How React works

- The React application is made of components
- React takes the components and creates a JavaScript representation of the DOM (Virtual DOM)
- React renders the DOM based on the Virtual DOM
- Whenever data changes within a component, a new Virtual DOM is rendered and compared to the current Virtual DOM
- The changes are updated in the browser

CDN

React

```
<script crossorigin src="https://unpkg.com/react@16/umd/react.development.js">
</script>


<script crossorigin src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
```

Babel

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

```
<script type="text/babel">
    ...
</script>
```

Components

```
<script type="text/babel"> javascript
    class App extends React.Component {
        render() {
          return (
            // jsx
          )
    }
}
```

`render()`

- Renders the template to the element

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

- Render App to #app

`{ }`

- Treat whatever is inside of `{ }` as JavaScript

State

- State of data/UI of component
- A JSON object storing local data of the component
- Changing the state of a component makes it re-render

```
state = {
  name: 'react',
  isCool: true,
}
```

```
{ this.state.name } // react
```

Events

```
handleClick(e){
  console.log(e.target);
}
```

```
<button onClick={this.handleClick}>click</button>
```

`this`

```
handleClick(e){
  console.log(this.state);

}
```

- Error, `this` is out of scope, undefined
- We must manually bind `this`
- Use an arrow function

```
handleClick = e => {
  console.log(this.state);
}
```

- Arrow functions bind `this` to whatever `this` is outside the function
- For regular functions, `this` refers to the object that called the function
- For `=>` functions, `this` is bound *lexically*, using the enclosing function scope as its `this` value

Changing State

```
this.state.name = 'cool' // no!
```

- Do not change the state directly

`setState`

```
this.setState({
    name: 'epic',
});
```

- Pass in an object representing the state
- Asynchronous

Forms

```
 <form onSubmit={this.handleSubmit}>
    <input type="text" onChange={ this.handleChange }/>
    <button>Submit</button>
</form>
```

- Remember to `e.preventDefault()` form submits

Single Page Applications

- React apps are SPAs
- Only one HTML page is served on the browser
- React controls what the user sees

Components Continued

- Components can be defined by functions or classes

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- No state
- Receive data from props
- UI components

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- State
- Lifecycle hooks
- Container components

Props

- Pass data from parent components to child components

```
<Component name="Jay JO" likes="biking" drift="true"></Component>
```

```
{ props.name }, { props.likes }, { props.drift }
```

Lists

Container

```
state = { // for class component
  members: [
    {},
    ...
  ]
}
```

```
class
<Component prop={this.state.members}/>
function
<Component prop={members}/>
```

Component

```
const memberList = props.members.map(member => {
    return (
        //html
    );
});

{ memberList }
```

Conditional Output

- Use the Ternary operator

```
const coolList = members.map(member => {
    return member.cool == 'true' ? (
        // html
    ) : null;
});
```

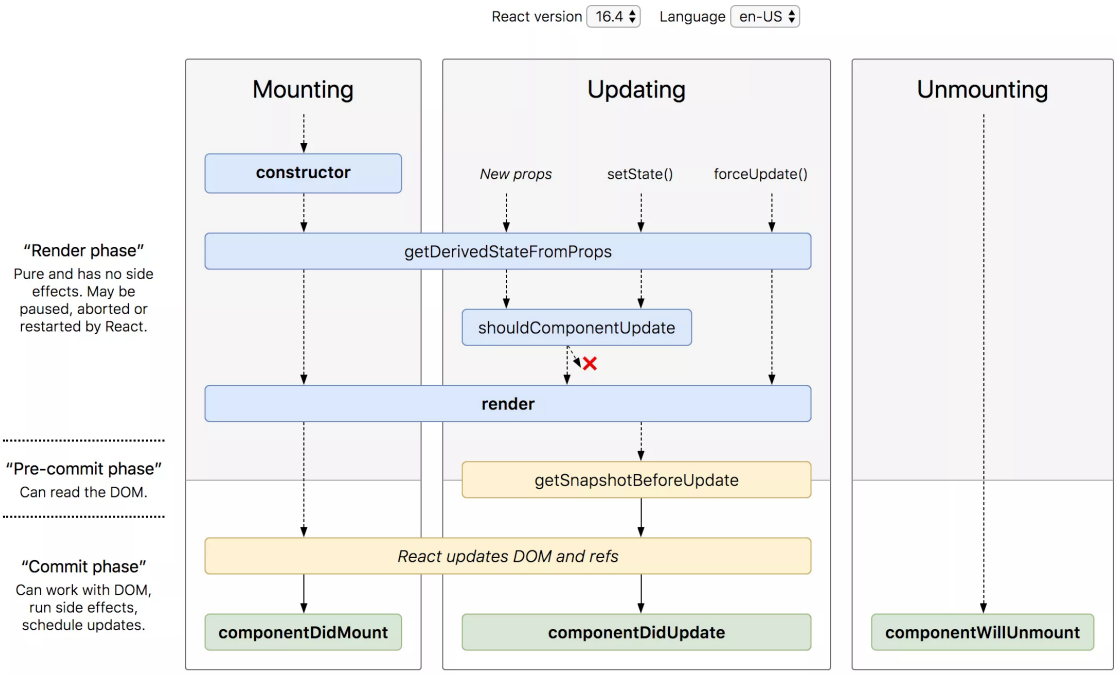
Functions as Props

- ex. accessing state of parent component
- Put a function to set state in the parent component and pass it as a prop to the child component
- `this.props.function()`

```
<button onClick={ () => {function(id)} }>x</button>
```

- Must be an arrow function or it will fire immediately

Lifecycle Methods



Mounting

- Mounting: Putting elements into the DOM
- 4 methods called in this order upon mount
- `render()` is required, others are optional

Constructor

- Called upon component instantiation
- Set up initial values such as `state`

getDerivedStateFromProps

- Called before rendering elements
- Set state based on props

render

- Renders JSX to the DOM

componentDidMount()

- Fires when component first mounts
- Good for grabbing external data (ex. Firebase)

Updating

- Updating data, self-explanatory

shouldComponentUpdate

getSnapshotBeforeUpdate

componentDidUpdate

Unmounting

componentWillUnmount

- Component is about to removed from the DOM

React Router

`npm install react-router-dom`

```
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  withRouter
} from "react-router-dom";
```

```
<Router>
    <Navbar />
    <Route path='/' component={Home} />
</Router>
```

- Add `exact` prop if necessary (anything that starts with `/` will also render in the above example)

```
<!-- Replace <a> tags with <Link> tags -->

<Link to="/">Home</Link>
```

- `NavLink` is `Link` but for styling

Page Redirects

```
// After 2 seconds, redirect to '/path'
setTimeout(() => {
  props.history.push('/path')
}, 2000);
```

- Router automatically adds its information to `props` for any component it loads up

- `<Route component={...} />`

`withRouter()`

```
export default withRouter(component)
```

- Higher order component
- Applies Router props to the component

`Switch`

- Only match one route
- Wraps `<Route/>` components, goes top down and looks to match the first link only and stops

Higher Order Components

- Wraps a component and gives them extra features

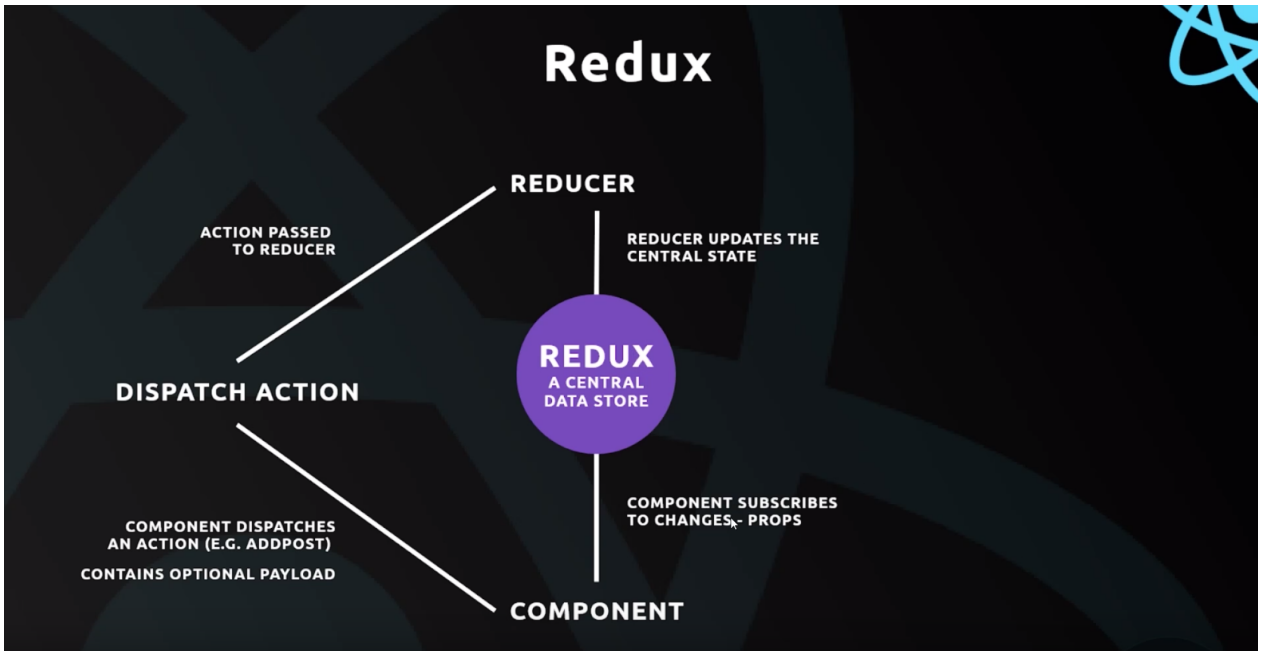
```
const Hoc = (WrappedComponent) => {
  // if the wrapped component received any props, return them
  return (props) => {
    return (
      // return jsx
      <WrappedComponent {...props} />
      // spread props back into component
    )
  }
}
```

Images

- Import image: `import Image from '/path'`

```
<img src={Image} />
```

Redux



- Have central storage for data
- Passing around props is bad

1. Define central store with redux
2. A component subscribes to changes in data; Redux passes data using props

To make a change:

1. Dispatch action
2. Actions describe changes with data payload
3. Action passed to Reducer
4. Reducer updates data store

```
const { createStore } = Redux;

// default state
const initState = {
  todos: [],
```

```
  posts: []
}

function myreducer(state = initState, action){
  if(action.type == 'ADD_TODO'){
    // return a new object representing the new state
    return {

      ...state, // all the other untouched properties
      todos: [...state.todos, action.todo]
    }
  }
}

const store = createStore(myreducer);

// listen to changes to store and react to them
store.subscribe(() => {
  console.log('state updated');
  console.log(store.getState());
})

// type describes action
const todoAction = { type: 'ADD_TODO', todo: 'play piano' }

// send action to Reducer
store.dispatch(todoAction);
```

Redux setup

`npm install redux react-redux`

```
// index.js
...
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import rootReducer from '.rootReducer'
```

```
const store = createStore(rootReducer);

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

- The `rootReducer` is what will interact with the store

```
// rootReducer.js

// need default state
const initState = {
    posts: []
}
const rootReducer = (state = initState, action) => {
    return state;
}
export default rootReducer
```

Mapping State to Props

- Connect component to Redux store
- Pass Redux props to the component

```
// Component
import { connect } from 'react-redux'
...
export default connect()(Component)
```

- `connect` is a function that returns a HOC that wraps the component
- `connect` takes in a parameter which maps the data from the store to the component props

```
// takes in store state as a prop
const mapStateToProps = (state) => {
  return {
    // store properties we want to return
    // ex.
    posts: state.posts
  }
}
```

```
// another example from react-plan
import { connect } from 'react-redux'

// takes state of store and returns object representing props
const mapStateToProps = state => {
    return{
        projects: state.project.projects
    }
}
export default connect(mapStateToProps)(Dashboard)
```

- `connect` listens for changes and calls `mapStateToProps` where we specify which props we provide to the component
- In this case, we provide a new prop called `projects`

Map Dispatch to Props

- To change state dispatch an action

```
// takes dispatch method as param
const mapDispatchToProps = (dispatch) => {
  return {
    // similar to mapStateToProps, maps properties to props of component
    // ex.
    deletePost: (id) => {
      // we dispatch this action whenever deletePost is called
```

```
      // we dispatch this action whenever deletePost is called
      dispatch({type: 'DELETE_POST', id: id})
    }
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(Component)
```

```
// rootReducer

const rootReducer = (state = initState, action) => {
  if(action.type = 'DELETE_POST'){
    ...
  }
  return state;
}
```

Combining Reducers

```
import authReducer from './authReducer'
import projectReducer from './projectReducer'
import { combineReducers } from 'redux'

const rootReducer = combineReducers ({
    // which reducers we want to combine and what we want to call them
    auth: authReducer,
    project: projectReducer,
})

export default rootReducer
```

Redux Thunk

- Perform async tasks inside action creators
- Returns a function:

- Halt dispatch
- Perform async request
- Resume dispatch

```
import { createStore, applyMiddleware } from 'redux'
import thunk from 'react-thunk'

const store = createStore(rootReducer, applyMiddleware(thunk));
```

- check out `CreateProject.js` in react-plan