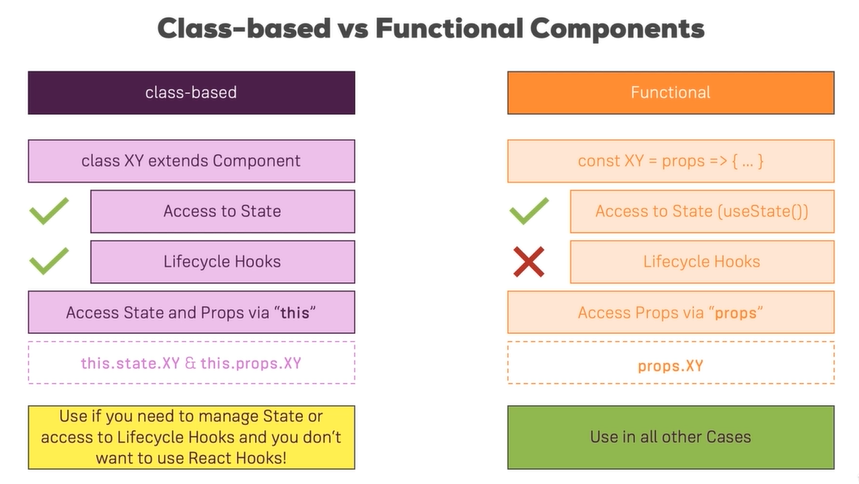
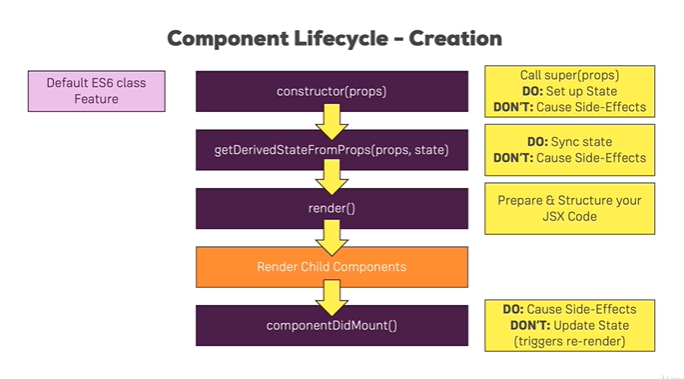
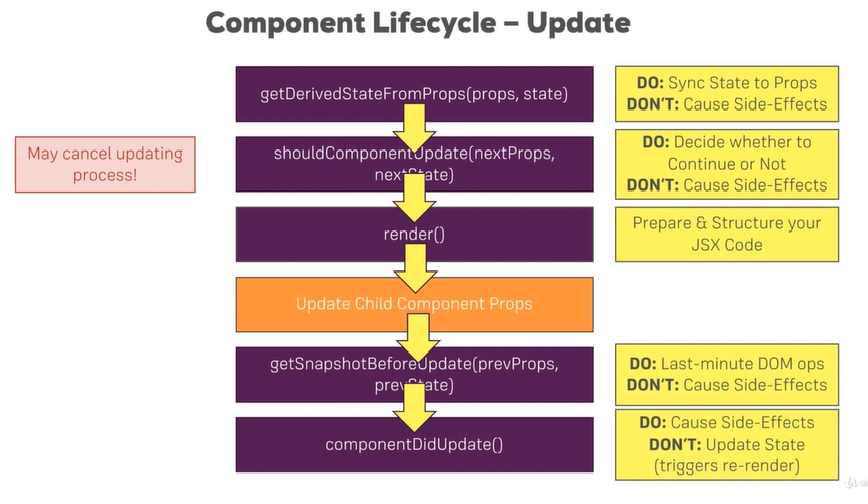
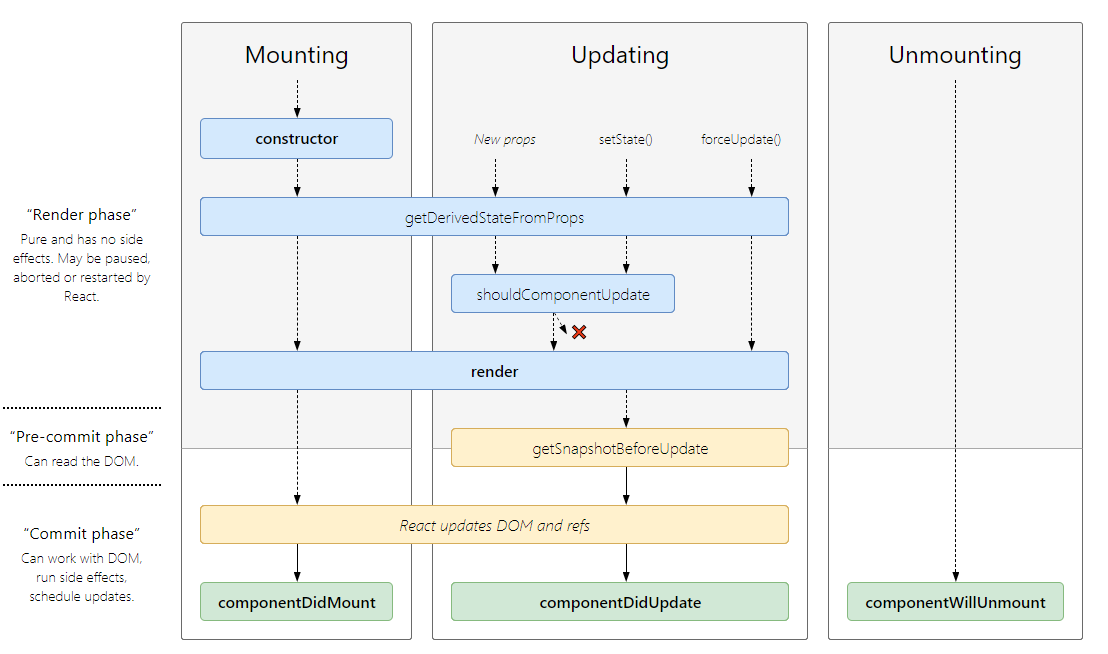
React 16.8

* useState() Hook introduce for state manipulation in functional component.
* Radium: library for adding inline Pseudo/media query in .jsx file.
* ErrorBoundary introduce for error handing.
* Stateless component = Presentational = Dumb: functional component without state management.
* Stateful component = container component.
* Class Based vs Funtional
* Component Lifecycle - Creation



* Component lifecycle- Update

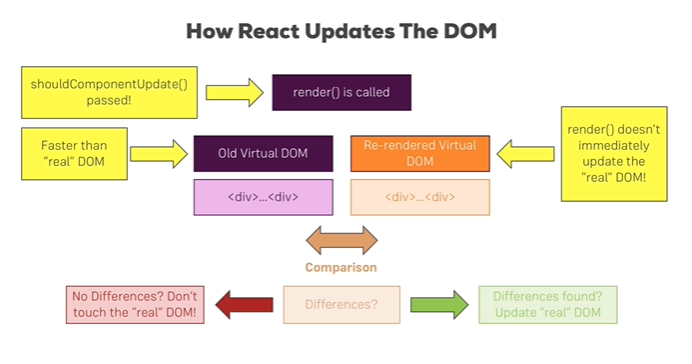




* useEffect() react hook: it just a combination of componentDidMount and componentDidUpdate.

Used in functional component

* React.memo(compoentName): used to improve performance of rendering component when changes in parent component in functional component.
* PureComponent instead of shouldComponentUpdate();
* React Dom Update



* Aux Rendering adjacent Jsx element
* React Fragment: similar as render in child. (React.Fragement)
* Higher order component: that wraps another component without any logics.
* Different types of declaration of HOC.
* setState correctly: this.setState((prevState, props)=>{ return { counter: prevState + 1} })
* Context API: to remove props chain for passing data from child to child,
* ContextType and useContext() hooks, react 16.6

Useful Resources & Links:

* More on useEffect(): <https://reactjs.org/docs/hooks-effect.html>
* State & Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>
* PropTypes: <https://reactjs.org/docs/typechecking-with-proptypes.html>
* Higher Order Components: <https://reactjs.org/docs/higher-order-components.html>
* Refs: <https://reactjs.org/docs/refs-and-the-dom.html>

@Note: With React 16.8, a new feature called "**React Hooks**" was introduced. This course already covers this feature and this module, up to this point, is 100% up-to-date with that feature (e.g. you learned about useEffect()).

**On Windows**, the Aux.js  filename (will be used in next lectures) is not allowed in ZIP archives. Hence when extracting the attached source code, you might get prompted to rename the Aux.js  file. You might also **face difficulties creating an Aux folder** and Aux.js file.

I really apologize for that inconvenience, Windows is really doing an amazing job here ;-).

Follow these fixes:

**1) Problems when unzipping the attached file:**

Simply **skip this step** (e.g. by pressing **"No"**) and **ignore** the upcoming error message.

In the extracted folder, you'll then find **all source files** EXCEPT for the Aux.js  file. In later course modules (where we work on the course project), the Aux.js  file can be found in an Aux/  subfolder inside hoc/ .

Make sure to take the Aux.js  file **attached to this lecture** and place it inside the hoc/  or hoc/Aux/  folder (which ever of the two you got).

**2) Problems with the creation of an Aux folder and/ or file:**

Simply name both differently. For example, you may create an Auxiliary  folder and name the file inside of it Auxiliary.js . Make sure to then adjust your imports (import Aux from './path/to/Auxiliary/Auxiliary' ) and you should be fine.

Available PropTypes

Source: <https://reactjs.org/docs/typechecking-with-proptypes.html>

1. import PropTypes from 'prop-types';
3. MyComponent.propTypes = {
4. // You can declare that a prop is a specific JS primitive. By default, these
5. // are all optional.
6. optionalArray: PropTypes.array,
7. optionalBool: PropTypes.bool,
8. optionalFunc: PropTypes.func,
9. optionalNumber: PropTypes.number,
10. optionalObject: PropTypes.object,
11. optionalString: PropTypes.string,
12. optionalSymbol: PropTypes.symbol,
14. // Anything that can be rendered: numbers, strings, elements or an array
15. // (or fragment) containing these types.
16. optionalNode: PropTypes.node,
18. // A React element.
19. optionalElement: PropTypes.element,
21. // You can also declare that a prop is an instance of a class. This uses
22. // JS's instanceof operator.
23. optionalMessage: PropTypes.instanceOf(Message),
25. // You can ensure that your prop is limited to specific values by treating
26. // it as an enum.
27. optionalEnum: PropTypes.oneOf(['News', 'Photos']),
29. // An object that could be one of many types
30. optionalUnion: PropTypes.oneOfType([
31. PropTypes.string,
32. PropTypes.number,
33. PropTypes.instanceOf(Message)
34. ]),
36. // An array of a certain type
37. optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
39. // An object with property values of a certain type
40. optionalObjectOf: PropTypes.objectOf(PropTypes.number),
42. // An object taking on a particular shape
43. optionalObjectWithShape: PropTypes.shape({
44. color: PropTypes.string,
45. fontSize: PropTypes.number
46. }),
48. // You can chain any of the above with `isRequired` to make sure a warning
49. // is shown if the prop isn't provided.
50. requiredFunc: PropTypes.func.isRequired,
52. // A value of any data type
53. requiredAny: PropTypes.any.isRequired,
55. // You can also specify a custom validator. It should return an Error
56. // object if the validation fails. Don't `console.warn` or throw, as this
57. // won't work inside `oneOfType`.
58. customProp: function(props, propName, componentName) {
59. if (!/matchme/.test(props[propName])) {
60. return new Error(
61. 'Invalid prop `' + propName + '` supplied to' +
62. ' `' + componentName + '`. Validation failed.'
63. );
64. }
65. },
67. // You can also supply a custom validator to `arrayOf` and `objectOf`.
68. // It should return an Error object if the validation fails. The validator
69. // will be called for each key in the array or object. The first two
70. // arguments of the validator are the array or object itself, and the
71. // current item's key.
72. customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location, propFullName) {
73. if (!/matchme/.test(propValue[key])) {
74. return new Error(
75. 'Invalid prop `' + propFullName + '` supplied to' +
76. ' `' + componentName + '`. Validation failed.'
77. );
78. }
79. })
80. };

#### Requiring Single Child

With PropTypes.element you can specify that only a single child can be passed to a component as children.

1. import PropTypes from 'prop-types';
3. class MyComponent extends React.Component {
4. render() {
5. // This must be exactly one element or it will warn.
6. const children = this.props.children;
7. return (
8. <div>
9. {children}
10. </div>
11. );
12. }
13. }
15. MyComponent.propTypes = {
16. children: PropTypes.element.isRequired
17. };

Default Prop Values

You can define default values for your props by assigning to the special defaultProps property:

1. class Greeting extends React.Component {
2. render() {
3. return (
4. <h1>Hello, {this.props.name}</h1>
5. );
6. }
7. }
9. // Specifies the default values for props:
10. Greeting.defaultProps = {
11. name: 'Stranger'
12. };
14. // Renders "Hello, Stranger":
15. ReactDOM.render(
16. <Greeting />,
17. document.getElementById('example')
18. );

The defaultProps will be used to ensure that this.props.name will have a value if it was not specified by the parent component. The propTypes typechecking happens after defaultProps are resolved, so typechecking will also apply to the defaultProps.

Useful Resources & Links

* State & Lifecycle: <https://reactjs.org/docs/state-and-lifecycle.html>
* PropTypes: <https://reactjs.org/docs/typechecking-with-proptypes.html>
* Higher Order Components: <https://reactjs.org/docs/higher-order-components.html>
* Refs: <https://reactjs.org/docs/refs-and-the-dom.html>
* Absolute vs Relative Paths (Article)
* You learned about <Link> , you learned about the to  property it uses.
* The path you can use in to can be either **absolute** or **relative**.

#### ****Absolute Paths****

* By default, if you just enter to="/some-path"  or to="some-path" , that's an **absolute path**.
* **Absolute path** means that it's**always appended right after your domain**. Therefore, both syntaxes (with and without leading slash) lead to example.com/some-path .

#### ****Relative Paths****

* Sometimes, you might want to create a relative path instead. This is especially useful, if your component is already loaded given a specific path (e.g. posts ) and you then want to append something to that existing path (so that you, for example, get /posts/new ).
* If you're on a component loaded via /posts , to="new"  would lead to example.com/new , **NOT** example.com/posts/new .
* To change this behavior, you have to find out which path you're on and add the new fragment to that existing path. You can do that with the url  property of props.match :
* <Link to={props.match.url + '/new'}>  will lead to example.com/posts/new  when placing this link in a component loaded on /posts . If you'd use the same <Link>  in a component loaded via /all-posts , the link would point to /all-posts/new .
* **There's no better or worse way of creating Link paths** - choose the one you need. Sometimes, you want to ensure that you always load the same path, no matter on which path you already are => Use absolute paths in this scenario.
* Use relative paths if you want to navigate relative to your existing path.

Parsing Query Parameters & the Fragment

You learned how to extract route parameters (=> :id  etc).

But how do you extract **search** (also referred to as "**query**") **parameters**(=> ?something=somevalue  at the end of the URL)? How do you extract the **fragment** (=> #something  at the end of the URL)?

**Query Params:**

You can pass them easily like this:

<Link to="/my-path?start=5">Go to Start</Link>

or

1. <Link
2. to={‌{
3. pathname: '/my-path',
4. search: '?start=5'
5. }}
6. >Go to Start</Link>

React router makes it easy to get access to the search string: props.location.search .

But that will only give you something like ?start=5

You probably want to get the key-value pair, without the ?  and the = . Here's a snippet which allows you to easily extract that information:

1. componentDidMount() {
2. const query = new URLSearchParams(this.props.location.search);
3. for (let param of query.entries()) {
4. console.log(param); // yields ['start', '5']
5. }
6. }

URLSearchParams  is a built-in object, shipping with vanilla JavaScript. It returns an object, which exposes the entries()  method. entries()  returns an Iterator - basically a construct which can be used in a for...of...  loop (as shown above).

When looping through query.entries() , you get **arrays** where the first element is the **key name** (e.g. start ) and the second element is the assigned **value** (e.g. 5 ).

**Fragment:**

You can pass it easily like this:

<Link to="/my-path#start-position">Go to Start</Link> or

1. <Link
2. to={‌{
3. pathname: '/my-path',
4. hash: 'start-position'
5. }}
6. >Go to Start</Link>

React router makes it easy to extract the fragment. You can simply access props.location.hash .

Useful Resources & Links

* Validate.js (you may import its functionality into your React projects): <https://validatejs.org/>
* Get more ideas about potential validation approaches: <https://react.rocks/tag/Validation>

Alternatives to the manual approach taken in this course:

* react-validation package: <https://www.npmjs.com/package/react-validation>
* formsy-react package: <https://github.com/christianalfoni/formsy-react>