

ReactJS Introduction

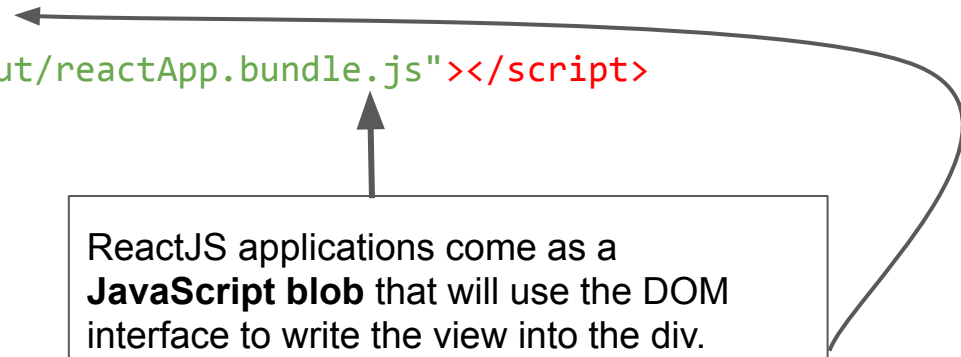
Mendel Rosenblum

ReactJS

- JavaScript framework for writing the web applications
 - Like AngularJS - Snappy response from running in browser
 - Less **opinionated**: only specifies rendering view and handling user interactions
- Uses Model-View-Controller pattern
 - View constructed from Components using pattern
 - Optional, but commonly used HTML templating
- Minimal server-side support dictated
- Focus on supporting for programming in the large and single page applications
 - Modules, reusable components, testing, etc.

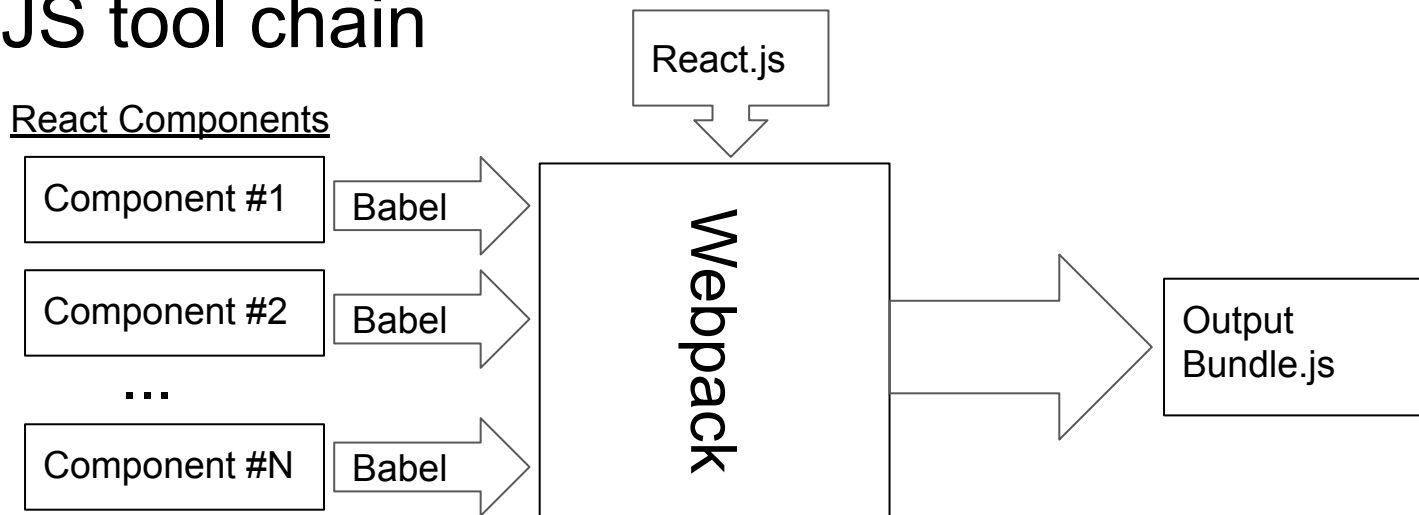
ReactJS Web Application Page

```
<!doctype html>
<html>
  <head>
    <title>CS142 Example</title>
  </head>
  <body>
    <div id="reactapp"></div>
    <script src="./webpackOutput/reactApp.bundle.js"></script>
  </body>
</html>
```



ReactJS applications come as a **JavaScript blob** that will use the DOM interface to write the view into the div.

ReactJS tool chain



Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

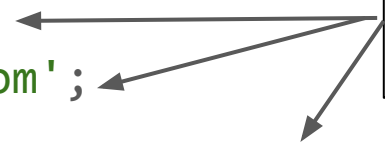
Webpack - Bundle modules and resources (CSS, images)

Output loadable with single script tag in any browser


reactApp.js - Render element into browser DOM

```
import React from 'react';  
import ReactDOM from 'react-dom';  
import ReactAppView from './components/ReactAppView';  
let viewTree = React.createElement(ReactAppView, null);  
let where = document.getElementById('reactapp');  
ReactDOM.render(viewTree, where);
```

ES6 Modules - Bring in
React and web app React
components.



Renders the tree of React elements (single component
named **ReactAppView**) into the browser's DOM at the
div with id=reactapp.



components/ReactAppView.js - ES6 class definition

```
import React from 'react';  
  
class ReactAppView extends React.Component {  
  constructor(props) {  
    super(props);  
    ...  
  }  
  render() { ...  
};  
  
export default ReactAppView;
```

← Inherits from React.Component. props is set to the attributes passed to the component.

← Require method render() - returns React element tree of the Component's view.

ReactAppView render() method

```
render() {  
  let label = React.createElement('label', null, 'Name: ');  
  let input = React.createElement('input',  
    { type: 'text', value: this.state.yourName,  
      onChange: (event) => this.handleChange(event) });  
  let h1 = React.createElement('h1', null,  
    'Hello ', this.state.yourName, '!');  
  return React.createElement('div', null, label, input, h1);  
}
```

```
<div>  
  <label>Name: </label>  
  <input type="text" ... />  
  <h1>Hello {this.state.yourName}!</h1>  
</div>
```

Returns element tree with div (label, input, and h1) elements

Name:

Hello !

ReactAppView render() method w/o variables

```
render() {  
  return React.createElement('div', null,  
    React.createElement('label', null, 'Name: '),  
    React.createElement('input',  
      { type: 'text', value: this.state.yourName,  
        onChange: (event) => this.handleChange(event) }),  
    React.createElement('h1', null,  
      'Hello ', this.state.yourName, '!')  
  );  
}
```



Use JSX to generate calls to createElement

```
render() {  
  return (  
    <div>  
      <label>Name: </label>  
      <input  
        type="text"  
        value={this.state.yourName}  
        onChange={(event) => this.handleChange(event)}  
      />  
      <h1>Hello {this.state.yourName}!</h1>  
    </div>  
  );  
}
```

- JSX makes building tree look like templated HTML embedded in JavaScript.

Component state and input handling

```
import React from 'react';  
  
class ReactAppView extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {yourName: ""};  
  }  
  handleChange(event) {  
    this.setState({ yourName: event.target.value });  
  }  
  ...  
}
```



Make `<h1>Hello {this.state.yourName}!</h1>`
work

- Input calls to `setState` which causes React to call `render()` again

One way binding: Type 'D' Character in input box

- JSX statement: `<input type="text" value={this.state.yourName} onChange={(event) => this.handleChange(event)} />`

Triggers `handleChange` call with `event.target.value == "D"`

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

`this.state.yourName` is changed to "D"

- React sees state change and calls render again:
- Feature of React - highly efficient re-rendering

Name:

Hello D!

Calling React Components from events: A problem

```
class ReactAppView extends React.Component {  
  ...  
  handleChange(event) {  
    this.setState({ yourName: event.target.value });  
  }  
  ...  
}
```

Understand why:

```
<input type="text" value={this.state.yourName} onChange={this.handleChange} />
```

Doesn't work!

Calling React Components from events workaround

- Create instance function bound to instance

```
class ReactAppView extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {yourName: ""};  
    this.handleChange = this.handleChange.bind(this);  
  }  
  handleChange(event) {  
    this.setState({ yourName: event.target.value });  
  }  
}
```

Calling React Components from events workaround

- Using public fields of classes with arrow functions

```
class ReactAppView extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = {yourName: ""};  
  }  
  handleChange = (event) => {  
    this.setState({ yourName: event.target.value });  
  }  
  ...  
}
```

Calling React Components from events workaround

- Using arrow functions in JSX

```
class ReactAppView extends React.Component {  
  ...  
  handleChange(event) {  
    this.setState({ yourName: event.target.value });  
  }  
  render() {  
    return (  
      <input type="text" value={this.state.yourName}  
        onChange={(event) => this.handleChange(event)} />  
    );  
  }  
}
```

A digression: camelCase vs dash-case

Word separator in multiword variable name

- Use dashes: `active-buffer-entry`
- Capitalize first letter of each word: `activeBufferEntry`

Issue: HTML is case-insensitive but JavaScript is not.

ReactJS's JSX has HTML-like stuff embedded in JavaScript.

ReactJS: Use camelCase for attributes

AngularJS: Used both: dashes in HTML and camelCase in JavaScript!

Programming with JSX

- Need to remember: JSX maps to calls to `React.createElement`
 - Writing in JavaScript HTML-like syntax that is converted to JavaScript function calls
- `React.createElement(type, props, ...children);`
 - `type`: HTML tag (e.g. `h1`, `p`) or `React.Component`
 - `props`: attributes (e.g. `type="text"`) Uses camelCase!
 - `children`: Zero or more children which can be either:
 - String or numbers
 - A React element
 - An Array of the above

JSX templates must return a valid children param

- Templates can have JavaScript scope variables and expressions
 - `<div>{foo}</div>`
 - Valid if foo is in scope (i.e. if foo would have been a valid function call parameter)
 - `<div>{foo + 'S' + computeEndingString()}</div>`
 - Valid if foo & computeEndingString in scope
- Template must evaluate to a value
 - `<div>{if (useSpanish) { ... } }</div>` - Doesn't work: if isn't an expression
 - Same problem with "for loops" and other JavaScript statements that don't return values
- Leads to contorted looking JSX: Example: Anonymous immediate functions
 - `<div>{ (function() { if ...; for ..; return val; })() }</div>`

Conditional render in JSX

- Use JavaScript Ternary operator (?:)

```
<div>{this.state.useSpanish ? <b>Hola</b> : "Hello"}</div>
```

- Use JavaScript variables

```
let greeting;  
const en = "Hello"; const sp = <b>Hola</b>;  
let {useSpanish} = this.props;  
if (useSpanish) {greeting = sp} else {greeting = en};
```

```
<div>{greeting}</div>
```

Iteration in JSX

- Use JavaScript array variables

```
let listItems = [];  
for (let i = 0; i < data.length; i++) {  
    listItems.push(<li key={data[i]}>Data Value {data[i]}</li>);  
}  
return <ul>{listItems}</ul>;
```

- Functional programming

```
<ul>{data.map((d) => <li key={d}>Data Value {d}</li>)}</ul>
```

key= attribute improves efficiency of rendering on data change

Styling with React/JSX - lots of different ways

```
import React from 'react';
import './ReactAppView.css';

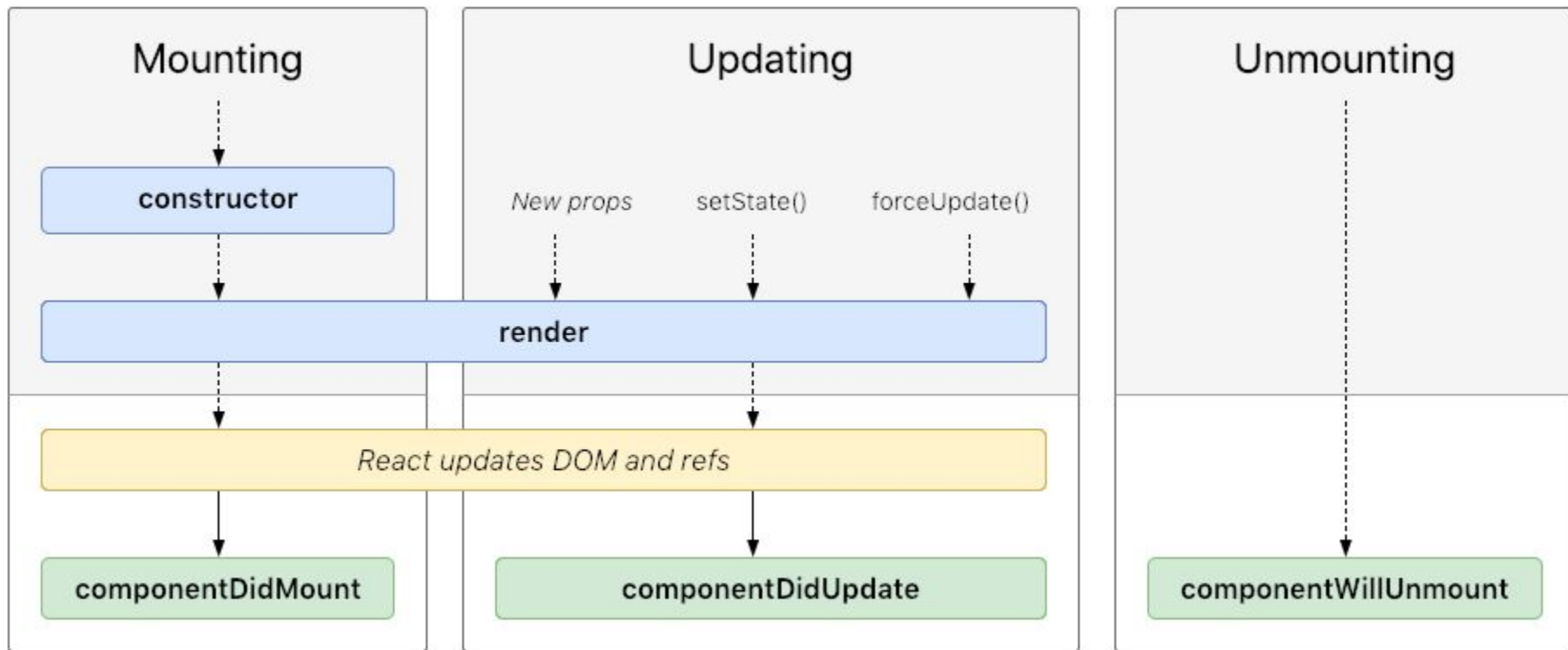
class ReactAppView extends React.Component {
  ...
  render() {
    return (
      <span className="cs142-code-name">
        ...
      </span>
    );
  }
}
```

Webpack can import CSS style sheets:

```
.cs142-code-name {
  font-family: Courier New, monospace;
}
```

Must use className= for HTML
class= attribute (JS keyword
conflict)

Component lifecycle and methods



Example of lifecycle methods - update UI every 2s

```
class Example extends React.Component {  
  ...  
  componentDidMount() {    // Start 2 sec counter  
    const incFunc =  
      () => this.setState({ counter: this.state.counter + 1 });  
    this.timerID = setInterval(incFunc, 2 * 1000);  
  }  
  
  componentWillUnmount() {    // Shutdown timer  
    clearInterval(this.timerID);  
  }  
  ...  
}
```

Stateless Components

- React Component can be function (not a class) if it only depends on props

```
function MyComponent(props) {  
  return <div>My name is {props.name}</div>;  
}
```

Or using destructuring...

```
function MyComponent({name}) {  
  return <div>My name is {name}</div>;  
}
```

- Much more concise than a class with render method
 - But what if you have one bit of state...

React Hooks - Add state to stateless components

- Inside of a "stateless" component add state: `useState(initialStateValue)`
 - `useState` parameter: `initialStateValue` - the initial value of the state
 - `useState` return value: An two element polymorphic array
 - 0th element - The current value of the state
 - 1st element - A set function to call (like `this.setState`)
- Example: a bit of state:
`const [bit, setBit] = useState(0);`
- How about lifecycle functions (e.g. `componentDidUpdate`, etc.)?
 - `useEffect(lifeCycleFunction, dependency array)`
 - `useEffect` parameter `lifeCycleFunction` - function to call when something changes

React Hooks Example - useState

```
import React, { useState } from 'react';  
function Example() {  
  const [count, setCount] = useState(0);  
  return (  
    <div>  
      <p>You clicked {count} times</p>  
      <button onClick={() => setCount(count + 1)}>  
        Click me  
      </button>  
    </div>  
  );  
}
```

React Hooks Example - useEffect Model fetching

```
import React, { useState, useEffect } from 'react';
function Example() {
  const [count, setCount] = useState(0);
  const [fetch, setFetch] = useState(false);
  useEffect(() => {setCount(modelFetch()); setFetch(true);}, [fetch]);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Communicating between React components

- Passing information from parent to child: Use props (attributes)

```
<ChildComponent param={infoForChildComponent} />
```

- Passing information from child to parent: Callbacks

```
this.parentCallback = (infoFromChild) =>  
  { /* processInfoFromChild */};
```

```
<ChildComponent callback={this.parentCallback}> />
```

- React Context (<https://reactjs.org/docs/context.html>)
 - Global variables for subtree of components