ReactJS

React (a.k.a. ReactJS or React.js) is a JavaScript library for creating user interfaces, open sourced to the world by Facebook and Instagram team in 2013. One might think of it as the "View" in the "Model-View-Controller" pattern. React's main goal is to make development of UI components easy and modular. It is intended to ease the process of building large applications using data that changes over time.

React was created by Jordan Walke, a software engineer at Facebook, with the influence of XHP, a PHP-based component system that is still in use at Facebook, but also by functional programming ideas.

Pete Hunt wanted to use React at Instagram, so he pushed to extract React from Facebook-specific code and open source it. React has gained a lot of popularity for its concept of a "virtual-DOM," which allows it to determine which parts of the DOM have changed by diffing the new version with the stored virtual DOM, and using the result to determine how to most efficiently update the browser's DOM.

Motivation for Using React

React is not a framework, but a library. It means that you don't have to fit your ideas into "their" structure, but rather use React whenever and wherever you like. React doesn't pretend to be a "golden hammer". It only does one thing - rendering the view. And does it well. React scales very well due to its declarativeness. It's very much unlike jQuery, where you have to manually update dependencies. React's components re-render itself when needed. React is really fast.

React is front end library developed by Facebook. It's used for handling view layer for web and mobile apps. ReactJS allows us to create reusable UI components. It is currently one of the most popular JavaScript libraries and it has strong foundation and large community behind it.

It encourages the creation of reusable UI components which present data that changes over time. React abstracts away the DOM from you, giving a simpler programming model and better performance. React can also render on the server using Node, and it can power native apps using React Native. React implements one-way reactive data flow which reduces boilerplate and is easier to reason about than traditional data binding.

React Features

- **JSX –** JSX is JavaScript syntax extension. It isn't necessary to use JSX in React development, but it is recommended.
- **Components –** React is all about components. You need to think of everything as a component. This will help you to maintain the code when working on larger scale projects.

- **Unidirectional data flow and Flux –** React implements one way data flow which makes it easy to reason about your app. Flux is a pattern that helps keeping your data unidirectional.
- **License –** React is licensed under the Facebook Inc. Documentation is licensed under CC BY 4.0.

React Advantages

- React uses virtual DOM which is JavaScript object. This will improve apps performance since JavaScript virtual DOM is faster than the regular DOM.
- React can be used on client and server side.
- Component and Data patterns improve readability which helps to maintain larger apps.
- React can be used with other frameworks.

React Limitations

- React only covers view layer of the app so you still need to choose other technologies to get a complete tooling set for development.
- React is using inline templating and JSX. This can seem awkward to some developers.

Who Uses React.js?

- **Instagram.com** is 100% built on React, both public site and internal tools.
- **Facebook.com's** commenting interface, business management tools, Lookback video editor, page insights, and most, if not all, new JS development.
- **Khan Academy** uses React for most new JS development.
- **Sberbank**, Russia's number one bank, is built with React.
- **The New York Times's** 2014 Red Carpet Project is built with React.

React as an Alternative Architecture

React does have an innovative architecture that can be used to build much more complex applications. The React core is a system for mapping a view hierarchy onto some sort of rendering back end and most often it targets the browser DOM. The React team has stated they are less concerned about the JavaScript reference implementation and instead focused on the architecture of React. In other words, they care about the alternative ideas they are putting out there for building client-side applications more than their specific implementation of them in React.js.

The innovation React brings is in three main areas:

1. **An alternative to event and data binding framework architectures.**

The React alternative is programming like you are throwing away your entire component and re-rendering it every time because it is simpler and easier to reason about. Event-based and data binding approaches frequently run into timing problems keeping track of which callback gets called first as well as makes it difficult to understand how a small change in state will affect performance.

2. Virtual DOM

React generates a virtual description of the DOM that we want and then diffs that description with a previous one and emits the changes. This can be done efficiently and programming becomes simpler. We don't think about synchronizing pieces of data with bindings and don't think about events firing. You just have a minimal set of state and paint the browser DOM with it. The performance improvements they have been able to achieve with this approach has definitely gotten other frameworks to take notice.

3. Components

If you break down your application into smaller components (main menu, footer, list, list item, datepicker, etc.) then it becomes easier to imagine building a complex application and it remaining maintainable. This is the promise of component-based architectures. Essentially it is jQuery plugins but standardized and better encapsulated so an Ember component, works with an Angular component, works with a React component and comes bundled with HTML, CSS, and JavaScript and none of it conflicts with other components on your page. This is the dream.

React doesn't quite deliver on the HTML, CSS, and JavaScript encapsulation story like Polymer, but React does have the huge advantage of not being a native browser feature. This abstraction allows it to be a technology that can be trusted today in the browser and on the server to have great performance and is not waiting for the browsers to catch up.

Components a better way to build an application

React scales better with complexity. Components with encapsulated behavior is how they are able to keep the complexity lower by only looking at smaller pieces. One particularly impressive feature of react is how components can be composed and reused inside other components by simply using the markup for the custom component.

For example, CommentList and CommentForm are re-used inside CommentBox by simply writing the custom element tags.

```
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList />
           <CommentForm />
```

```
        </div>
      );
    }
});
```

Key differentiators

Virtual DOM

React uses a concept called the Virtual DOM that 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.

Imagine you had an object that you modeled around a person. It had every relevant property a person could possibly have, and mirrored the persons current state. This is basically what React does with the DOM.

Now think about if you took that object and made some changes. Added a mustache, some sweet biceps and Steve Buscemi eyes. In React-land, when we apply these changes, two things take place. First, React runs a "diffing" algorithm, which identifies what has changed. The second step is reconciliation, where it updates the DOM with the results of diff.

The way React works, rather than taking the real person and rebuilding them from the ground up, it would only change the face and the arms. This means that if you had text in an input and a render took place, as long as the input's parent node wasn't scheduled for reconciliation, the text would stay undisturbed.

Because React is using a fake DOM and not a real one, it also opens up a fun new possibility. We can render that fake DOM on the server, and boom, server side React views.

One way binding

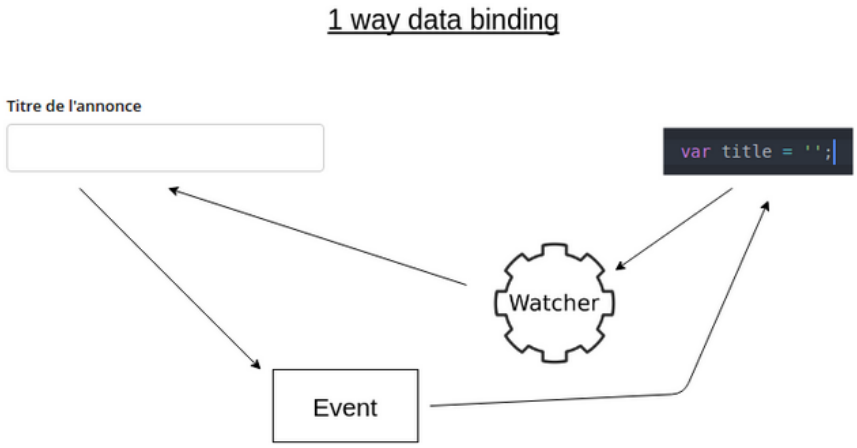
React doesn't have a mechanism to allow the HTML to change the component. The HTML can only raise events that the component responds to. The typical example is by using **onChange.**

```
//js
render() {
   return <input value={this.state.value} onChange={this.handleChange} />
}
handleChange(e) {
   this.setState({value: e.target.value});
}
```

The value of the <input /> is controlled entirely by the render function. The only way to update this value is from the component itself, which is done by attaching an **onChange** event to the

<input /> which sets **this.state.value** to with the React component method **setState**. The <input /> does not have direct access to the components state, and so it cannot make changes. This is one-way binding.

one way binding only binds the value of the model to the view and does not have an additional watcher to determine if the value in the view has been changed by the user.



Example:

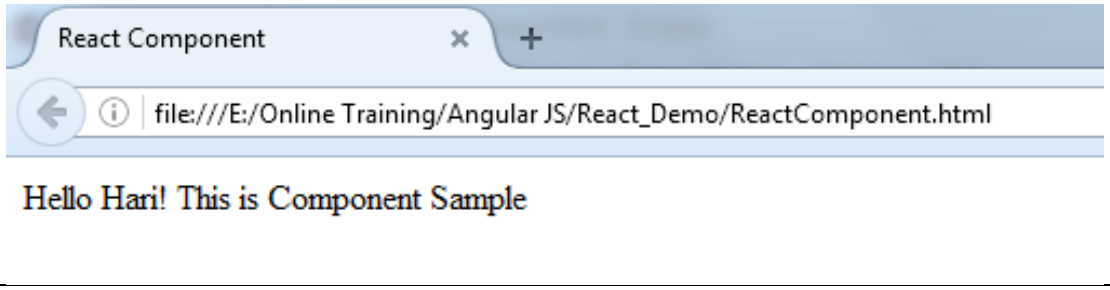
```
class SimpleComponent extends React.Component {
  constructor(props, ...args) {
    super(props, ...args);
    this.handleChange = this.handleChange.bind(this);
    this.state = {value: this.props.value};
  }
  handleChange(e) {
    this.setState({value: e.target.value});
  }
  render() {
    return (
      <div>
        <p>Value: {this.state.value}</p>
        <input value={this.state.value}
            onChange={this.handleChange} />
      </div>
      );
  }
}

React.render(<SimpleComponent value={'end'} />, document.body);
```

React Components

1. **React component**

Components are the heart and soul of React. React components are very simple. You can think of them as simple functions that take in props and state (discussed later) and render HTML. With this in mind, components are easy to reason about. Create a new component class using React.createClass. Components have one requirement; they must implement render, a function that tells the component what to... render.

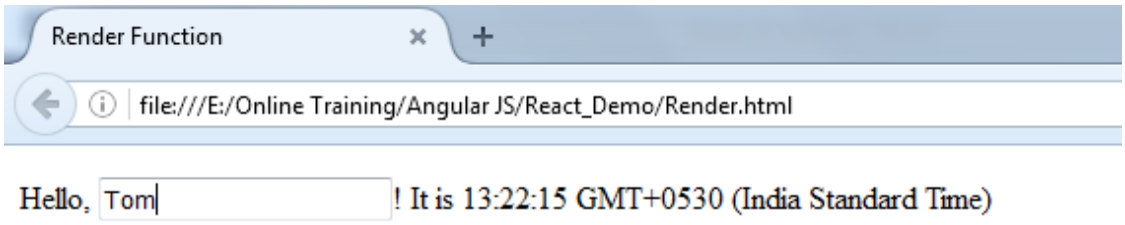
```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Component</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
<div id="container"></div>
<script type="text/jsx">
// Create a component named MessageComponent
var MessageComponent = React.createClass({
  render: function() {
    return (
      <div>{this.props.message}</div>
    );
  }
});

// Render an instance of MessageComponent into document.body
ReactDOM.render(
  <MessageComponent message="Hello Hari! This is Component Sample" />,
  document.body
);
</script>
</body>
</html>
```



2. Render function

The render() function should be pure, meaning that it does not modify component state, it returns the same result each time it's invoked, and it does not read from or write to the DOM or otherwise interact with the browser. Keeping render() pure makes server rendering more practical and makes components easier to think about.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Render Function</title>
        <script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>
    <script src="https://npmcdn.com/react-dom@15.3.0/dist/react-
dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
</head>
<body>
       <div id="container"></div>
       <script type="text/babel">
var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>
        Hello, <input type="text" placeholder="Enter your Name" />!
        It is {this.props.date.toTimeString()}
      </p>
    );
  }
});

setInterval(function() {
  ReactDOM.render(
    <HelloWorld date={new Date()} />,
    document.getElementById('container')
  );
}, 500);
</script>
</body>
</html>
```



Open Render.html in a web browser and type your name into the text field. Notice that React is only changing the time string in the UI — any input you put in the text field remains, even though you haven't written any code to manage this behavior. React figures it out for you and does the right thing.

The way we are able to figure this out is that React does not manipulate the DOM unless it needs to. It uses a fast, internal mock DOM to perform diffs and computes the most efficient DOM mutation for you.

The inputs to this component are called props — short for "properties". They're passed as attributes in JSX syntax. You should think of these as immutable within the component, that is, never write to **this.props.**

3. Component API

Instances of a React Component are created internally in React when rendering. These instances are reused in subsequent renders, and can be accessed in your component methods as this. The only way to get a handle to a React Component instance outside of React is by storing the return value of ReactDOM.render. Inside other Components, you may use refs to achieve the same result.

a) Set State

setState() method is used for updating the state of the component. This method will not replace the state but only add changes to original state.
API

Get Initial State
Implement the function getInitialState, which returns... the initial state of the component. This is an object map of keys to values.

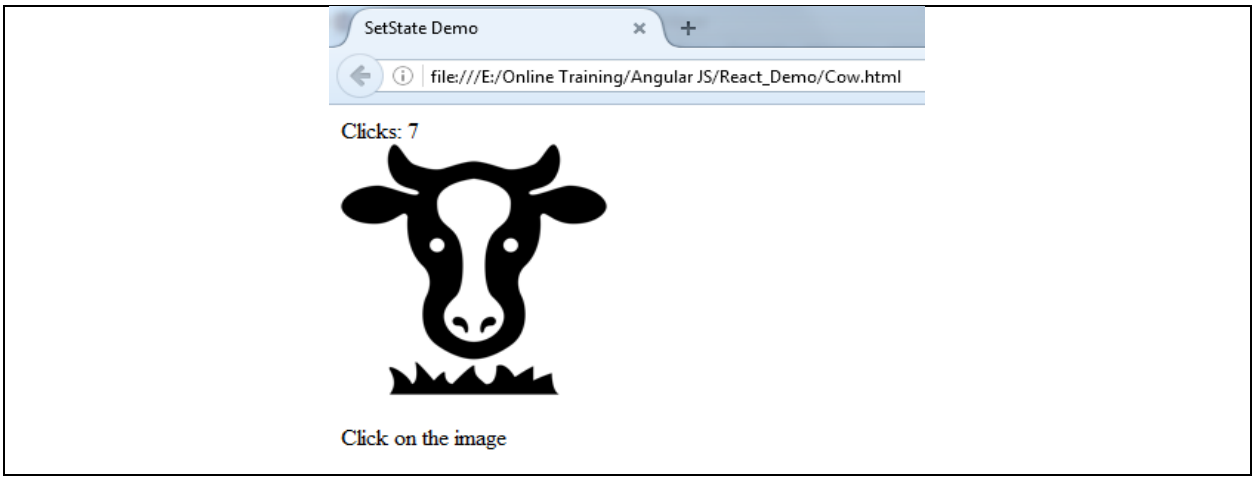
```
getInitialState: function() {
  return {
    clicks: 0
  };
}
```

This.State
To access a component's state, use this.state,

This.Setstate
To update a component's state, call this.setState with an object map of keys to updated values. Keys that are not provided are not affected.

When a component's state changes, render is called with the new state and the UI is updated to the new output. This is the heart of React.

```
<!DOCTYPE html>
<html>
<head>
             <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
       <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
       <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>SetState Demo</title>
</head>
<body>
  <div id="container"></div>
  <script type="text/jsx">
var CowClicker = React.createClass({
  getInitialState: function() {
    return {
      clicks: 0
    };
  },
onCowClick: function(evt) {
    this.setState({
      clicks: this.state.clicks + 1
    });
  },
render: function() {
    return (
      <div>
        <div>Clicks: {this.state.clicks}</div>
        <img src="http://s3.bypaulshen.com/buildwithreact/cow.png"
          onClick={this.onCowClick}
          className="cow"
        />
        <p>Click on the image</p>
      </div>
    );
  }
});
ReactDOM.render(<CowClicker />,document.getElementById('container'));
</script>
</body>
</html>
```



b) **replaceState**

With setState the current and previous states are merged. With replaceState, it throws out the current state, and replaces it with only what you provide. Usually setState is used unless you really need to remove keys for some reason; but setting them to false/null is usually a more explicit tactic.

While it's possible it could change; replaceState currently uses the object passed as the state, i.e. replaceState(x), and once it's set this.state === x. This is a little lighter than setState, so it could be used as an optimization if thousands of components are setting their states frequently.

If your current state is {a: 1}, and you call this.setState({b: 2}); when the state is applied, it will be {a: 1, b: 2}. If you called this.replaceState({b: 2}) your state would be {b: 2}.

Example:

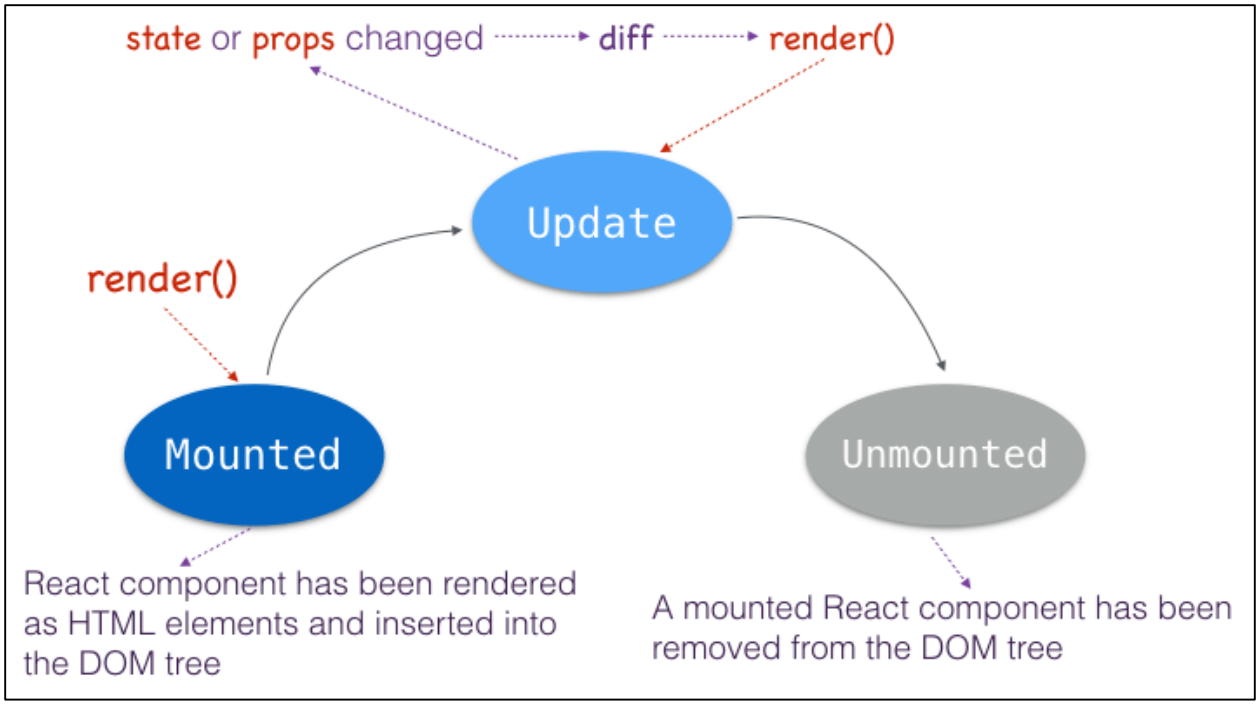
```
// let's say that this.state is {foo: 98}
        this.setState ({bar: 107})
// this.state is now {foo: 98, bar: 107}
        this.setState ({foo: 99})
// this.state is now {foo: 99, bar: 107}
        this.replaceState ({name: "tom"})
// this.state. is now {name: "tom"}
```

4. Component Specs & Life Cycle

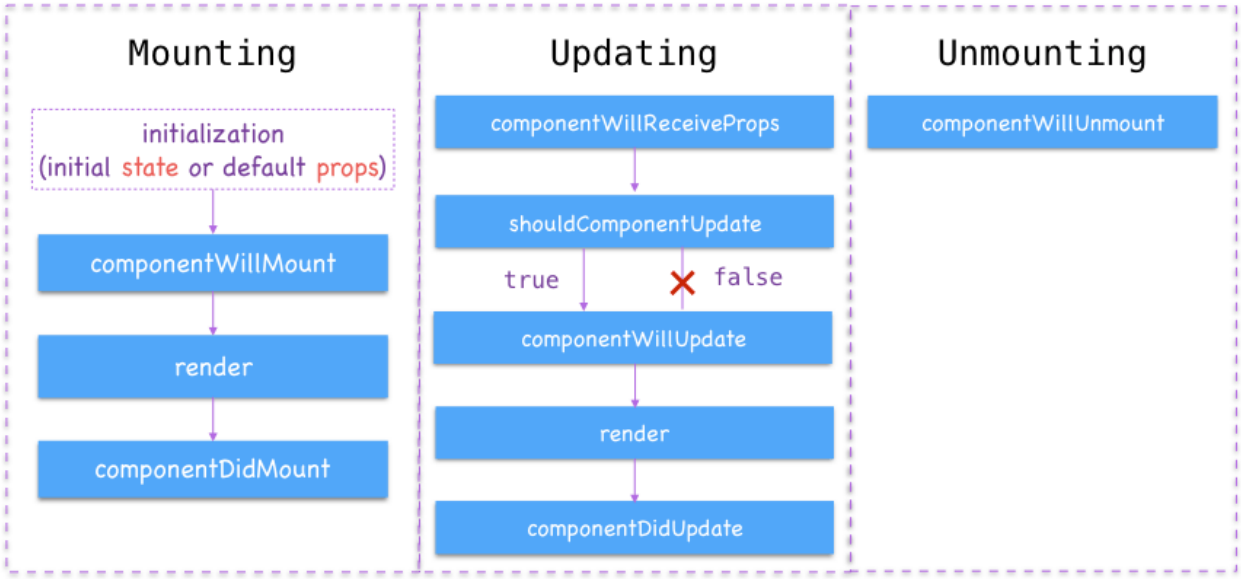
The render method is the only required spec for creating a component, but there are several lifecycle methods & specs we can use that are mighty helpful when you actually want your component to do anything.

a) Lifecycle

A React component in browser can be any of the following three statuses: **mounted, update and unmounted.**



React component lifecycle can be divided into three phases according to these statuses: **mounting, updating and unmounting.**



b) Lifecycle Methods

- componentWillMount – Invoked once, on both client & server before rendering occurs.
- componentDidMount – Invoked once, only on the client, after rendering occurs.
- shouldComponentUpdate – Return value determines whether component should update.
- componentWillUnmount – Invoked prior to unmounting component.

c) Specs

- getInitialState – Return value is the initial value for state.
- getDefaultProps – Sets fallback props values if props aren't supplied.
- mixins – An array of objects, used to extend the current component's functionality.

4.1 Mounting

React.js exposed interfaces or hook methods in each phase of component lifecycle.

a) Initializing state

You can optionally set initial state value in constructor() method of the component if you are using ES6 syntax.

```
const tom_and_jerry = [
   {
      name: 'Tom',
      score: 55
   },
   {
      name: 'Jerry',
      score: 80
   }
];

class ScoreBoard extends React.Component {
   constructor(props) {
      super(props);
      this.state = { players: tom_and_jerry }
   }

   // ...
}
```

If you are using ES5 syntax, getInitialState() in the right place to initialize component state.

```
var ScoreBoard = React.createClass({
   getInitialState: function() {
      return {
         players: tom_and_jerry
      }
   },

   // ...
});
```

The getInitialState() method is called only one time before the component is mounted.

Initialization of state should typically only be done in a top level component, which acts as a role of controller view in your page.

b) Default props

You can also define default values of component props (properties) if the parent component does not declare their values.

Return default props using ES7+ static property initializer.

```
class SinglePlayer extends React.Component {
   static defaultProps = {
      name: 'Nobody',
      score: 0
   }

   // ...
}
```

Default props in ES6:

```
class SinglePlayer extends React.Component {
   // ...
}

SinglePlayer.defaultProps = {
   name: 'Nobody',
   score: 0
}
```

You can define getDefaultProps() method in ES5.

```
var SinglePlayer = React.createClass({
   getDefaultProps: function() {
      return {
         name: 'Nobody',
         score: 0
      }
   }
});
```

The getDefaultProps() method is called only once before any instance of the component is created. So you should avoid using this.props inside getDefaultProps() method.

c) componentWillMount()

The componentWillMount() method is invoked only once before initial rendering.

It is also a good place to set initial state value inside componentWillMount().

```
class SinglePlayer extends React.Component {
   componentWillMount() {
      this.setState({
         isPassed: this.props.score >= 60
      });

      alert('componentWillMount => ' + this.props.name);
      console.log('componentWillMount => ' + this.props.name);
   }

   // ...
}
```

d) componentDidMount()

This lifecycle method will be invoked after rendering. It is the right place to access DOM of the component.

```
class ScoreBoard extends React.Component {
   constructor(props) {
      super(props);
      this._handleScroll = this.handleScroll.bind(this);
   }
   handleScroll() {}
   componentDidMount() {
      alert('componentDidMount in NoticeBoard');
      window.addEventListener('scroll', this._handleScroll);
   }

   // ...
}
```

4.2 Updating

a) componentWillReceiveProps()

void componentWillReceiveProps(object nextProps)

This method will be invoked when a component is receiving new props. componentWillReceiveProps() won't be called for the initial rendering.

```
class SinglePlayer extends React.Component {
   componentWillReceiveProps(nextProps) {
      // Calculate state according to props changes
```

```
      this.setState({
        isPassed: nextProps.score >= 60
      });
    }
}
```

The old props can be accessed via this.props inside componentWillReceiveProps(). Typically, you can set state according to changes of props in this method.

b) shouldComponentUpdate()

boolean shouldComponentUpdate(object nextProps, object nextState)

- **shouldComponentUpdate()** will be invoked before rendering when new props or state are being received. This method won't be called on initial rendering.
- **shouldComponentUpdate()** returns true by default.

This method is usually an opportunity to prevent the unnecessary rerendering considering performance. Just let shouldComponentUpdate() return false, then the render() method of the component will be completely skipped until the next props or state change.

```
class SinglePlayer extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    // Don't rerender if score doesn't change,
    if ( nextProps.score == this.props.score ) {
      return false;
    }

    return true;
  }
}
```

c) componentWillUpdate()

void componentWillUpdate(object nextProps, object nextState)

Invoked just before render(), but after shouldComponentUpdate() (of course, return a true). This method is not called for the initial rendering.

Use this as an opportunity to prepare for an update.

```
class SinglePlayer extends React.Component {
  componentWillUpdate(nextProps, nextState) {
    alert('componentWillUpdate => ' + this.props.name);
    console.log('componentWillUpdate => ' + this.props.name);
  }
}
```

d) componentDidUpdate()

void componentDidUpdate(object prevProps, object prevState)

Invoked immediately after the component's updates are flushed to the DOM. This method is not called for the initial rendering.

You can perform DOM operations after an update inside this function.

```
class SinglePlayer extends React.Component {
   componentDidUpdate(prevProps, prevState) {
      alert('componentDidUpdate => ' + this.props.name);
      console.log('componentDidUpdate => ' + this.props.name);
   }
}
```

4.3 Unmounting

void componentWillUnmount()

This is invoked immediately before a component is unmounted or removed from the DOM.

Use this as an opportunity to perform cleanup operations. For example, unbind event listeners here to avoid memory leaking.

```
class ScoreBoard extends React.Component {
   componentWillUnmount() {
      window.removeEventListener('scroll', this._handleScroll);
   }
}
```

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>React Component Life Cycle Demo</title>

    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
    <div id="app"></div>

    <script type="text/jsx">
        const tom_and_jerry = [
            {
```

```
                name: 'Tom',
                score: 55
            },
            {
                name: 'Jerry',
                score: 80
            }
        ];

        class SinglePlayer extends React.Component {
            constructor(props) {
                super(props);
                this.state = { isPassed: false }
            }
            componentWillMount() {
                // Mark it as 'Pass' if score >= 60
                this.setState({
                    isPassed: this.props.score >= 60
                });

                console.log('componentWillMount => ' + this.props.name);
                alert('componentWillMount => ' + this.props.name);
            }
            componentDidMount() {
                console.log('componentDidMount => ' + this.props.name);
                alert('componentDidMount => ' + this.props.name);
            }
            componentWillReceiveProps(nextProps) {
                // Calculate state according to props changes
                this.setState({
                    isPassed: nextProps.score >= 60
                });

                console.log('componentWillReceiveProps => ' + this.props.name + ':
' + nextProps.score);
                alert('componentWillReceiveProps => ' + this.props.name + ': ' +
nextProps.score);
            }
            shouldComponentUpdate(nextProps, nextState) {
                // Don't rerender if score doesn't change,
                if ( nextProps.score == this.props.score ) {
                    console.log('shouldComponentUpdate => ' + this.props.name + '?
false');
                    alert('shouldComponentUpdate => ' + this.props.name + '?
false');
                    return false;
                }

                console.log('shouldComponentUpdate => ' + this.props.name + '?
true');
                alert('shouldComponentUpdate => ' + this.props.name + '? true');
                return true;
            }
            componentWillUpdate(nextProps, nextState) {
                console.log('componentWillUpdate => ' + this.props.name);
```

```
                alert('componentWillUpdate => ' + this.props.name);
            }
            componentDidUpdate(prevProps, prevState) {
                console.log('componentDidUpdate => ' + this.props.name);
                alert('componentDidUpdate => ' + this.props.name);
            }
            componentWillUnmount() {
                console.log('componentDidUpdate => ' + this.props.name);
                alert('componentDidUpdate => ' + this.props.name);
            }
            render() {
                console.log("render => " + this.props.name);
                return (
                    <div>
                        <h5><span>Name: </span>{this.props.name}</h5>
                        <p><span>Score: </span><em>{this.props.score}</em></p>
                        <p><span>Pass: </span><input type="checkbox"
defaultChecked={this.state.isPassed} disabled={true}  /></p>
                    </div>
                );
            }
        }

        class ScoreBoard extends React.Component {
            constructor(props) {
                super(props);
                this.state = {
                    players: tom_and_jerry
                };
            }
            changeScore(amount) {
                if ( typeof(amount) != "number" ) {
                    return;
                }

                let players = this.state.players;
                let tom = players[0];
                tom.score = tom.score + amount;

                tom.score = (tom.score > 100) ? 100 : tom.score;
                tom.score = (tom.score < 0) ? 0 : tom.score;

                players[0] = tom;
                this.setState({ players: players });
            }
            render() {
                return (
                    <div>
                        <h4>Score Board</h4>
                        <div>
                            <button onClick={ (amount) => this.changeScore(5)
}>Score of Tom: +5</button>
                            <button onClick={ (amount) => this.changeScore(-5)
}>Score of Tom: -5</button>
                        </div>
```

```
                        {
                            this.state.players.map((v, idx) => {
                                return <SinglePlayer key={idx} name={v.name}
score={v.score} />
                            })
                        }
                    </div>
                );
            }
        }



        class App extends React.Component {
            render() {
                return (
                    <div>
                        <h1>React Component Lifecycle Demo</h1>
                        <ScoreBoard />
                    </div>
                )
            }
        }

        // Mount root App component
        ReactDOM.render(<App />, document.getElementById('app'));
    </script>
</body>
</html>
```

5. ReactJS Component – State

State is a double-edged sword. On the one hand, it's useful for sanely controlling internal data. On the other, it can lead to all sorts of mayhem, if not properly controlled.

Every component has a state object and a props object. State is set using the setState method. Calling setState triggers UI updates and is the bread and butter of React's interactivity. If we want to set an initial state before any interaction occurs we can use the getInitialState method.

State, in React component, is internal dataset which affects the rendering of the component. To some extent state can be considered as the private data or data model of React components. React component state is mutable. Once the internal state of a React component is changed, the component will re-render itself according to the new state.

5.1. Setting Initial State

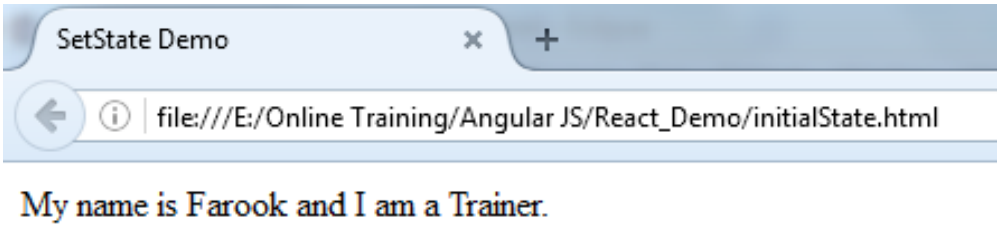
Before we can use state, we need to declare a default set of values for the initial state. This is done by defining a method called getInitialState() and returning an object:

Example:

```
<!DOCTYPE html>
<html>
<head>
```

```
            <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>SetState Demo</title>
</head>
<body>
  <div id="container"></div>
  <script type="text/jsx">
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      name : "Farook",
      job  : "Trainer"
    };
  },
  render : function() {
    return <div>
      My name is {this.state.name}
      and I am a {this.state.job}.
    </div>;
  }
});
ReactDOM.render(<InterfaceComponent />,document.getElementById('container'));

  </script>
</body>
</html>
```

Output:



The getInitialState : function() method tells the component which values should be available from the first render cycle, until the state values are changed. You should never try to use state values without first declaring them in this manner.

5.2. Setting State

Setting state should only be done from inside the component. State should be treated as private data, but there are times when you may need to update it.

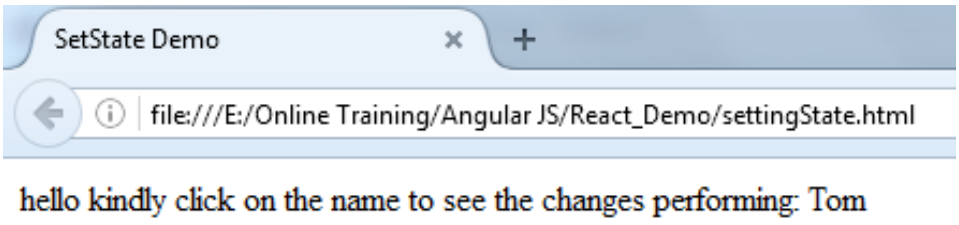
Example:

```
<html>
<head>
```

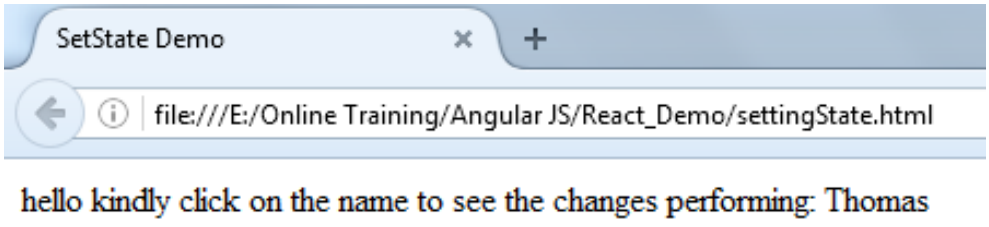
```
            <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
       <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
       <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>SetState Demo</title>
</head>
<body>
  <div id="container"></div>
  <script type="text/jsx">
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      name : "Tom"
    };
  },
  handleClick : function() {
    this.setState({
      name : "Thomas"
    });
  },
  render : function() {
    return <div onClick={this.handleClick}>
      hello kindly click on the name to see the changes performing:
{this.state.name}
    </div>;
  }
});
ReactDOM.render(<InterfaceComponent />,document.getElementById('container'));
  </script>

</body>
</html>
```

Output:

Before Click:



After Click:



5.3. Replacing State

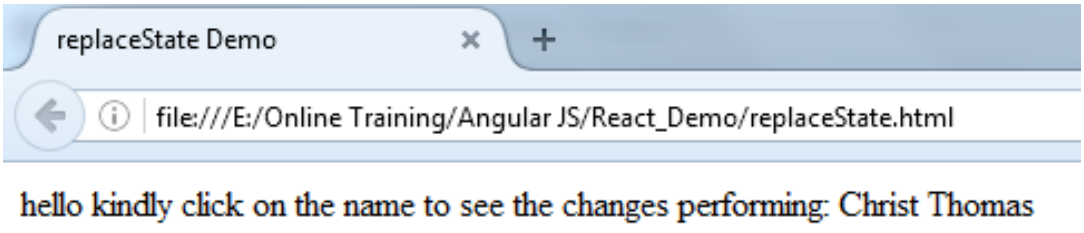
It's also possible to replace values in the state by using the replaceState() method.

Example:

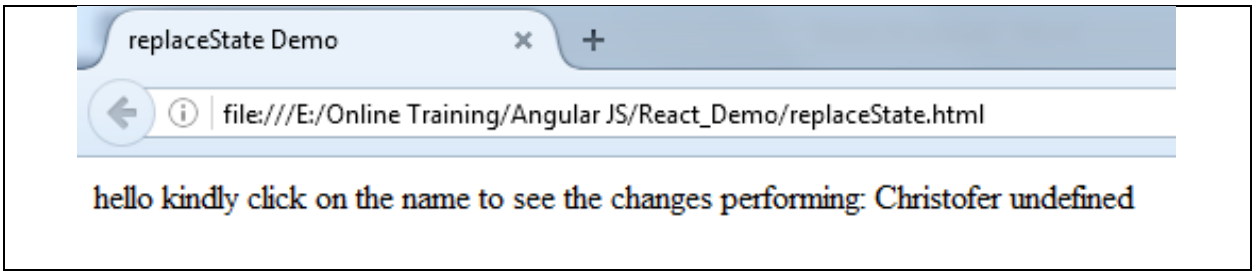
```
<html>
<head>
            <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>replaceState Demo</title>
</head>
<body>
  <div id="container"></div>
  <script type="text/jsx">
var InterfaceComponent = React.createClass({
  getInitialState : function() {
    return {
      first : "Christ",
      last  : "Thomas"
    };
  },
  handleClick : function() {
    this.replaceState({
      first : "Christofer"
    });
  },
  render : function() {
    return <div onClick={this.handleClick}>
      hello kindly click on the name to see the changes performing: {
this.state.first + " " + this.state.last }
    </div>;
  }
});
ReactDOM.render(<InterfaceComponent />,document.getElementById('container'));
</script>
</body>
</html>
```

Output:

Before Click:



After Click:



The use of state should be as limited as it can get. When you use state, you run the risk of introducing a number of (sometimes subtle) errors in the behavior and rendering of your components. A general rule is not to use state for static components. If your component does not need to change, based on external factors, then do not use state. It's better to calculate rendered values in the render() method.

6. ReactJS – Props

Props, which is short for properties, are properties of React component. Props look like HTML attributes. Values of props in a React component are commonly passed from the parent component.

Props are just passed into, so props are immutable in the eyes of the child components and shouldn't be changed directly by the child components.

When we provide a name attribute (and value) this gets passed along to the component. If you come from a history of Object Oriented Programming languages, you may think that these properties are passed to a constructor and that you need to manage them from there. on't fall into this trap!

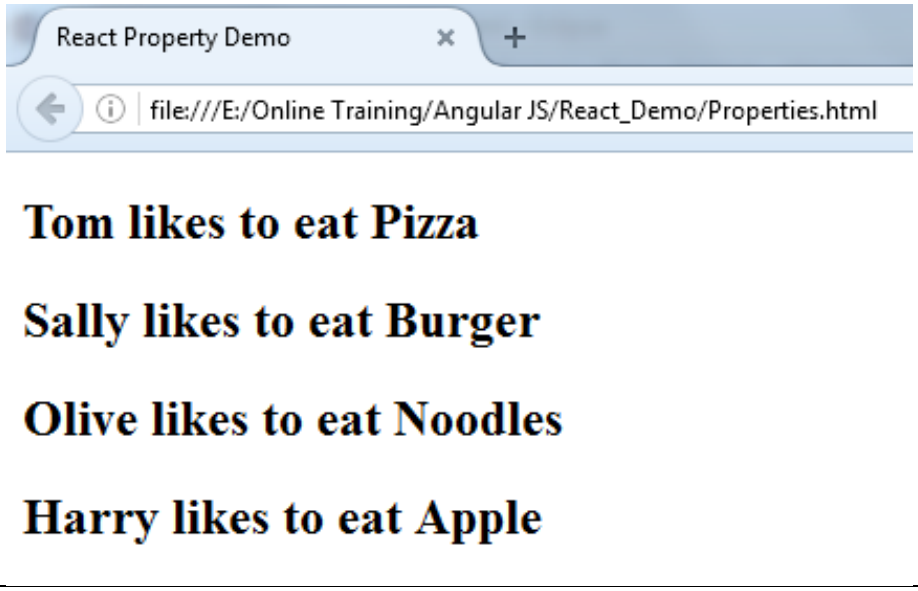
They are actually passed to a constructor, but you should never try to override this behavior with your own constructor functionality. You should only use these values from this.props:

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>React Property Demo</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
</head>
<body>

    <div id="content"></div>

    <script type="text/jsx">

        var MyComponent = React.createClass({
```

```
            render: function() {
                return(
                        <h2>{this.props.user} likes to eat {this.props.food}</h2>
                );
            }
        });

        React.render(
                <div>
                    <MyComponent user="Tom" food="Pizza" />
                    <MyComponent user="Sally" food="Burger" />
                    <MyComponent user="Olive" food="Noodles" />
                    <MyComponent user="Harry" food="Apple" />
                </div>,
                document.getElementById('content')
        );

    </script>

</body>
</html>
```

Output:



7. ReactJS - Events

React also has a built in cross browser events system. The events are attached as properties of components and can trigger methods.

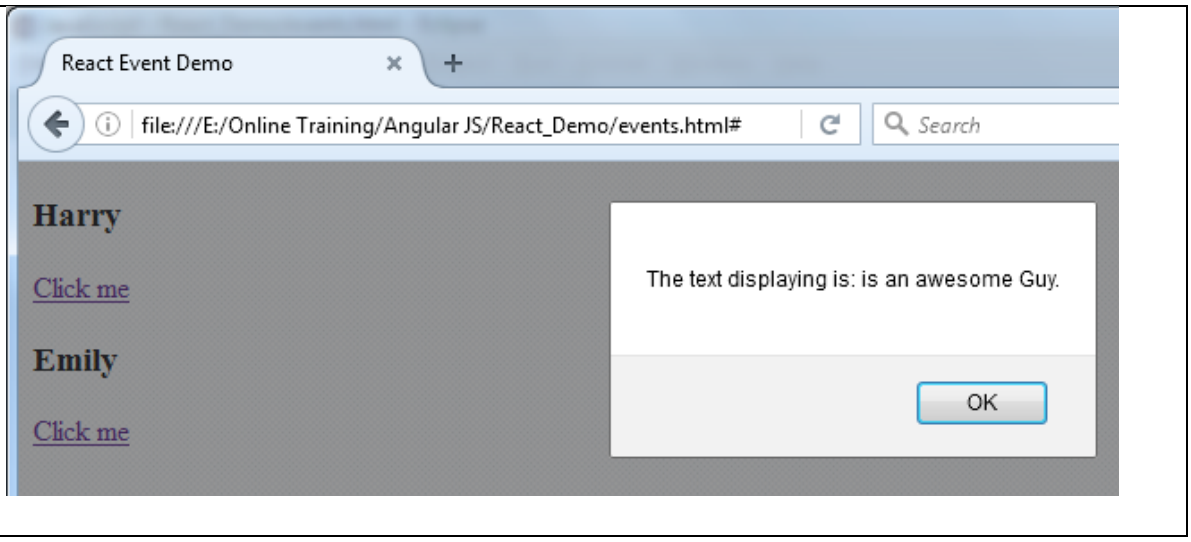
Supported Events

React normalizes events so that they have consistent properties across different browsers. The event handlers are triggered by an event in the bubbling phase. To register an event handler for the capture phase, append Capture to the event name;

for example, instead of using onClick, you would use **onClickCapture** to handle the click event in the capture phase.

Example:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>React Event Demo</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
</head>
<body>

    <div id="content"></div>

    <script type="text/jsx">

        var MyComponent = React.createClass({
            doSomething: function() {
                alert("The text displaying is: " + this.props.children);
            },
            render: function() {
                return(
                        <div>
                            <h3>{this.props.user}</h3>
                            <a onClick={this.doSomething} href="#" >Click me</a>
                        </div>
                );
            }
        });

        React.render(
                <div>
                    <MyComponent user="Harry">is an awesome Guy.</MyComponent>
                    <MyComponent user="Emily">is a gorgeous Girl.</MyComponent>
                </div>,
                document.getElementById('content')
        );

    </script>

</body>
</html>
Output:
```



8. ReactJS – JSX

It's called JSX, and it is a Javascript XML syntax transform. This lets you write HTML-ish tags in your Javascript. React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, but there are some pros that comes with it.

- JSX is faster because it performs optimization while compiling code to JavaScript.
- It is also type-safe and most of the errors can be caught during compilation.
- JSX makes it easier and faster to write templates if you are familiar with HTML.

JSX is a statically-typed, object-oriented programming language compiling to standalone JavaScript. The reason why JSX was developed is our need for a more robust programming language than JavaScript. JSX is, however, fairly close to JavaScript especially in its statements and expressions.

Statically-typed programming language is robust because certain sorts of problems, for example typos in variable names or missing function definitions, are detected at compile-time. This is important especially in middle- to large-scale software development in which a number of engineers may be engaged.

Therefore, JSX is designed as a statically-typed language. All the values and variables have a static type and you can only assign a correctly-typed value to a variable. In addition, all the functions including closures have types which are determined by the types of parameters and the return values, where you cannot call a function with incorrectly typed arguments.

Also, another important reason why JSX was developed is to boost JavaScript performance. JavaScript itself is not so slow but large-scale development tends to have many abstraction layers, e.g. proxy classes and accessor methods, which often have negative impact on performance. JSX boosts performance by inline expansion: function bodies are expanded to where they are being

called, if the functions being called could be determined at compile-time. This is the power of the statically-typed language in terms of performance.

JSX Syntax:

```
/** @jsx React.DOM */
React.render(
  React.createElement('h1', null, 'Hello!'),
  document.getElementById('myDiv')
);
```

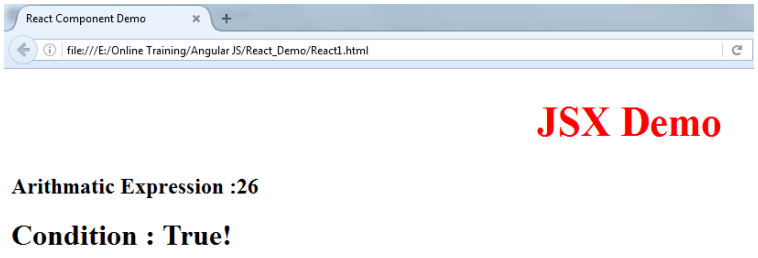
Using JSX

JSX looks like regular HTML in most cases.

Example includes: Nested Elements, Styling, JS Expressions

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>React Component Demo</title>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
</head>
<body>
    <div id="content"></div>
    <script type="text/jsx">
       var MyComponent = React.createClass({
            render: function() {
                        var i = 10;
                        var jsxStyle = {
                              fontSize: 45,
                              color: '#FF0000'
                              }
                return (
                                    <div>
                                    <center><h1 style={jsxStyle}> JSX
Demo</h1></center>
                                  <h2>Arithmetic Expression   :{12+14}</h2>
                                    <h1>Condition : {i == 10 ? 'True!' :
'False'}</h1>
                                </div>
                    );
            }
        });
        React.render(<MyComponent />, document.getElementById('content'));
    </script>
```

```
</body>
</html>
```



Nested Elements

If you want to return more elements, you need to wrap it with one container element.

Naming Convention

HTML tags are always using lowercase tag names, while React components starts with Uppercase.

Comments

When writing comments you need to put curly brackets {} when you want to write comment within children section of a tag. It is good practice to always use {} when writing comments since you want to be consistent when writing the app.

Example: {// single line comment}

{/*Multi Line Comment*/}

Difference between props and state

- state is internal and controlled by the component itself.
- props are external and controlled by whatever renders the component.

9. Reactjs - Mixins

Mixins are a very simple thing that makes your React apps even more reusable. Components are the best way to reuse code in React, but sometimes very different components may share some common functionality. These are sometimes called cross-cutting concerns. React provides mixins to solve this problem.

Why Mixins?

React shies away from sub classing components, but as we all know, repeating yourself is just as bad as writing the same code over and over. So to give the same functionality to multiple components, you encapsulate it into a mixin and include it in your classes. One could go so far as to say React favors composition over inheritance.

Creating a mixin:

```
var DefaultMixin = {
```

```
    getDefaultProps: function () {
        return {name: "Henry"};
    }
};
```

To use this mixin, we simply ensure our component has a mixins property and wrap our mixin in an array as its value. we can include our mixin in any number of components:

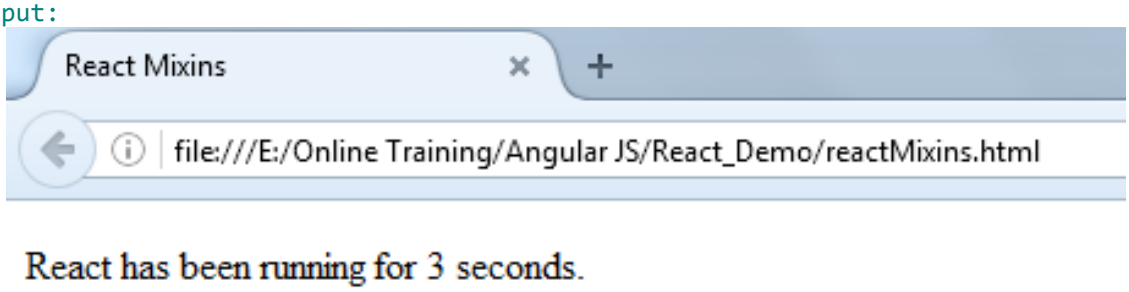
```
var ComponentOne = React.createClass({
  mixins: [DefaultMixin],
  render: function() {
    return <h2>Hello {this.props.name}</h2>;
  }
});

ReactDOM.render (<ComponentOne />, document.body);
```

Example1: One common use case is a component wanting to update itself on a time interval. It's easy to use setInterval(), but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know when a component is about to be created or destroyed.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Mixins</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
<div id="container"></div>
<script type="text/jsx">
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};
```

```
var Clock = React.createClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

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

</script>
</body>
</html>
```

Output:

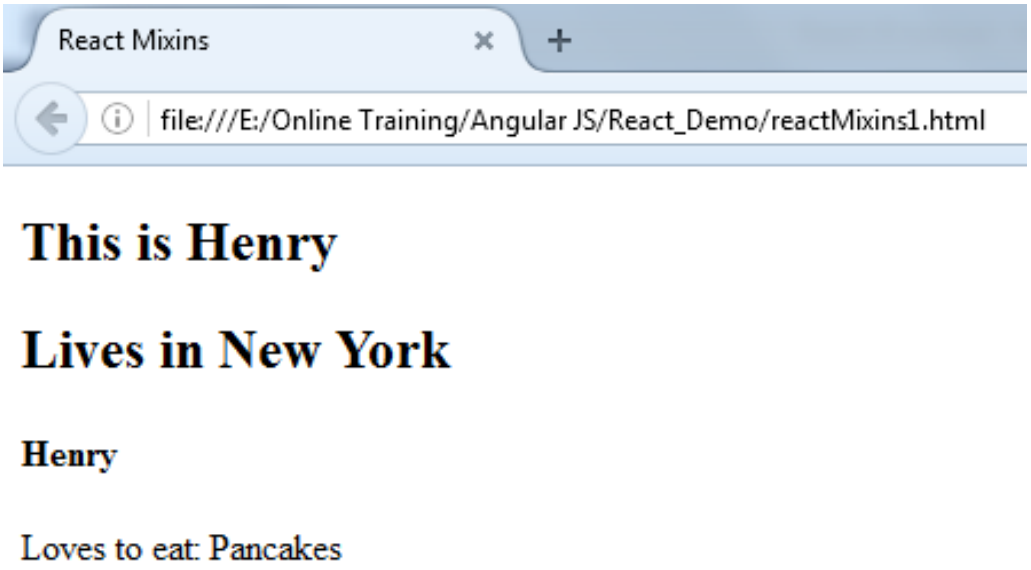


Example 2:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Mixins</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
```

```
<div id="container"></div>
<script type="text/jsx">
var DefaultMixin = {
    getDefaultProps: function () {
        return {name: "Henry"};
    }
};

var OuterComponent = React.createClass({
    render: function () {
        return (
            <div>
                <ComponentOne />
                <ComponentTwo city="New York"/>
                <ComponentThree/>
            </div>
        );
    }
});

var ComponentOne = React.createClass({
    mixins: [DefaultMixin],
    render: function() {
        return <h2>This is {this.props.name}</h2>;
    }
});

var ComponentTwo = React.createClass({
    mixins: [DefaultMixin],
    render: function() {
        return <h2>Lives in {this.props.city}</h2>;
    }
});

var ComponentThree = React.createClass({
    mixins: [DefaultMixin],
    getDefaultProps: function () {
        return {food: "Pancakes"};
    },
    render: function () {
        return (
            <div>
                <h4>{this.props.name}</h4>
                          <p>Loves to eat: {this.props.food}</p>
            </div>
        );
    }
});

ReactDOM.render(<OuterComponent/>, document.getElementById('container'));
</script>
</body>
</html>
```

Output:



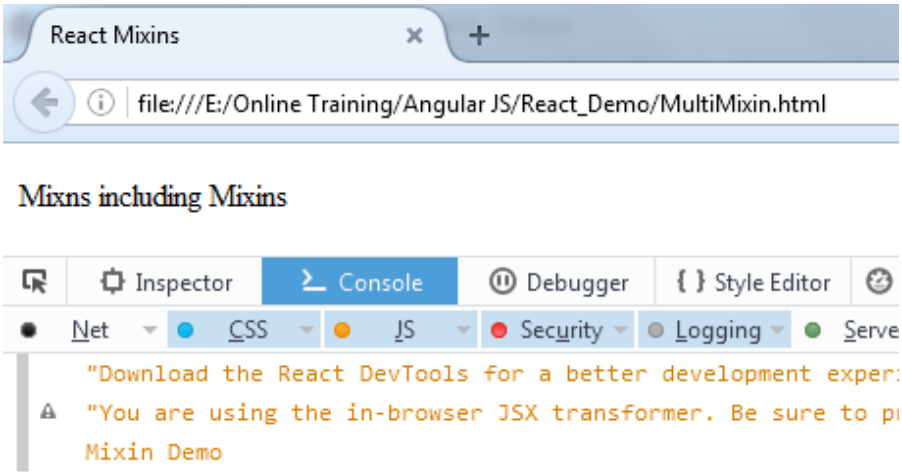
mixins is something that you can define multiple times, meaning that mixins can include mixins:

Example:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Mixins</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
<div id="container"></div>
<script type="text/jsx">
var UselessMixin = {
    componentDidMount: function () {
      console.log("Mixin Demo");
    }
};

var LolMixin = {
   mixins: [UselessMixin]
};

var MixnsOpinion = React.createClass({
   mixins: [LolMixin],
```

```
   render: function () {
       return (<p>Mixns including Mixins</p>);
   }
});

ReactDOM.render(<MixnsOpinion />, document.getElementById('container'));
</script>
</body>
</html>
```

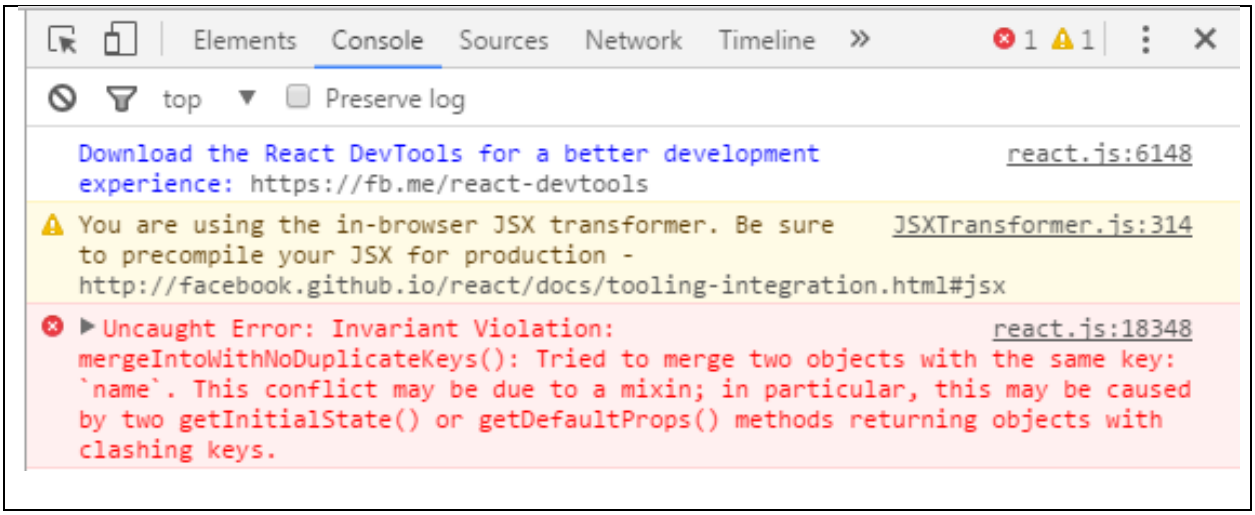
Output:



There are a few things to note when using mixins that can cause headaches. Fortunately, the list seems pretty small. The ability to have multiple calls to getDefaultProps (and getIntialState as well) can get you into trouble if you attempt to define the same prop in different places:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Mixins</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
<div id="container"></div>
<script type="text/jsx">
var DefaultMixin = {
    getDefaultProps: function () {
        return {name: "Henry"};
```

```
    }
};

var OuterComponent = React.createClass({
    render: function () {
        return (
            <div>
                <ComponentOne />
                <ComponentTwo city="New York"/>
                          <ComponentThree/>
            </div>
        );
    }
});

var ComponentOne = React.createClass({
    mixins: [DefaultMixin],
    render: function() {
        return <h2>This is {this.props.name}</h2>;
    }
});

var ComponentTwo = React.createClass({
    mixins: [DefaultMixin],
    render: function() {
        return <h2>Lives in {this.props.city}</h2>;
    }
});

var ComponentThree = React.createClass({
    mixins: [DefaultMixin],
    getDefaultProps: function () {
        return {food: "Pancakes",
                          name: "Tommy"};
    },
    render: function () {
        return (
            <div>
                <h4>{this.props.name}</h4>
                          <p>Loves to eat: {this.props.food}</p>
            </div>
        );
    }
});

ReactDOM.render(<OuterComponent/>, document.getElementById('container'));
</script>
</body>
</html>

Output:
```

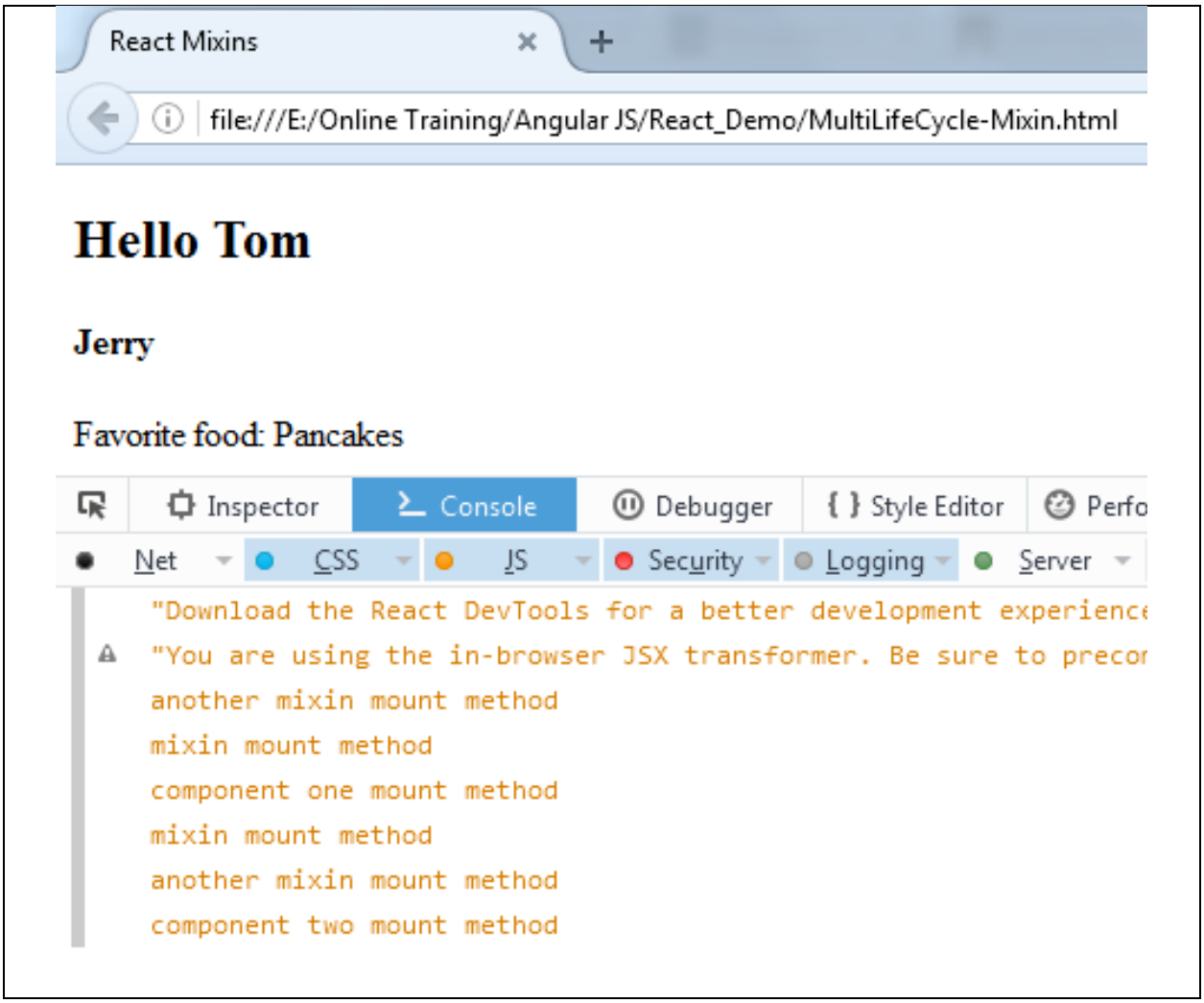


This same error will happen if we attempt to set the same state key in multiple calls to getIntialState.

Multiple Lifecycle Method Call Order of Execution

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>React Mixins</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
</head>
<body>
<div id="container"></div>
<script type="text/jsx">
var LogOnMountMixin = {
    componentDidMount: function () {
        console.log("mixin mount method");
    },
};

var MoreLogOnMountMixin = {
    componentDidMount: function () {
        console.log("another mixin mount method");
    },
};

var OuterComponent = React.createClass({
    render: function () {
        return (
            <div>
```

```
                <ComponentOne name="Tom" />
                <ComponentTwo name="Jerry" />
            </div>
        );
    }
});

var ComponentOne = React.createClass({
    mixins: [MoreLogOnMountMixin, LogOnMountMixin],
    componentDidMount: function () {
        console.log("component one mount method");
    },
    render: function() {
        return <h2>Hello {this.props.name}</h2>;
    }
});

var ComponentTwo = React.createClass({
    mixins: [ LogOnMountMixin, MoreLogOnMountMixin],
    getDefaultProps: function () {
        return {food: "Pancakes"};
    },
    componentDidMount: function () {
        console.log("component two mount method");
    },
    render: function () {
        return (
            <div>
                <h4>{this.props.name}</h4>
                <p>Favorite food: {this.props.food}</p>
            </div>
        );
    }
});
ReactDOM.render(<OuterComponent/>, document.getElementById('container'));

</script>
</body>
</html>

Output:
```



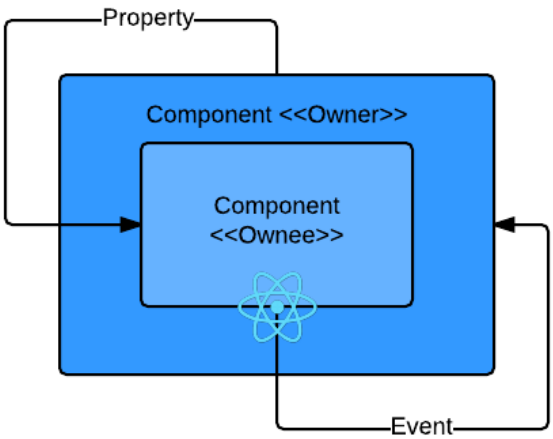
Component inter communication

1. **Component composition**

ReactJS follows the current trend of component oriented Web UI design. Component oriented design is nothing new; we use to write applications in Qt (Cross platform application framework) which uses widgets. Java Server Faces, WebForms and others do the same on server side. Nowadays, we can see the same trend hitting client side, like WebComponents, Directives in Angular, FlightJS, and of course ReactJS.

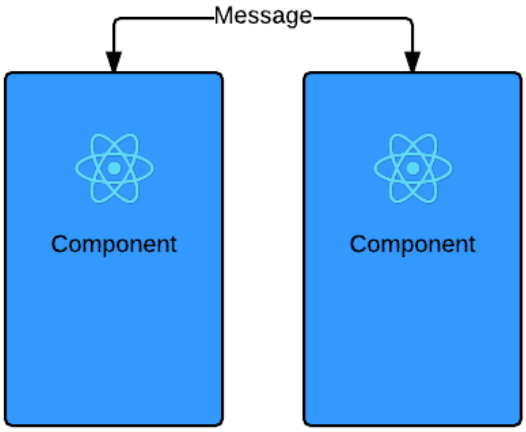
Two models of Composition are:

a. **A Coupled Composition** means that one component is the owner of another component. The owning component can be considered as a parent, while the owned component becomes a child. This kind of coupling leads to a typical component hierarchy with parent and children components.



Coupled Composition

b. **A Decoupled Composition** is used when none of the components is owner, nor owned. These components exist on the same screen, nevertheless they are not bound or related to each other.



Decoupled Composition

The composition modality defines the way how components communicate which each other.

2. **Pass data from parent to child**

Receiving State from Parent Component (props, propTypes, getDefaultProps). props is the data which is passed to the child component from the parent component. This allows our React architecture to stay pretty straight forward. Handle state is the highest most parent component which needs to use the specific data, and if you have a child component that also needs that data, pass that data down as props.

Have two components now. One parent, one child. The parent is going to keep track of the state and pass a part of that state down to the child as props.

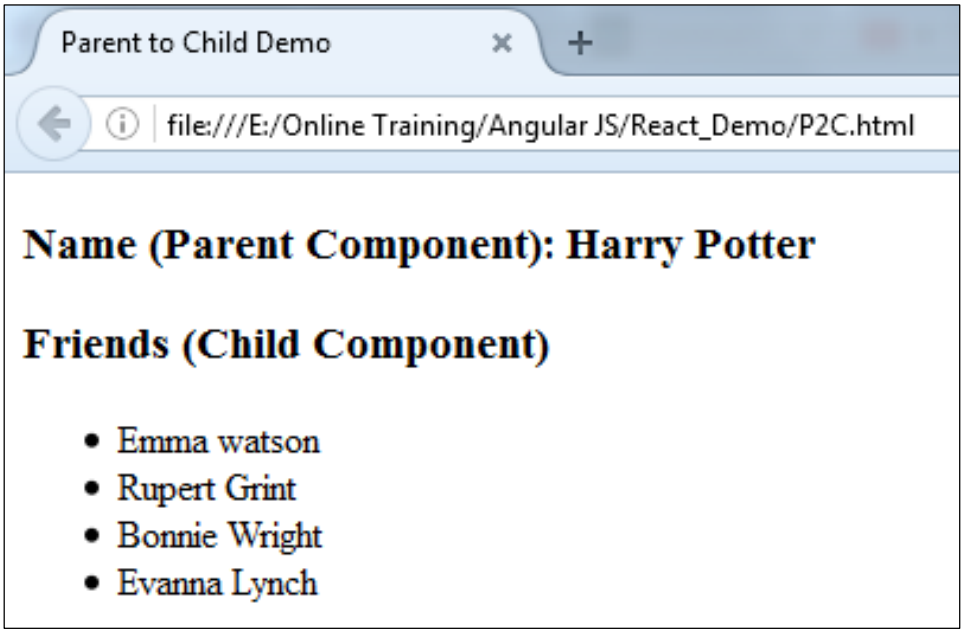
There really isn't much going on in component, We have an initial state and we pass part of that initial state to another component. The majority of the new code will come from this child component.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Parent to Child Demo</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
    <div id="content"></div>
    <script type="text/jsx">
var FriendsContainer = React.createClass({
    getInitialState: function(){
      return {
        name: 'Harry Potter',
        friends: ['Emma watson', 'Rupert Grint', 'Bonnie Wright','Evanna Lynch']
      }
    },
    render: function(){
      return (
        <div>
          <h3> Name (Parent Component): {this.state.name} </h3>
          <ShowList names={this.state.friends} />
        </div>
      )
    }
});

var ShowList = React.createClass({
  render: function(){
    var listItems = this.props.names.map(function(friend){
      return <li> {friend} </li>;
    });
    return (
        <div>
          <h3> Friends (Child Component) </h3>
          <ul>
            {listItems}
          </ul>
        </div>
    )
  }
});

ReactDOM.render(<FriendsContainer />, document.getElementById('content'));
```

```
</script>
</body>
</html>
```

Output:



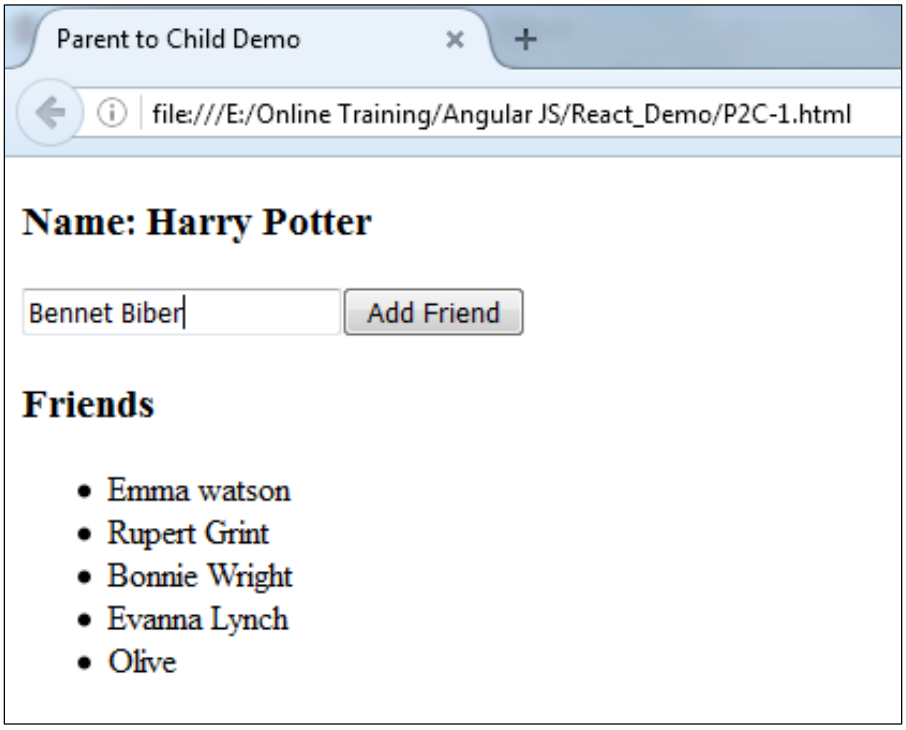
Example 2: Passing a List to a Child Component with a Setter Method with Default Props

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Parent to Child Demo</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
    <div id="content"></div>
    <script type="text/jsx">
    var FriendsContainer = React.createClass({
    getInitialState: function(){
      return {
        name: 'Harry Potter',
        friends: ['Emma watson', 'Rupert Grint', 'Bonnie Wright','Evanna Lynch']
      }
    },
    addFriend: function(friend){
      this.state.friends.push(friend);
      this.setState({
        friends: this.state.friends
```

```
      });
    },
    render: function(){
      return (
        <div>
          <h3> Name: {this.state.name} </h3>
          <AddFriend addNew={this.addFriend} />
          <ShowList names={this.state.friends} />
        </div>
      )
    }
});

var AddFriend = React.createClass({
  getInitialState: function(){
    return {
      newFriend: ''
    }
  },
  propTypes: {
    addNew: React.PropTypes.func.isRequired
  },
  updateNewFriend: function(e){
    this.setState({
      newFriend: e.target.value
    });
  },
  handleAddNew: function(){
    this.props.addNew(this.state.newFriend);
    this.setState({
      newFriend: ''
    });
  },
  render: function(){
    return (
        <div>
          <input type="text" value={this.state.newFriend}
onChange={this.updateNewFriend} />
          <button onClick={this.handleAddNew}> Add Friend </button>
        </div>
    );
  }
});

var ShowList = React.createClass({
  getDefaultProps: function(){
    return {
      names: []
    }
  },
  render: function(){
    var listItems = this.props.names.map(function(friend){
      return <li> {friend} </li>;
    });
    return (
```

```
        <div>
          <h3> Friends </h3>
          <ul>
            {listItems}
          </ul>
        </div>
    )
  }
});

ReactDOM.render(<FriendsContainer />, document.getElementById('content'));
    </script>
</body>
</html>
```

Output:



3. Hierarchy problem

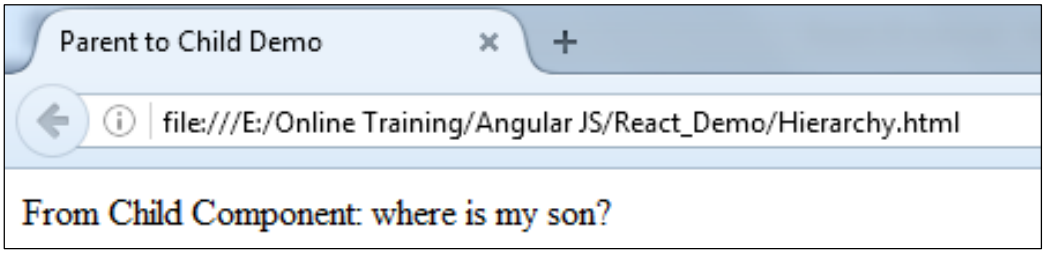
One disadvantage of this technique is if you want to pass down a prop to a grandson (you have a hierarchy of components): the son has to handle it first, then pass it to the grandson. So with a more complex hierarchy, it's going to be impossible to maintain.

A solution exists to avoid this issue and automatically have the parent talking to its grandson without the child to pass properties manually, but it is still not officially fully supported by the React team: the context.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Parent to Child Demo</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
    <div id="content"></div>
    <script type="text/jsx">
      var MyContainer = React.createClass({
    render: function() {
        return  <Intermediate text="where is my son?" />;
              }
      });

      var Intermediate = React.createClass({
       render: function() {
    // Intermediate doesn't care of "text", but it has to pass it down nonetheless
    return <Child text={this.props.text} />;
              }
      });

      var Child = React.createClass({
    render: function() {
        return <span>From Child Component: {this.props.text}</span>;
              }
      });
ReactDOM.render(<MyContainer />, document.getElementById('content'));
</script>
</body>
</html>
```

Output:



4. **Pass Data from Child to Parent**

The parent will pass a callback through a prop: we can pass anything through them, they are not DOM attributes, they are pure Javascript object.

Here is an example where the <ToggleButton> notify its owner its state changed. The parent listens to this event and change its own state too to adapt its message:

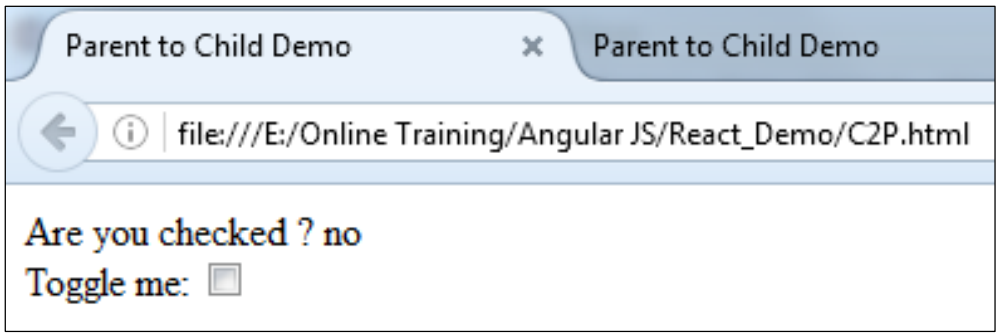
When I click on the input, the parent gets notified, and changes its message to 'yes'.

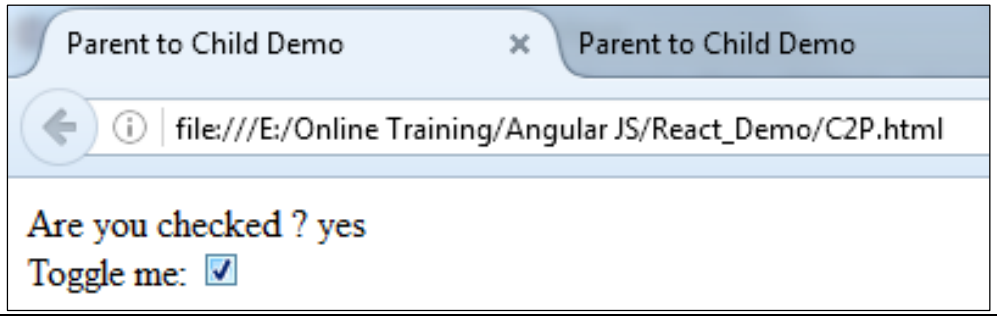
We have the same problem than before: if you have intermediate components in-between, you have to pass your callback through the props of all of the intermediate components to get to your target.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Parent to Child Demo</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
    <div id="content"></div>
    <script type="text/jsx">
    var MyContainer = React.createClass({
    getInitialState: function() {
        return { checked: false };
    },
    onChildChanged: function(newState) {
        this.setState({ checked: newState });
    },
    render: function() {
        return  <div>
                  <div>Are you checked ? {this.state.checked ? 'yes' : 'no'}</div>
                  <ToggleButton text="Toggle me"
initialChecked={this.state.checked} callbackParent={this.onChildChanged} />
                </div>;
    }
});

var ToggleButton = React.createClass({
  getInitialState: function() {
    // we ONLY set the initial state from the props
    return { checked: this.props.initialChecked };
  },
  onTextChanged: function() {
```

```
    var newState = !this.state.checked;
    this.setState({ checked: newState });
    this.props.callbackParent(newState); // hey parent, I've changed!
  },
    render: function() {
        return <label>{this.props.text}: <input type="checkbox"
checked={this.state.checked} onChange={this.onTextChanged}/></label>;
    }
});

ReactDOM.render(<MyContainer />, document.getElementById('content'));
    </script>
</body>
</html>
```

Output:





React internals

Reconciliation algorithm

React's key design decision is to make the API seem like it re-renders the whole app on every update. This makes writing applications a lot easier but is also an incredible challenge to make it tractable. With powerful heuristics we can manage to turn a O(n3) problem into a O(n) one.

Generating the minimum number of operations to transform one tree into another is a complex and well-studied problem. The state of the art algorithms have a complexity in the order of O(n3) where n is the number of nodes in the tree.

Since an optimal algorithm is not tractable, we implement a non-optimal O(n) algorithm using heuristics based on two assumptions:

1. Two components of the same class will generate similar trees and two components of different classes will generate different trees.
2. It is possible to provide a unique key for elements that is stable across different renders.

In practice, these assumptions are ridiculously fast for almost all practical use cases.

Pair-wise diff
In order to do a tree diff, we first need to be able to diff two nodes

- **Different Node Types**

If the node type is different, React is going to treat them as two different sub-trees, throw away the first one and build/insert the second one.

Code

```
renderA: <div />
renderB: <span />
=> [removeNode <div />], [insertNode <span />]
```

The same logic is used for custom components. If they are not of the same type, React is not going to even try at matching what they render. It is just going to remove the first one from the DOM and insert the second one.

Code

```
renderA: <Header />
renderB: <Content />
=> [removeNode <Header />], [insertNode <Content />]
```

It is very unlikely that a <Header> element is going to generate a DOM that is going to look like what a <Content> would generate. Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

It is important to remember that the reconciliation algorithm is an implementation detail. React could re-render the whole app on every action; the end result would be the same. We are regularly refining the heuristics in order to make common use cases faster.

In the current implementation, you can express the fact that a sub-tree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will re-render that full sub-tree.

Because we rely on two heuristics, if the assumptions behind them are not met, performance will suffer.

The algorithm will not try to match sub-trees of different components classes. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class. In practice, we haven't found this to be an issue.

Keys should be stable, predictable, and unique. Unstable keys (like those produced by Math.random()) will cause many nodes to be unnecessarily re-created, which can cause performance degradation and lost state in child components.

List-wise diff

1. **Problematic Case**

In order to do children reconciliation, React adopts a very naive approach. It goes over both lists of children at the same time and generates a mutation whenever there's a difference.

For example if you add an element at the end:

```
Code

renderA: <div><span>first</span></div>
renderB: <div><span>first</span><span>second</span></div>
=> [insertNode <span>second</span>]
```

Inserting an element at the beginning is problematic. React is going to see that both nodes are spans and therefore run into a mutation mode.

```
Code

renderA: <div><span>first</span></div>
renderB: <div><span>second</span><span>first</span></div>
=> [replaceAttribute textContent 'second'], [insertNode <span>first</span>]
```

2. **Keys**

In order to solve this seemingly intractable issue, an optional attribute has been introduced. You can provide for each child a key that is going to be used to do the matching. If you specify a key, React is now able to find insertion, deletion, substitution and moves in O(n) using a hash table.
In practice, finding a key is not really hard. Most of the time, the element you are going to display already has a unique id. When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. Remember that the key only has to be unique among its siblings, not globally unique.

```
Code

renderA: <div><span key="first">first</span></div>
renderB: <div><span key="second">second</span><span key="first">first</span></div>
=> [insertNode <span>second</span>]
```

Component Styling in ReactJS

1. CSS Modules

For generations, we have styled HTML content using CSS. Things were good. With CSS, we had a good separation between the content and the presentation. The selector syntax gave us a lot of flexibility in choosing which elements to style and which ones to skip. We couldn't even find too many issues to hate the whole cascading thing that CSS is all about.

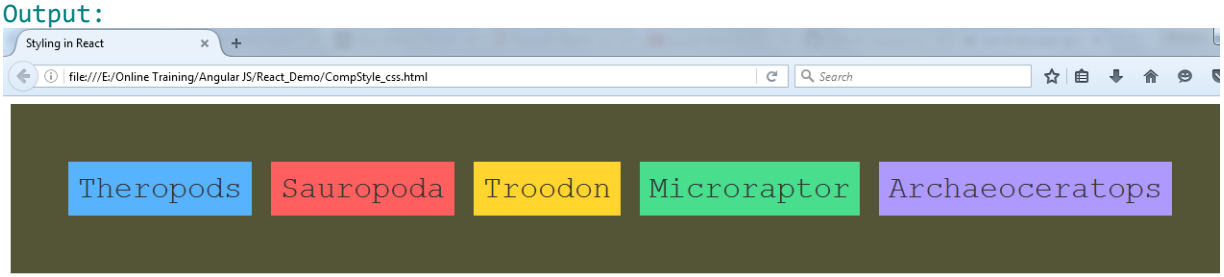
Two aspects of styling are component styling and object Styling.

Example:

```
<!DOCTYPE html>
<html>

<head>
  <title>Styling in React</title>
  <script src="https://fb.me/react-15.1.0.js"></script>
  <script src="https://fb.me/react-dom-15.1.0.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>

  <style>
    #container {
      padding: 50px;
      background-color: #545437;
    }
  </style>
</head>

<body>
  <div id="container"></div>
 <script type="text/babel">

  var Data = React.createClass({
  render: function() {
      var DataStyle = {
             padding: 10,
             margin: 10,
             backgroundColor: this.props.bgcolor,
             color: "#333",
             display: "inline-block",
             fontFamily: "monospace",
             fontSize: "32",
             textAlign: "center"
      };

      return (
        <div style={DataStyle}>
          {this.props.children}
        </div>
      );
    }
});
```

```
  var destination = document.querySelector("#container");
  ReactDOM.render(
    <div>
     <Data bgcolor="#58B3FF">Theropods</Data>
    <Data bgcolor="#FF605F">Sauropoda</Data>
    <Data bgcolor="#FFD52E">Troodon</Data>
    <Data bgcolor="#49DD8E">Microraptor</Data>
    <Data bgcolor="#AE99FF">Archaeoceratops</Data>
    </div>,
    destination
  );

</script>
</body>
</html>
```

Output:



2. LESS CSS

Less is a CSS pre-processor, meaning that it extends the CSS language, adding features that allow variables, mixins, functions and many other techniques that allow you to make CSS that is more maintainable, theme able and extendable.

Example: Flipping tile using CSS3 animations and React JS.

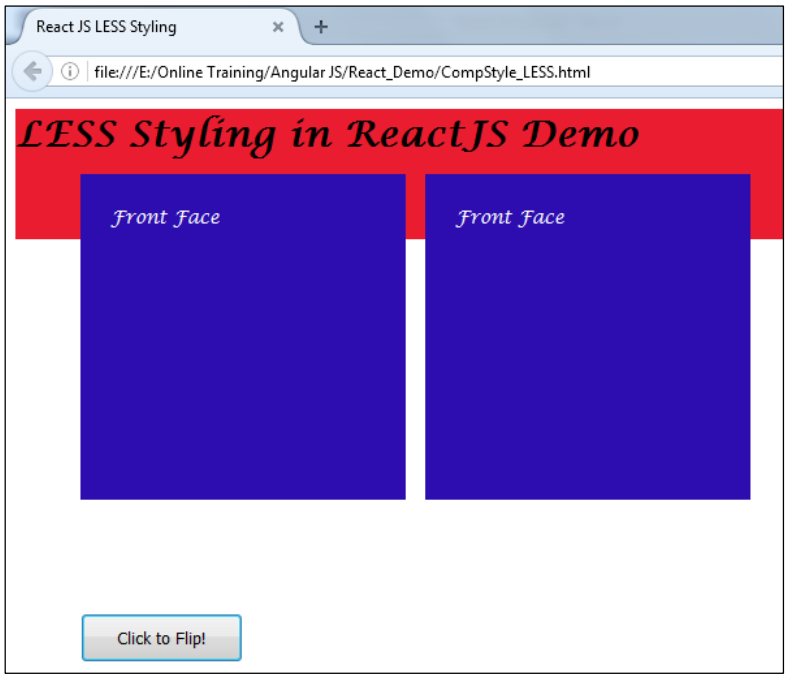
```
<!DOCTYPE html>

<html>
       <head>
              <title>React JS LESS Styling</title>
              <style type="text/less">
#app {
      padding: 50px;
      background-color: #EA1D30;
    }
                    .transition (@value1,@value2:X,...) { @value:
~`"@{arguments}".replace(/[\[\]]|\,\sX/g, '')`; -webkit-transition: @value; -moz-
transition: @value; -ms-transition: @value; -o-transition: @value; transition:
@value; }
                    .transform (@value1,@value2:X,...) { @value:
~`"@{arguments}".replace(/[\[\]]|\,\sX/g, '')`; transform:@value; -ms-
transform:@value; -webkit-transform:@value; -o-transform:@value; -moz-
transform:@value; }
```

```
                    .transform-style(@style:preserve-3d) { transform-style:@style;
-webkit-transform-style:@style; -moz-transform-style:@style; -ms-transform-
style:@style; }
            .backface-visibility-hidden { backface-visibility:hidden; -webkit-
backface-visibility:hidden; }
                    .perspective(@amount: 1000px) { perspective:@amount; -webkit-
perspective:@amount; -moz-perspective:@amount; -ms-perspective:@amount; }
                    .tile { color:white; padding:23px; box-sizing:border-box; -moz-
box-sizing:border-box; }

                    body { font-family:Lucida Calligraphy; font-size:14px; }
                    body>span, body>h1 { float:left; width:100%; margin:0;
padding:0; margin-bottom:10px; }

                    span { color:#800000; }

                    div.button-container { float:left; width:250%; margin-top:87px;
                          button { width:auto; padding:7px 22px; }
                    }


                    .flipper-container { float:left; width:250px; height:250px;
margin-right:15px; display:block; .perspective;
                          span { color:white; }
                          >div.flipper { float:left; width:100%; height:100%;
position:relative; .transform-style(preserve-3d);
                                .front, .back { float:left; display:block;
width:100%; height:100%; .backface-visibility-hidden; position:absolute; top:0;
left:0; .transform-style(preserve-3d); .transition(-webkit-transform ease 500ms);
.transition(transform ease 500ms); }

                                .front {
                                      z-index:2;
                                      background:#2D0DB0;

                                      /* front tile styles go here! */
                                }

                                .back {
                                      background:#B00D9A;

                                      /* back tile styles go here! */
                                }
                          }
                    }

                    .flipper-container.horizontal {
                          .front { .transform(rotateY(0deg)); }
                          .back { .transform(rotateY(-180deg)); }

                          div.flipper.flipped {
                                .front { .transform(rotateY(180deg)); }
                                .back { .transform(rotateY(0deg)); }
                          }
                    }
```

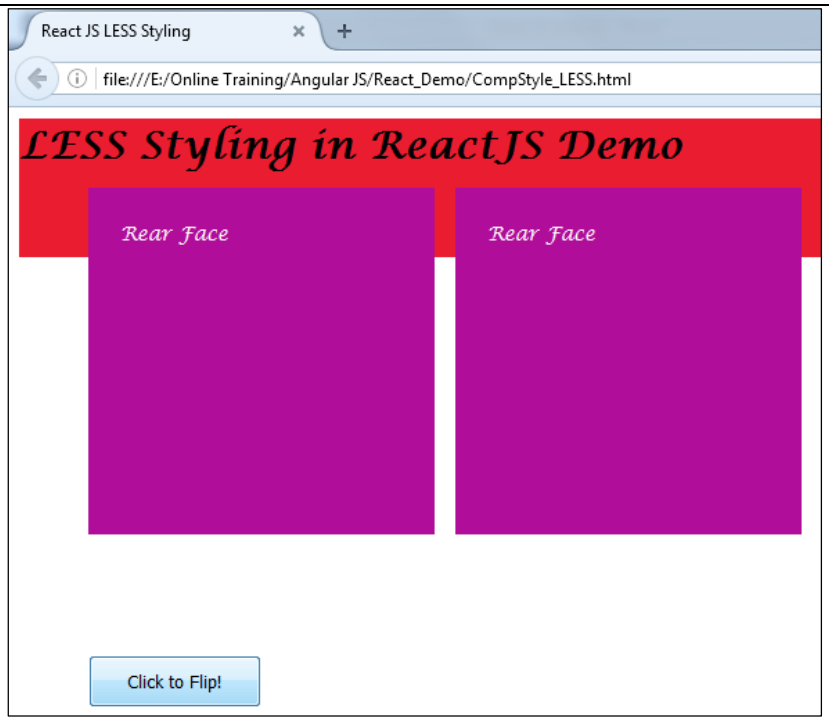
```
                        .flipper-container.vertical {
                                .front { .transform(rotateX(0deg)); }
                                .back { .transform(rotateX(-180deg)); }

                                div.flipper.flipped {
                                        .front { .transform(rotateX(180deg)); }
                                        .back { .transform(rotateX(0deg)); }
                                }
                        }
                </style>
                <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.12.2/react.js"></script>
                <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.12.2/JSXTransformer.js"></scrip
t>
                <script type="text/javascript"
src="https://cdnjs.cloudflare.com/ajax/libs/less.js/1.7.5/less.min.js"></script>

                <script type="text/jsx">
                        var App = React.createClass({
                                getInitialState: function() {
                                        return {
                                                flipped: false
                                        };
                                },

                                flip: function() {
                                        this.setState({ flipped: !this.state.flipped });
                                },

                                render: function() {
                                        return <div>
                                                <Flipper flipped={this.state.flipped}
orientation="horizontal" />
                                                <Flipper flipped={this.state.flipped}
orientation="vertical" />

                                                <div className="button-container">
                                                        <button onClick={this.flip}>Click to
Flip!</button>
                                                </div>
                                        </div>;
                                }
                        });

                        var Flipper = React.createClass({
                                render: function() {
                                        return <div className={"flipper-container " +
this.props.orientation}>
                                                <div className={"flipper" +
(this.props.flipped ? " flipped" : "")}>
                                                        <Front>Front Face</Front>
                                                        <Back>Rear Face</Back>
                                                </div>
```

```
                                    </div>;
                              }
                        });

                        var Front = React.createClass({
                              render: function() {
                                    return <div className="front
tile">{this.props.children}</div>;
                              }
                        });

                        var Back = React.createClass({
                              render: function() {
                                    return <div className="back
tile">{this.props.children}</div>;
                              }
                        });

                        React.render(<App />, document.getElementById('app'));
                  </script>
            </head>
            <body>
                  <h1>LESS Styling in ReactJS Demo</h1>

                  <div id="app"></div>
            </body>
</html>
```

Output:



After Clicking the button "Click to Flip"



The App class is responsible for rendering most of the application, including various example Flippers and the button used to start the flip animation.

The getInitialState method sets up the App's state so that the tiles are not flipped by default. The flip method inverts the flipped state property, and the render method actually draws the application. In it, we're rendering two Flippers: one to flip horizontally and the other to flip vertically. There's also a button which is used to trigger the flip animation.

The Flipper class is responsible for drawing the various divs and setting each of the classes used in the animation to flip from front to back. You can see in the render method that there are also two more classes referenced: Front and Back. These classes contain what's shown on the front of the tile in the unflipped state, and what's on the back, which is shown when the tile is flipped.

The Front and Back classes are responsible for showing the appropriate content based on the flipped status of the parent Flipper class. The two classes are practically identical, with only a single class name separating the two, but I felt that they should be separated for readability's sake.

The children prop is a special prop which contains the child content of the class. When we're rendering each of the classes in the Flipper class above, you'll see the content for the two classes to be "the front!" and "the back!", respectively. That content gets placed in the children prop, which we're rendering in each of the tile classes render methods.

There are a couple of fancy CSS rules that are in play here.

- **transform-style:** The transform-style rule unsurprisingly indicates a style for the transform.
- **perspective:** This defines where the orientation and distance the viewer has of the 3D transformation. That sounds a little strange, but basically, it defines the number of pixels the rendered element is from the view.
- **backface-visibility:** This property indicates the visibility status of the reverse side of elements. This makes no sense for static elements on a page, but in our case, it's important to hide the reverse side so we don't see a flipped version of the tile. In our example, the text "FRONT FACE" is printed on both the front and back of the tile. If we set this property to hidden, only the front side is visible.

Performance optimizations

1. PureRenderMixin

You are already aware of the fact React.js is fast, but did you know there is a way to speed up the rendering of React components even more?
The PureRenderMixin does a shallow comparison on our props and state values. It compares the existing value of prop or state to new values.

Behind the scenes the PureRenderMixin is extending the lifecycle method shouldComponentUpdate. It determines if the render method is callable and your component will re-render.

Why PureRenderMixin?

The PureRenderMixin can save you a whole bunch of time because it means you don't have to worry about littering your components with if statements to check if anything has changed, the mixin handles everything behind the scenes for us automatically.

The mixin in simple terms is doing a check like this:

```
shouldComponentUpdate: function() {
      return !!(this.props === nextProps);
   }
```

Limitations of PureRenderMixin

PureRenderMixin does not compare deeply nested data structures. So if you supply an object to a property, it will not update and call the render function when you expected it too. Unless of course your data structure is immutable which you can use React's immutability helpers.

You also cannot render dynamic markup. The PureRenderMixin expects the same props/state and markup being rendered inside of your render method. This is why the plugin is called "PureRender" because it expects the render method to always be the same.

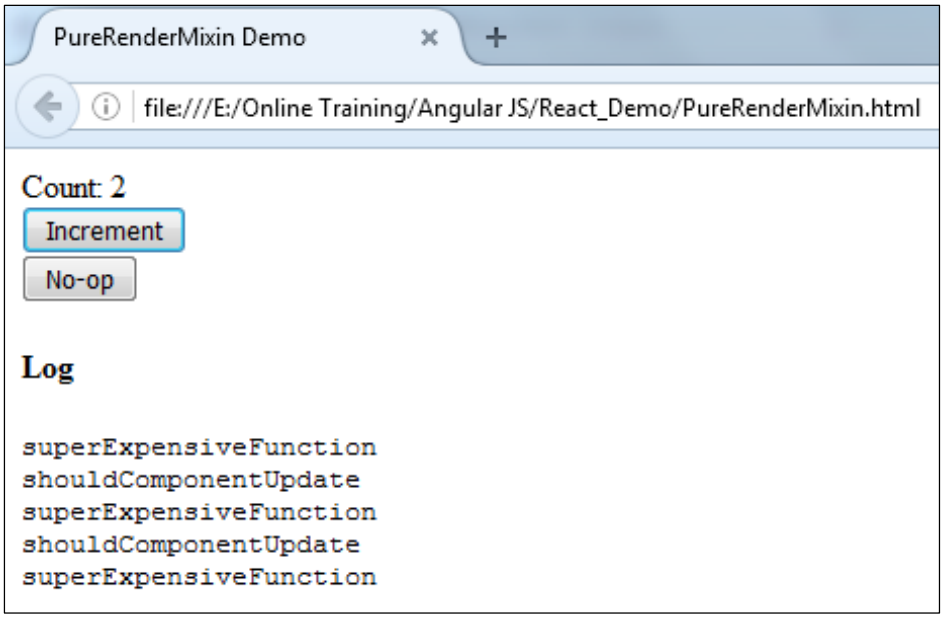
There are two reasons why you may want to optimize with shouldComponentUpdate.

1. You have used profiler tools and it is telling you that there is significant wasted time rendering for nothing.
2. You are able to make strong guarantees about when a component can and cannot change.

Example:

```
<!DOCTYPE html>
<html>
<head>
             <script src="https://fb.me/react-0.14.3.js"></script>
             <script src="https://fb.me/react-dom-0.14.3.js"></script>
             <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>JS Bin</title>
</head>
<body>
  <div id="container"></div>
  <h4>Log</h4>
  <pre id="logoutput"></pre>
  <script type="text/jsx">
  function log(msg) {
  document.getElementById('logoutput').innerHTML += `${msg}\n`;
}

function superExpensiveFunction(input) {
  log('superExpensiveFunction');
  return input;
}

var MyComponent = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    }
  },

  shouldComponentUpdate: function(nextProps, nextState) {
    log('shouldComponentUpdate');
    return nextState.count !== this.state.count;
  },

  onIncrementClick: function() {
    this.setState({
      count: this.state.count + 1
    });
  },

  onNoopClick: function() {
    this.setState({
      notusedstate: Math.random(),
    });
  },
```

```
  render: function() {
    var output = superExpensiveFunction(this.state.count);
    return (
      <div>
        <div>Count: {output}</div>
        <div><button onClick={this.onIncrementClick}>Increment</button></div>
        <div><button onClick={this.onNoopClick}>No-op</button></div>
      </div>
    );
  }
});

ReactDOM.render(<MyComponent />,document.getElementById('container')
);
</script>
</body>
</html>
```



Let's look at how shouldComponentUpdate is used in React's source. It reads, perform the update unless shouldComponentUpdate is implemented and returns false. If shouldComponentUpdate returns false, React acts as if the component's render output is the same as the previous pass and returns early. Notice that by bypassing render, the component's children are also skipped as well as some lifecycle hooks. Using forceUpdate does not check shouldComponentUpdate.

Example 2: Parting Caution

Incorrect usage of shouldComponentUpdate can lead to more trouble than it's worth. Think of shouldComponentUpdate as a hint that makes React components smarter. However, if this hint is wrong (or ever becomes wrong), the component may stop working completely.

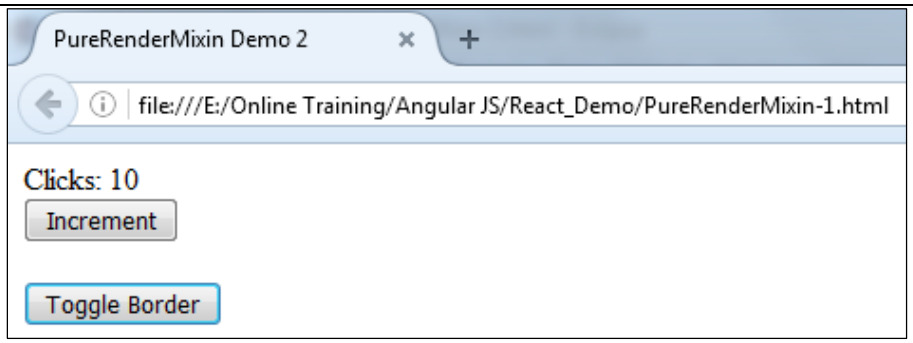
As an exercise, uncomment shouldComponentUpdate in the following Counter component and notice that Toggle Border does not work anymore.

```
<!DOCTYPE html>
<html>
```

```
<head>
<script src="https://fb.me/react-0.14.3.js"></script>
             <script src="https://fb.me/react-dom-0.14.3.js"></script>
             <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
  <meta charset="utf-8">
  <title>PureRenderMixin Demo 2</title>
</head>
<body>
  <div id="container"></div>
       <script type="text/jsx">
function shallowEqual(objA, objB) {
  if (objA === objB) {
    return true;
  }
  var key;
  // Test for A's keys different from B.
  for (key in objA) {
    if (objA.hasOwnProperty(key) &&
        (!objB.hasOwnProperty(key) || objA[key] !== objB[key])) {
      return false;
    }
  }
  // Test for B's keys missing from A.
  for (key in objB) {
    if (objB.hasOwnProperty(key) && !objA.hasOwnProperty(key)) {
      return false;
    }
  }
  return true;
}

var Counter = React.createClass({
  getInitialState: function() {
    return {
      count: 0
    };
  },

  onCounterClick: function(evt) {
    this.setState({
      count: this.state.count + 1
    });
  },

//   shouldComponentUpdate: function(nextProps, nextState) {
//     return (
//       nextState.count !== this.state.count ||
//       shallowEqual(nextProps, this.props)
//     );
//   },

  render: function() {
    console.log('rendering');
```

```
    return (
      <div {...this.props}>
        <div>Clicks: {this.state.count}</div>
        <button onClick={this.onCounterClick}>Increment</button>
      </div>
    );
  }
});

var CounterWrapper = React.createClass({
  getInitialState: function() {
    return {
      showborder: false,
    };
  },

  onToggleBorderClick: function() {
    this.setState({
      showborder: !this.state.showborder,
    });
  },

  render: function() {
    var style = {
      marginBottom: '20px'
    };
    if (this.state.showborder) {
      style.border = '1px solid #64994A';
    }
    return (
      <div>
        <Counter style={style} />
        <button onClick={this.onToggleBorderClick}>Toggle Border</button>
      </div>
    );
  }
});

ReactDOM.render(<CounterWrapper />,document.getElementById('container')
);

      </script>
</body>
</html>
```

Output:



2. Expensive DOM Manipulations

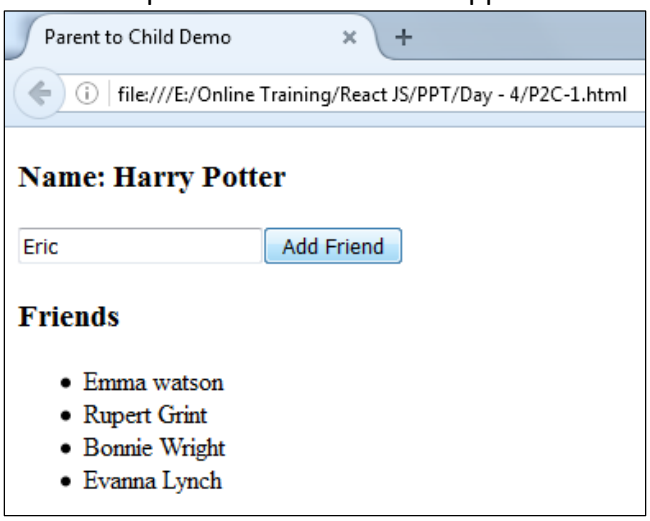
a. Document Object Model

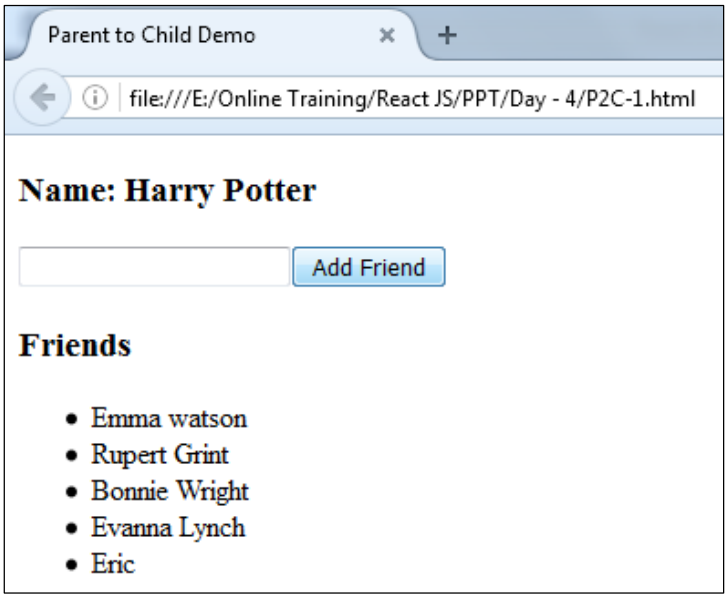
DOM is a programming interface for HTML or XML which provides a structured way to present a document. DOM represents documents in a logical manner. Usually this is mapped with a tree structure. In DOM documents are modeled with objects. It represents not only the structure of the document, but also the behavior. So DOM represents objects. This provides a way to represent object in HTML or XML not a way to persist objects in HTML or XML. DOM provides a API which is language independent. JavaScript is the most common and widely used language to access and manipulate the DOM.

b. Virtual DOM

React's solution for expensive DOM manipulations is Virtual DOM. V-DOM is a lightweight representation of the real DOM. V-DOM is more like a fake DOM. It's an in memory representation of real DOM within React.JS. V-DOM clones real DOM and it's available as a large, single JavaScript object. In React once the application loads, it takes the real DOM and creates the representation of V-DOM.

Let's take a scenario of DOM manipulation within a React application.





This depicts a few DOM manipulations where new nodes are being added. In this application once we type the Friends name and hit 'Add Friend' React will take over the control and it will execute a full re-render within the V-DOM. This won't be expensive since React is not performing any changes on real DOM. After re-rendering processes finishes with the V-DOM, React compares the V-DOM with real DOM.

In order to do this comparison React uses a special algorithm called the diff algorithm. Tree diff is so efficient in comparing two different DOMs. This takes place within few milliseconds. If there are any changes after the comparison, like in this case where few nodes gets added, React immediately patches changes from V-DOM to real DOM. This process doesn't require expensive traversing since React already knows which nodes got changed. Therefore this solution will be more performant than using regular DOM. With React when mutations occur, they can be batched together. This will increase the performance since a minimal number of traversing is needed to change the state of a user interface.

Concept of Virtual DOM is not specific to React. There can be many implementations of the idea of V-DOM. In fact this library implements the V-DOM concept and the diffing algorithm for efficient change detection and patches.

3. Performance Tools

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a shouldComponentUpdate hook where you can add optimization hints to React's diff algorithm.

By using React, you instantly get some performance gains without any extra work:

Because React handles all DOM manipulations, you largely avoid issues regarding DOM parsing and layout. Behind the scenes, React maintains a virtual DOM in JavaScript, which it can use to quickly determine the minimal changes needed to bring the document to the desired state.

Because a React component's state is stored in JavaScript, we avoid accessing the DOM. A classic performance issue is accessing the DOM at inopportune moments, which can result in issues like forced synchronous layouts

In addition to giving you an overview of your app's overall performance, ReactPerf is a profiling tool that tells you exactly where you need to put these hooks.

ReactJS – Context

One of React's biggest strengths is that it's easy to track the flow of data through your React components. When you look at a component, you can easily see exactly which props are being passed in which makes your apps easy to reason about.

Occasionally, you want to pass data through the component tree without having to pass the props down manually at every level.

When working with React in a deeply nested hierarchy, often times we will find ourselves in a situation where we need to pass down props from the parent as is to the children and the grandchildren. Although passing down properties is great, but writing code in such a manner encourages relying on memory to pass down the necessary props.

React has something called as Child contexts that allows a parent element to specify a context aka a set of properties which can be accessed from all of its children and grandchildren via the this.context object.

There are 2 aspects to how you would go about implementing this in your code.

1. In the parent, the get Child Context method is used to return an object that is passed own as the child context. The prop types of all the keys in this object must also be defined in the child ContextTypes property of the parent.
2. Any child that needs to consume the properties passed down from its parents must whitelist the properties it wants to access via the 'contextTypes' property.

One may argue that this makes the code a bit more verbose, because now we end up creating an additional key contextTypes on every child component. However, the reality is that you can specify the 'contextTypes' property on a mixin. That way you can avoid rewriting the whitelisting boilerplate code by simply using the mixin on all the child components that need to access those properties from its context.

Context is an advanced and experimental feature. Most applications will never need to use context. Especially if you are just getting started with React, you likely do not want to use context. Using context will make your code harder to understand because it makes the data flow less clear. It is similar to using global variables to pass state through your application.
If you have to use context, use it sparingly.

Regardless of whether you're building an application or a library, try to isolate your use of context to a small area and avoid using the context API directly when possible so that it's easier to upgrade when the API changes.

The root of the context must define how the context looks like and what type each element in it has. In our case, Parent is a root. It's also worth noting that the component which created the context, cannot access it directly.

Why not to use contexts?

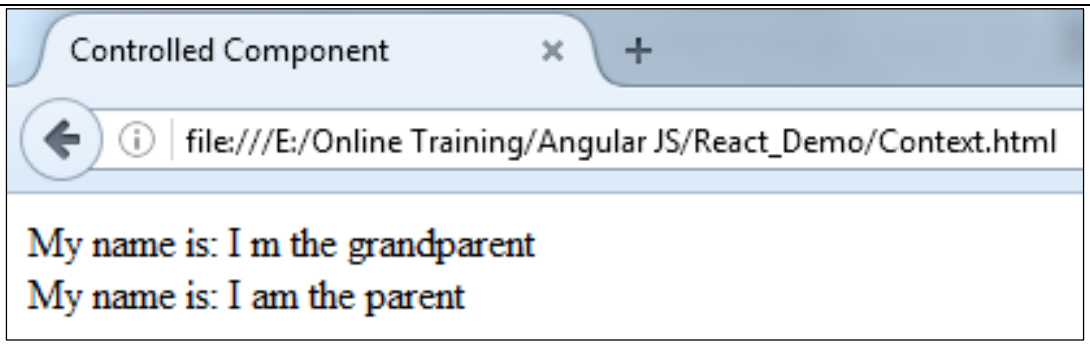
Contexts aren't the best way pass domain related models. They couple your code and make it less reusable. The readability of the code may also hurt.

Contexts feature is still under development. React developers don't guarantee that the behavior and API will stay the same. You should try to isolate parts of your systems using contexts.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Controlled Component</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
 <div id="content"></div>
 <script type="text/jsx">
var Grandparent = React.createClass({

    childContextTypes: {
         foo: React.PropTypes.string.isRequired
    },

    getChildContext: function() {
         return { foo: "Im the grandparent" };
    },
```

```
    render: function() {
         return <Parent />;
    }
});


var Parent = React.createClass({

    childContextTypes: {
         bar: React.PropTypes.string.isRequired
    },

    getChildContext: function() {
         return { bar: "I am the parent" };
    },

    render: function() {
         return <Child />;
    }
});

var Child = React.createClass({

    contextTypes: {
        foo: React.PropTypes.string.isRequired,
        bar: React.PropTypes.string.isRequired
    },

    render: function() {
        return (
          <div>
            <div>My name is: {this.context.foo}</div>
            <div>My name is: {this.context.bar}</div>
          </div>
        )
    }
});

// Finally you render the grandparent
ReactDOM.render(<Grandparent />, document.getElementById('content'));
</script>
</body>
</html>

Output:
```



Pass down props using React childContextTypes in a deeply nested component tree:

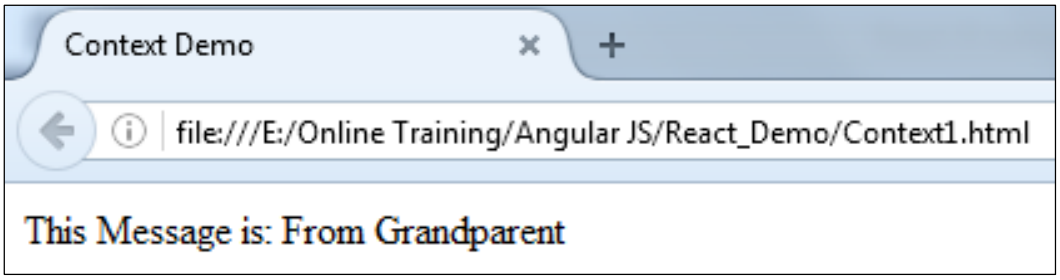
In our previous Example on React childContextTypes we saw how React lets you accumulate props along a nested hierarchy. There is one more characteristic of childContextTypes that is extremely handy.

Properties specified in childContextTypes are propagated along the hierarchy even if none of the intermediate components in the hierarchy implement either the getChildContext method or have the childContextTypes property.

Example:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>Context Demo</title>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
      <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
      <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/JSXTransformer.js"></scrip
t>
</head>
<body>
 <div id="content"></div>
 <script type="text/jsx">
var Grandparent = React.createClass({
    childContextTypes: {
         message: React.PropTypes.string.isRequired
    },

    getChildContext: function() {
         return { message: "From Grandparent" };
    },

    render: function() {
         return <Parent />;
    }
```

```
});

var Parent = React.createClass({
    // Notice how this component does not have either the
    // getChildContext method or the childContextTypes map.
    render: function() {
        return <Child />;
    }
});

var Child = React.createClass({
    contextTypes: {
        message: React.PropTypes.string.isRequired
    },

    render: function() {
        return <div>This Message is: {this.context.message}</div>;
    }
});

// Outputs: "This Message is: From Grandparent"
ReactDOM.render(<Grandparent />, document.getElementById('content'));
</script>
</body>
</html>
```



React Router

React Router is a complete routing library for React.

React Router keeps your UI in sync with the URL. It has a simple API with powerful features like lazy code loading, dynamic route matching, and location transition handling built right in. Make the URL your first thought, not an after-thought.

React Router is a powerful routing library built on top of React that helps you add new screens and flows to your application incredibly quickly, all while keeping the URL in sync with what's being displayed on the page.

Single-page apps are different from the more traditional multi-page apps that you see everywhere. The biggest difference is that navigating a single-page app doesn't involve going to

an entirely new page. Instead, your pages (commonly known as views in this context) typically load inline within the same page itself:

When you are loading content inline, things get a little challenging. The hard part is not loading the content itself. That is relatively easy. The hard part is making sure that single-page apps behave in a way that is consistent with what your users are used to. More specifically, when users navigate your app, they expect that:

The URL displayed in the address bar always reflects the thing that they are viewing.
They can use the browser's back and forward buttons...successfully.
They can navigate to a particular view (aka deep link) directly using the appropriate URL.
With multi-page apps, these three things come for free. There is nothing extra you have to do for any of it. With single-page apps, because you aren't navigating to an entirely new page, you have to do real work to deal with these three things that your users expect to just work. You need to ensure that navigating within your app adjusts the URL appropriately. You need to ensure your browser's history is properly synchronized with each navigation to allow users to use the back and forward buttons. If users bookmark a particular view or copy/paste a URL to access later, you need to ensure that your single-page app takes the user to the correct place.

To deal with all of this, you have a bucket full of techniques commonly known as routing. Routing is where you try to map URLs to destinations that aren't physical pages such as the individual views in your single-page app. That sounds complicated, but fortunately there are a bunch of JavaScript libraries that help us out with this. One such JavaScript library is “React Router”. React Router provides routing capabilities to single-page apps built in React, and what makes it nice is that extends what you already know about React in familiar ways to give you all of this routing awesomeness.

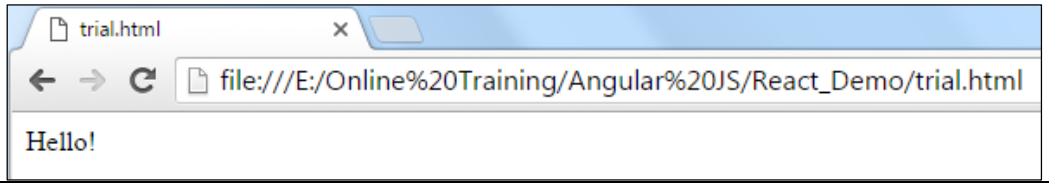
Building the App

The first thing we need to do is get the boilerplate markup and code for our app up and running. Create a new HTML document and add the following content into it:

This starting point is almost the same as what you've seen for all of our other examples. This is just a nearly blank app that happens to load the React and React-DOM libraries. If you preview what you have in your browser, you'll see a very lonely Hello! Displayed.

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>
  <script src="https://npmcdn.com/react-dom@15.3.0/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>

  <style>  </style>
</head>
<body>

  <div id="container">  </div>
  <script type="text/babel">
    var destination = document.querySelector("#container");

    ReactDOM.render(
      <div>
        Hello!
      </div>,
      destination
    );
  </script>
</body>
</html>
```

Output:



React Router isn't a part of React itself, we need to add a reference to it. In our markup, find where we have our existing script references and add the following highlighted line:

<script src="https://npmcdn.com/react-router@2.4.0/umd/ReactRouter.min.js"></script>
<script src="https://npmcdn.com/react-router/umd/ReactRouter.min.js"></script>

By adding any of these line, we ensure the React Router library is loaded alongside the core React, ReactDOM, and Babe libraries. At this point, we are in a good state to start building our app and taking advantage of the sweet functionality React Router brings to the table.

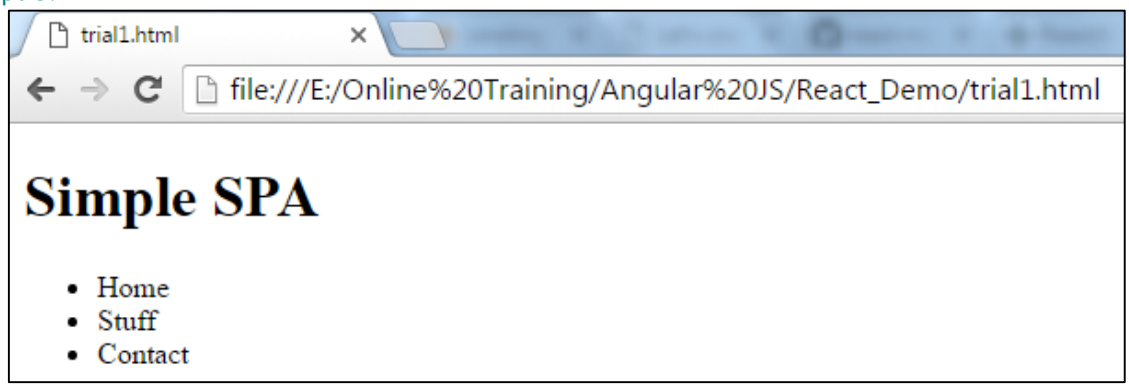
Displaying the Initial Frame

When building a single-page app, there will always be a part of your page that will remain static. This static part, also referred to as an app frame, could just be one invisible HTML element that acts as the container for all of your content, or could include some additional visual things like a header, footer, navigation, etc. In our case, our app frame will involve our navigation header and an empty area for content to load in. To display this, we are going to create a component that is going to be responsible for this.

```
<!DOCTYPE html>
<html>
<head>
  <script src="https://npmcdn.com/react@15.3.0/dist/react.js"></script>
  <script src="https://npmcdn.com/react-dom@15.3.0/dist/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.23/browser.min.js"></script>

  <style>

  </style>
</head>

<body>

  <div id="container">

  </div>

  <script type="text/babel">
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">

        </div>
      </div>
    )
  }
});
    var destination = document.querySelector("#container");

    ReactDOM.render(
      <div>
       <App/>
      </div>,
      destination
    );
```

```
  </script>
</body>
</html>
```

Output:



Going a bit deeper, what we've done is just create a component called App and display it via our ReactDOM.render call. The important thing to call out is that there is nothing React Router specific here. ABSOLUTELY NOTHING!
(Explain with trial1.html)

Let's fix that by throwing React Router into the mix. Replace the contents of your ReactDOM.render call with the following:

```
ReactDOM.render(
  <ReactRouter.Router>
    <ReactRouter.Route path="/" component={App}>

    </ReactRouter.Route>
  </ReactRouter.Router>,
  destination
);
```

The Router component is part of the React Router API, and its job is to deal with all of the routing-related logic our app will need. Inside this component, we specify what is known as the routing configuration. That is a fancy term that people use to describe the mapping between URLs and the views. The specifics of that are handled by another component called Route:
(Explain with trial2.html)

The Route component takes several props that help define what to display at what URL. The path prop specifies the URL we are interested in matching. In this case, it is the root aka /. The component prop allows you to specify the name of the component you wish to display.

Displaying the Home Page

The way React Router provides you with all of this routing functionality is by mimicking concepts in React you are already familiar with – namely components, props, and JSX.
Now, it's time to go even further. What we want to do next is define the content that we will display as part of our home view.

To do this, let’s create a component called Home that is going to contain the markup we want to display. Just above where we have your App component defined. **(Explain Trial3.html)**

As you can see, our Home component doesn't do anything special. It just returns a blob of HTML. Now, what we want to do is display the contents of our Home component when the page loads. This component is the equivalent of our app's "home page". The way we do this is simple. Inside our App component, we have a div with a class value of content. We are going to load our Home component inside there.

You see our navigation header, and then you can see the contents of our Home component. While this approach works, it is actually wrong thing to do. It is wrong because it complicates our desire to load other pieces of content as the user is navigating around our app. We've essentially hard-coded our app to only display the Home component. That's a problem, but we'll come back to that in a little bit.

Before we continue making progress on our app, let's take a short break and make some stylistic improvements to what we have so far. Right now, our app looks very plain...and like something straight out of the 1800's. To fix this, we are going to rely on our dear old friend, CSS. Inside the style tag, let’s go ahead and add few style rules:

We are using CSS in its markup form. We aren't doing the inline style object approach that we've used in the past. The reason has to do with convenience. Our components aren't going to be re-used outside of our particular app, and we really want to take advantage of CSS inheritance to minimize duplicated markup. Otherwise, if we didn't use regular CSS, we'll end up with a bunch of giant style objects defined for almost every element in our markup. That would make even the most patient among us annoyed when reading the code.
Anyway, once you have added all of this CSS, our app will start to look much better:

Avoiding the ReactRouter Prefix
We have just one more cleanup related task before we return to our regularly scheduled programming. Have you noticed that every single time we call something defined by the React Router API, we prefix that something with the word ReactRouter?

```
<ReactRouter.Router>
  <ReactRouter.Route path="/" component={App}>

  </ReactRouter.Route>
</ReactRouter.Router>
```

That is a bit verbose to have to repeat for every API call we make, and this is going to be more of a problem as we dive further into the React Router API and use more things from inside it.

The fix for this involves using a new ES6 trick where you can manually specify which values will automatically get prefixed. Towards the top of your script tag, add the following:

```
var { Router,
    Route,
    IndexRoute,
    IndexLink,
    Link } = ReactRouter;
```

Once you've added this code, every time you use one of the values defined inside the brackets, the prefix ReactRouter will automatically be added for you when your app runs. This means, you can now go back to your ReactDOM.render method and remove the ReactRouter prefix from our Router and Route component instances:

```
ReactDOM.render(
 <Router>
  <Route path="/" component={App}>

  </Route>
 </Router>,
 destination
);
```

Displaying the Home Page Correctly

We ended a few sections ago by saying that the way we currently have our home page displayed. While you get the desired result when our page loads, this approach doesn't really make it easy for us to load anything other than the home page when users navigate around. The call to our Home component is hard coded inside App.

The correct solution involves letting React Router handle which component to call depending on what your current URL structure is. This involves nesting Route components inside Route components to better define the URL-to-view mapping further. Let’s Go back to our ReactDOM.render method, and make the following change:

```
ReactDOM.render(
 <Router>
  <Route path="/" component={App}>
   <IndexRoute component={Home}/>
  </Route>
 </Router>,
 destination);
```

Inside our root Route element, we are defining another route element of type IndexRoute (more on who this is in a second!) and setting its view to be our Home component. There is one more change we need to make. Inside our App component, let's remove the call to the Home component and replace it with the following highlighted line:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
          <li>Stuff</li>
          <li>Contact</li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

The difference this time is that we are displaying the home content properly in a way that doesn't prevent other content from being displayed instead. This is because of two things:

What gets displayed inside App is controlled by the result of **this.props.children** instead of a hard-coded component.
Our Route element inside **ReactDOM.render** contains an IndexRoute element whose sole purpose for existing is to declare which component will be displayed when your app initially loads.

Creating the Navigation Links
Right now, we just have our frame and home view setup. There isn't really anything else for a user to do here outside of just seeing what we have set as the home "page". Let's fix that by creating some navigation links. More specifically, let's linkify the navigation elements we already have:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li>Home</li>
```

```
        <li>Stuff</li>
        <li>Contact</li>
       </ul>
       <div className="content">
        {this.props.children}
       </div>
      </div>
    )
  }
});
```

The way you specify navigation links in React Router isn't by directly using the tried and tested a tag and throwing in a path via the href attribute. Instead, you specify your navigation link using React Router's Link components that are similar to a tags but offer a lot more functionality. To see the Link component in action, go ahead and modify our existing navigation elements to look like the following highlighted lines:

```
var App = React.createClass({
  render: function() {
    return (
      <div>
        <h1>Simple SPA</h1>
        <ul className="header">
          <li><Link to="/">Home</Link></li>
          <li><Link to="/stuff">Stuff</Link></li>
          <li><Link to="/contact">Contact</Link></li>
        </ul>
        <div className="content">
          {this.props.children}
        </div>
      </div>
    )
  }
});
```

Notice what have done here. Our Link components specify a prop called to. This prop specifies the value of the URL we will display in the address bar. Indirectly, it also specifies the location we will be telling React Router we are virtually navigating to. Our Home link takes users to the root (/), the Stuff link takes users to a location called stuff, and the Contact link takes users to a location called contact.

If you preview your page and click on the links (which will now be visible because the CSS for them will have kicked in), you won't see anything new display. You will just see your Home content because that is all that we had specified earlier. With that said, you can see the URLs

updating in the address bar. You'll see your current page followed by a #/contact, #/stuff, or #/ depending on which of the links you clicked.

What we have just added are the Stuff and Contact components that simply render out HTML. All that remains is for us to update our routing configuration to include these two components and display them at the appropriate URL.

In our ReactDOM.render method, go ahead and add the following two highlighted lines:

```
ReactDOM.render(
 <Router>
  <Route path="/" component={App}>
   <IndexRoute component={Home}/>
   <Route path="stuff" component={Stuff} />
   <Route path="contact" component={Contact} />
  </Route>
 </Router>,
 destination
);
```

All we are doing here is updating our routing logic to display the Stuff component if the URL contains the word stuff and to display the Contact component if the URL contains the word contact. If you preview your page now, click on the Stuff and Contact links. If everything worked out fine, you'll see these views get loaded inside our app frame when you navigate to them.

Creating Active Links

The last thing we are going to tackle is something that greatly increases the usability of our app. Depending on which page you are currently displaying, we are going to highlight that link with a blue background.

The way you accomplish this in React Router is by setting a prop called activeClassName on your Link instances with the name of the CSS class that will get set when that link is currently active. To make this happen, go back to your App component and make the highlighted changes:

```
var App = React.createClass({
 render: function() {
  return (
   <div>
    <h1>Simple SPA</h1>
    <ul className="header">
     <li><IndexLink to="/" activeClassName="active">Home</IndexLink></li>
     <li><Link to="/stuff" activeClassName="active">Stuff</Link></li>
     <li><Link to="/contact" activeClassName="active">Contact</Link></li>
    </ul>
    <div className="content">
```

```
      {this.props.children}
     </div>
    </div>
  )
 }
});
```

We specify the activeClassName prop and set it to a value of active. This ensures that whenever a link is clicked (and its path becomes active), the link element's class attribute at runtime gets set to a value of active. To ensure our active links are styled differently, go ahead and add the following CSS:

```
.active {
 background-color: #0099FF;
}
```

Example: Before we go further, let's take a look at the following example:

Route Configuration

A route configuration is basically a set of instructions that tell a router how to try to match the URL and what code to run when it does.

URL	Components
/	App -> Dashboard
/about	App -> About
/inbox	App -> Inbox
/messages/:id	App -> Inbox -> Message

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8" />
    <title>React Config Demo</title>
    <style type="text/scss">
.app {
  display: flex;
}

aside.primary-aside {
  width: 200px;
}

main {
  flex: 1;
}
</style>
```

```
            <script
src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react.js"></script>
       <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.3/react-
dom.js"></script>
       <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
       <script src="https://cdnjs.cloudflare.com/ajax/libs/react-
router/2.7.0/ReactRouter.min.js"></script>

 <script type="text/babel">
var { Router, Route, IndexRoute, Link, browserHistory } = ReactRouter
const App = React.createClass({
  render() {
    return (
      <div>
        <h1>React Router Config Demo</h1>
        <ul>
          <li><Link to="/about">About</Link></li>
          <li><Link to="/inbox">Inbox</Link></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
})

const About = React.createClass({
  render() {
    return <h3>About</h3>
  }
})

const Inbox = React.createClass({
  render() {
    return (
      <div>
        <h2>Inbox</h2>
        {this.props.children || "Welcome to your Inbox"}
      </div>
    )
  }
})

const Message = React.createClass({
  render() {
    return <h3>Message {this.props.params.id}</h3>
  }
})

ReactDOM.render((
  <Router>
    <Route path="/" component={App}>
      <Route path="about" component={About} />
      <Route path="inbox" component={Inbox}>
        <Route path="messages/:id" component={Message} />
      </Route>
```

```
    </Route>
  </Router>
), document.getElementById('content'))

</script>
</head>

<body>
<div id="content"></div>
</body>
</html>
```

Output: