Diving deeper into components and React Internals

·) A better project structure:

Typically containee components like App.js which are involved with managing state shouldn't be involved with UI rendering much.

↓

render method should be lean.

↓

i) PersonList (persons) component

(pass array of persons, mapping and rendering can then be performed)

Persons

ii) cockpit (the code before persons)

person.js → finctional component → not managing state

↓

can be done with hooks though

*) Class vs functional components

↓

•) access to state
•) lifecycle hooks
•) access state and props via "this"

↓

to be used to manage states or access lifecycle hooks, and React hooks is not wanted

→ •) access to state (using useState)
•) cannot use lifecycle hooks
•) props as an argument and used accordingly

↓

all other cases for older versions (use for presentation components)

*) modifying index.js to pass props to App component which will be employed using this.props, passing title of App, and rendering that by passing to appropriate children components.

*) component lifecycle → generally available in class based components. set of functions that we can add to any class based component, and react will execute them for us.

i) creation : a) constructor (props) → default ES6 class feature

↓

call super (props)

↓

set up state, don't cause side effects

↘ receives the props of this component

b) getDerivedStateFromProps (props, state)

sync state, don't cause side effects

↓

when props change for class based component, state can be sinked to them. (very rare, niche cases)

↓

props changed, and internal state needs to be updated.

c) render() : prepare & structure JSX code,

don't cause side effects

(sending HTTP requests, setting any timeouts, anything that can block the rendering process)

render child components

↓

once they are rendered and their lifecycle hooks finished

↳ our lifecycle hook for creation completed when componentDidMount is called.

↓

typical hook to make HTTP requests

↓

don't update state

(can cause sideeffects)

```
constructor (props) {
    super (props);
    console.log ('[App.js] constructor');
}
```

The state can also be initialized here, rather than the way it is done in our class component (outside using state keyword (relatively modern syntax))

↓

```
static getDerivedStateFromProps(props, state) {
    console.log('[App.js] getDerivedStateFromProps', props);
    return state;
}
```

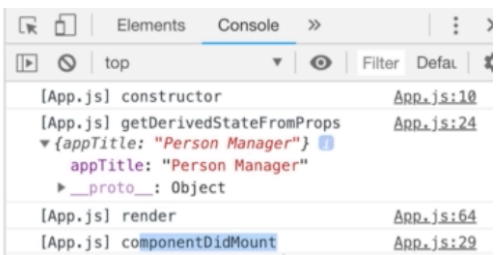
↳ returning the updated state (same in this case)

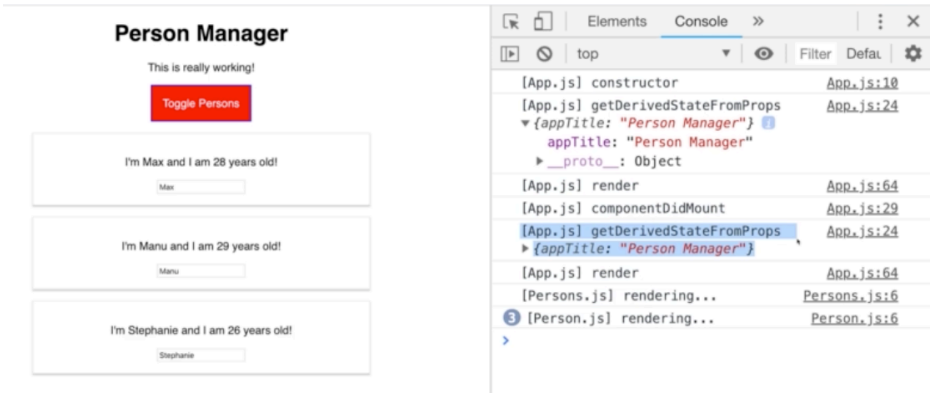
↓

render() → add console.log to understand the progress

↓

```
componentDidMount() {
    console.log();
}
```





ii) updation : a) getDerivedStateFromProps (props, state)

for prop changes — sync state, don't cause sideffects

b) shouldComponentUpdate (nextProps, nextState)

↓

allows us to cancel the updation process

↓

decide whether React should continue evaluating and re-rendering the component.

↓

Allows us to prevent unnecessary updates

c) render() → constructs virtualDOM → preparing and structuring JSX code

d) update child components (recursive process)

e) getSnapshotBeforeUpdate (prevProps, prevState)

f) componentDidUpdate → A lifecycle hook that signals that we're done with the updating (can cause side effects)

↓

may lead to ∞ loop.

*) convert Person, and Persons component to class based components.

in Persons component →

comment ↓ no initial state, hence no sense

```
static getDerivedStateFromProps(props, state){
    console.log('[Persons.js] getDerivedStateFromProps');
    return state;
}
```

upcoming props ↑

```
shouldComponentUpdate(nextProps, nextState){
```

↓ Which will have an effect right after this update which is about to take place.

```
    console.log('[Persons.js] shouldComponentUpdate');
    return true;
}
```

```
getSnapshotBeforeUpdate(prevProps, prevState) {
    console.log('[Persons.js] getSnapshotBeforeUpdate');
    return null;
}
render()
```

→ this gets passed to componentDidUpdate.

```
componentDidUpdate() {
    console.log('[Persons.js] componentDidUpdate');
}
```

comment out componentWillMount in App.js (deprecated)

```
componentWillReceiveProps(props) { }
```

↓ won't work now (legacy)

iii) updation for state changes → App.js

```
componentDidUpdate() {
    console.log();
}
```

→ can add getSnapshot method as well here

```
shouldComponentUpdate(nextProps, nextState) {
    return true;
}
```

→ if false → update won't happen

•) componentDidUpdate, componentDidMount & shouldComponentUpdate are the most important lifecycle hooks.

*) functional component equivalent of lifecycle hooks

↓

useEffect hook (React hook)

↓

function that can be added in our functional components.

```
useEffect ( () => {
    console.log ('[Cockpit.js] useEffect');
});
```

↓

executes for every render cycle of the component ← created

↓

can use this for componentDidUpdate (Sending an HTTP request)

↓

componentDidMount & componentDidUpdate in One effect

*) controlling the useEffect() Behaviour

```
useEffect ( () => {
    console.log ('[Cockpit.js] useEffect');
    setTimeout ( () => {
        alert('Saved data to cloud');
    }, 1000);
}, [props.persons]);
```

an array where we simply point at all the variables or all the data that is actually used in the effect, and their change triggers the effect.

↓

i.e. it should run when one of the dependencies changes

↓

[] empty array means no dependencies, hence gets run only at the time of creation, and change to any component doesn't matter here.

*) Cleaning up with Lifecycle hooks and useEffect():

When the toggle button is clicked to not show persons, then it is not being rendered.

↓

Here some cleanup job can be performed.

componentWillUnmount () { } ; → any code that needs to run before the component is removed.

↓

via useEffect, we can return an anonymous function, that runs after every render cycle (runs before the main useEffect function runs, but after the first render cycle)

↓

```
useEffect ( () => {
    console.log ( '[Cockpit.js] useEffect' );
    setTimeout ( () => {
        alert ( 'Saved data to cloud' );
    }, 1000 );
    return ( ) => {
        console.log ( '[Cockpit.js] cleanup' );
    };
}, [] );
```

↓

i) First run on creation render

ii) Based on the arguments sent in the second array, re-render and execution of the returned function happens on update.

iii) for empty array, the returned fn. is run on unmounting.

↓

Multiple useEffect() can be employed in the function.

↓

timer shows even when toggle button is clicked, we can stop that from happening in the function returned by clearTimeout(timer);

returned function → run for updates only.
(anonymous fn.)

↓

the dependencies can be configured.

↓

can be made for unmount of the component as well.

*) using shouldComponentUpdate for optimization

Persons get re-rendered every time App is re-rendered, even if it affects only the cockpit

↓

tree design execution. Need to control updation at component level

(Rendering options
↓
paint flashing)
to check the components updating in the real DOM.

```
shouldComponentUpdate(nextProps, nextState) {
  console.log('[Persons.js] shouldComponentUpdate');
  if (nextProps.persons !== this.props.persons) {
    return true;
  } else {
    return false;
  }
}
```

→ this works with deep copy, due to the diff. reference space.

*) optimization for functional components.

↓

wrapping the functional component in React.memo()

↓

uses memoization to re-render only if its input changes (stores snapshot, in case no change, share it)

•) not always great to use optimization, especially when parent state leads to children component change every time, since the code execution has its own runtime overhead associated.

*) if we are checking for update to all props members, then we should work with Pure Component rather than a simple component (performs a complete update check)

*) How React updates the real DOM ?

compares virtual DOMS. → old virtual DOM, and future re-rendered virtual DOM.

(faster than comparing with real DOM)

virtual DOM → DOM representation in JS

Differences are changed in the real DOM (only at places the differences were detected)

*) Rendering adjacent JSX elements.

↓

only one root element to be rendered in the component (the div containing the other elements)

↓

to return adjacent JSX elements (>1) we should go for an array. This will be accepted as long as each element has a key associated to it.

(to efficiently update or reorder these elements)

↓

another way would be to use higher order components → to act like wrapping components only.

↙

empty wrapper

create an hoc folder → Aux.js file

↓

aux component, return props.children;

↙ export

special property that simply outputs whatever gets entered between the opening and closing tag.

↓

allows rendering of adjacent elements w/o explicitly rendering an extra DOM element to the real DOM.

*) built in aux component → <React.Fragment>
:
</React.fragment>

↓

or import fragment & use it directly.

*) Higher Order Components → just wraps another component and then maybe adds some extra logic to it.

withclass.js → div with classes to be applied passed down.

```
import React from 'react';

const withClass = props => (
  <div className={props.classes}>{props.children}</div>
);

export default withClass;
```

method 1

through a fnctional component

```
const withClass = (WrappedComponent, className) => {
  return props => (
    <div className={className}>
      <WrappedComponent />
    </div>
  );
};
```

method 2

through a function (this can then be called on the appropriate component with the required classes)

*) passing unknown props → when using high order components, special attention should be paid to passing props. The props should be forwarded to the children by the wrapping component.

↓

can be done by using the spread operator

↓

```
<WrappedContent {...props} />
```

*) Setting state correctly → when the value of the variable of the state being mutated depends on the previous state value. Then we should employ the second method of updating / setting state.

```
this.setState((prevState, props) => {
    return {
        persons: persons,
        changeCounter: prevState.changeCounter + 1
    };
});
};
```

Since using this.state to update is not executed eagerly; and ∴ not guaranteed to be the proper state to work with.

*) using propTypes → npm install --save prop-types

import PropTypes from 'prop-types';

```
Person.propTypes = {
  click : PropTypes.func ,
  name : PropTypes.string ,
  age : Proptypes.number ,
  changed : PropTypes.func
}
```

special property that can be added to any component object, that React will watchout for in development mode, and give us a warning if we then pass in incorrect props

*) using Refs → special property which can be passed to any component, detected and understood by React

↓

Here a reference to the element this property is set on is provided to, and it can accordingly be worked around.

```
key= 15
ref={(inputEl) => {inputEl.focus()}}
type="text"
```

method 1

↓

Another method is to store the reference to a variable in the entire class scope, and that can be referred to or used in the

componentDidMount function.

↓

```
componentDidMount () {
    this.inputElement.focus();
}
```

↓

Another alternate method: would be to create a ReactRef object and then assign the reference to it and use it everywhere. This object is created in the constructor of the component.

↓

```
constructor (props) {
    super(props);
    this.inputElementRef = React.createRef();
}
```

↓

any reference object given by React.

```
ref = {this.inputElementRef}
```

↓

react makes the connection b/w the reference and the variable

↓

```
in componentDidMount () {
    this.inputElementRef.current.focus();
}
```

•) refs in functional components → using React Hooks

↓

useRef hook

instead of the

createRef function

```
const toggleBtnRef = useRef(null);

toggleBtnRef.current.click();
```

↓

this should be executed once

the reference is made,

and for that the component

needs to complete rendering

↓

this can be done using

useEffect()

↙

executed once the entire component

is initially rendered.

*) understanding propchain problems → multilevel

prop passing can get complex and hard to

manage.

↓

logic for state in component A, needs to

be used in D passed through B and C,

even when B and C don't really care about it.

*) ContextAPI → context folder → auth-context.js

```
const authContext = React.createContext({
                        authenticated: false,
                        login: () => {}});
```

i) React context actually allows us to initialize our context with a default value.

ii) context is a globally available JS object (we decide where it is available)

↓

JS object that can be passed between React components w/o using props

*) import AuthContext from '../context/auth-context';

↓

Should wrap all parts of our application that need access to this context.

↓

```
<AuthContext.Provider value={{
```

↙

at the place where context is provided

```
authenticated: this.state.authenticated,
login: loginHandler
}}>
```

<AuthContext.Consumer>

</AuthContext.Consumer>

↓

used where the context is to be consumed.

```
<AuthContext.Provider
  value={{
    authenticated: this.state.authenticated,
    login: this.loginHandler
  }}
>
  {this.state.showCockpit ? (
    <Cockpit
      title={this.props.appTitle}
      showPersons={this.state.showPersons}
      personsLength={this.state.persons.length}
      clicked={this.togglePersonsHandler}
    />
  ) : null}
  {persons}
</AuthContext.Provider>
```

```
<AuthContext.Consumer>
  {context =>
    context.authenticated ? <p>Authenticated!</p> : <p>Please log
  }
</AuthContext.Consumer>
```

◦) contextType & useContext()

The previous method allows access to context only in JSX elements due to the need around components to provide context for.

↓

another method involving a special static property called contextType

```
static contextType = AuthContext;
```

static property → can be accessed from outside w/o the need to instantiate an object of the class

connects component to context, and gives access to a new property called this.context property.

↓

Usable in class based components

```
componentDidMount() {
  // this.inputElement.focus();
  this.inputElementRef.current.focus();
  console.log(this.context.authenticated);
}
```

•) Not available with functional components

↓

useContext hook

↓

allows us to get access to our context anywhere in our functional component function body.

↓

```
const authContext = useContext(AuthContext);
```

(React makes the connection for us behind the scenes)

```
console.log(authContext.authenticated);
```