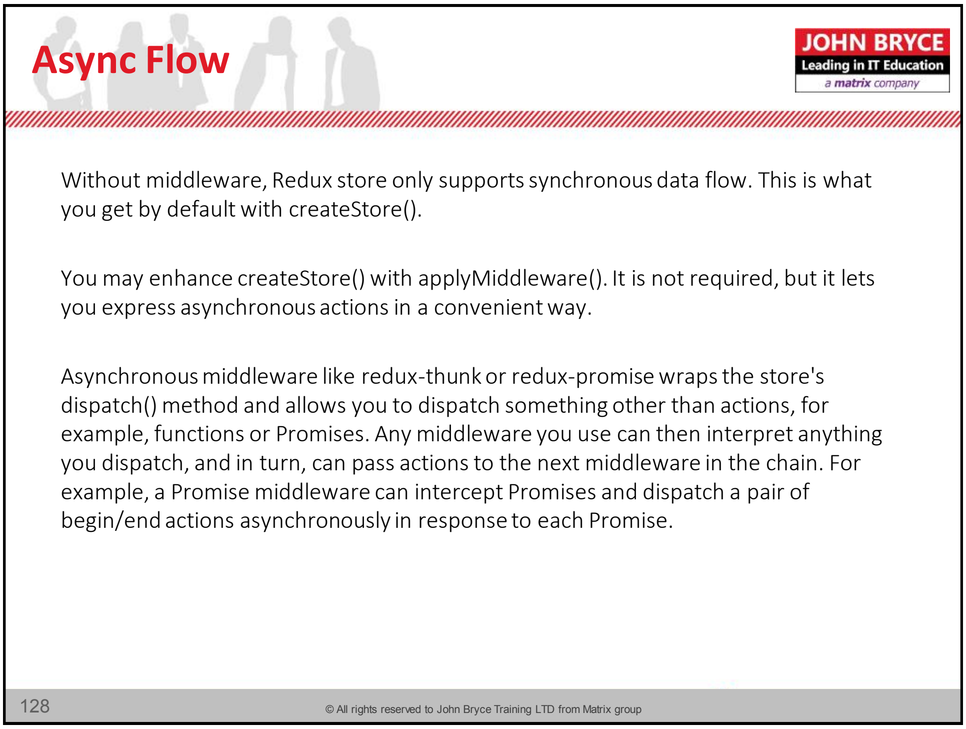
The reducers may handle this action by resetting isFetching. Additionally, some reducers may want to store the error message so the UI can display it.



Redux async libraries

These are currently the most popular 3rd party libraries for async calls in redux

- redux-thunk
- redux-promise
- redux-saga

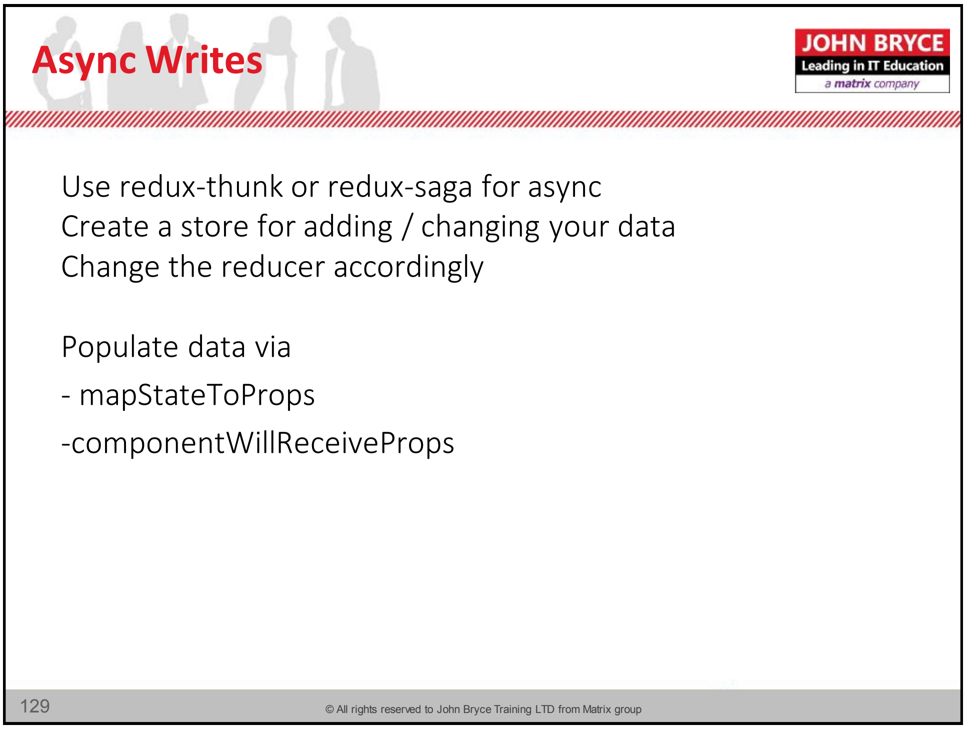


Async Flow

Without middleware, Redux store only supports synchronous data flow. This is what you get by default with createStore().

You may enhance createStore() with applyMiddleware(). It is not required, but it lets you express asynchronous actions in a convenient way.

Asynchronous middleware like redux-thunk or redux-promise wraps the store's dispatch() method and allows you to dispatch something other than actions, for example, functions or Promises. Any middleware you use can then interpret anything you dispatch, and in turn, can pass actions to the next middleware in the chain. For example, a Promise middleware can intercept Promises and dispatch a pair of begin/end actions asynchronously in response to each Promise.



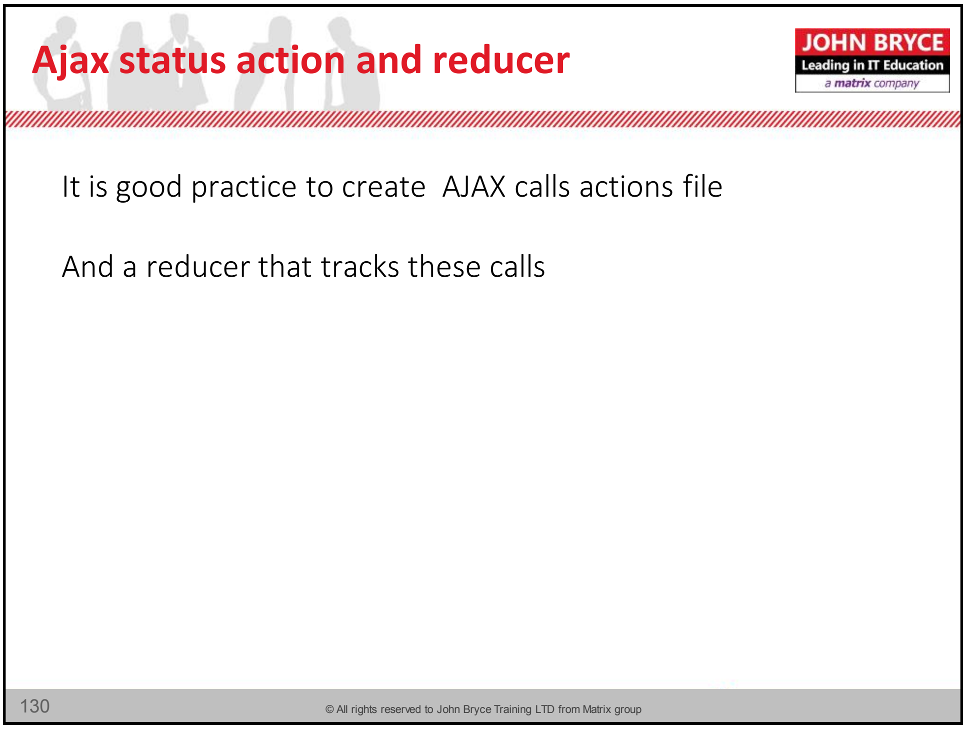
Async Writes

Use redux-thunk or redux-saga for async
Create a store for adding / changing your data
Change the reducer accordingly

Populate data via

- mapStateToProps

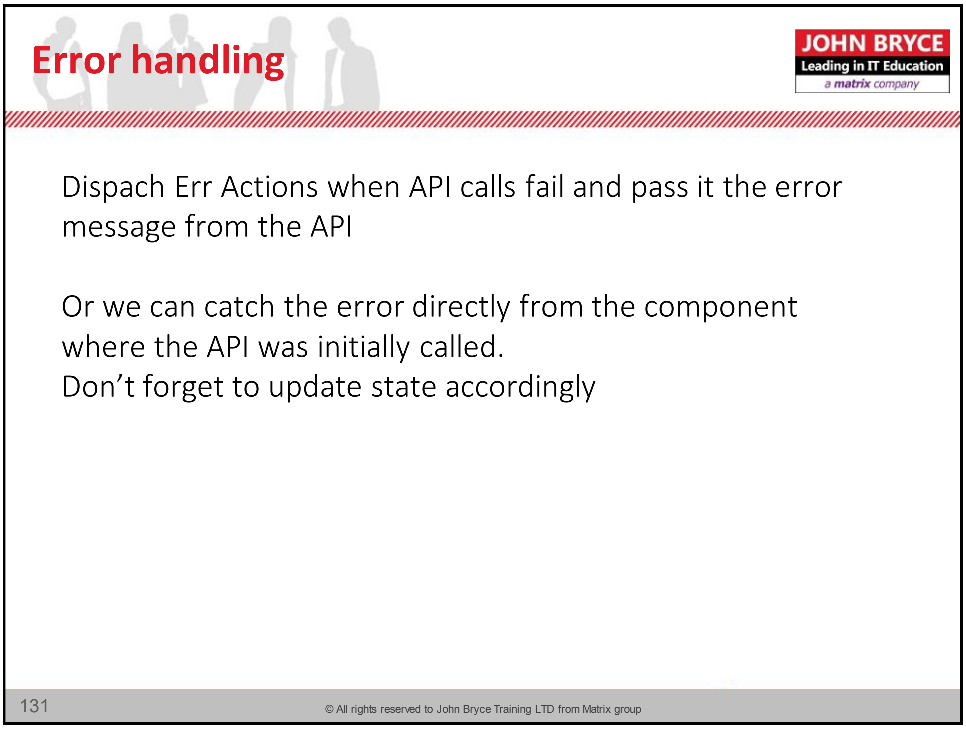
-componentWillReceiveProps



Ajax status action and reducer

It is good practice to create AJAX calls actions file

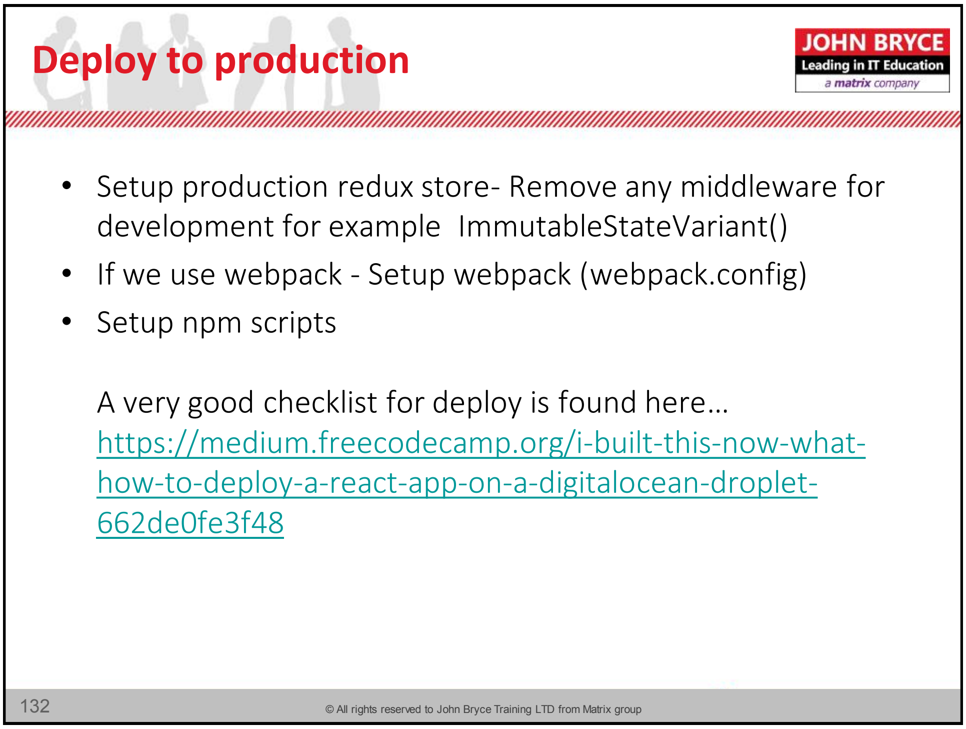
And a reducer that tracks these calls



Error handling

Dispach Err Actions when API calls fail and pass it the error message from the API

Or we can catch the error directly from the component where the API was initially called.
Don’t forget to update state accordingly



Deploy to production

- Setup production redux store- Remove any middleware for development for example ImmutableStateVariant()
- If we use webpack - Setup webpack (webpack.config)
- Setup npm scripts

A very good checklist for deploy is found here…
https://medium.freecodecamp.org/i-built-this-now-what-how-to-deploy-a-react-app-on-a-digitalocean-droplet-662de0fe3f48

JOHN BRYCE
Leading in IT Education
a matrix company

Thank You!

133