**ToolKit React**

[Tutorial](https://www.youtube.com/watch?v=9lCmbth63k0&t=13s&ab_channel=JustinKim)

[Documentation](https://redux-toolkit.js.org/introduction/quick-start)

Immer - allow immutable be edited in a mutable way

For example, we can use [lodash](https://lodash.com/) to do the [pull](https://lodash.com/docs/4.17.15#pull), [merge](https://lodash.com/docs/4.17.15#merge) and everything with out spread out the state.

createSlice = createReducer() + createAction()

[createReducer](https://redux-toolkit.js.org/api/createReducer)(parameter1, parameter2)

parameter1 : A initial state

parameter2 : A updated state, after received the action

**Direct State Mutation** [**#**](https://redux-toolkit.js.org/api/createReducer#direct-state-mutation)

[createAction](https://redux-toolkit.js.org/api/createAction)(parameter)

const INCREMENT = 'counter/increment'

Route : Name of the slice for that state / name of the action for that state

1. function increment(amount: number) {
2. return {
3. type: INCREMENT,
4. payload: amount,
5. }
6. }

parameter : amount: number

(Can be defined how many parameters in action creator)

Return a action with an object type and a optional payload

Then these pass to your reducer

[createSlice()](https://redux-toolkit.js.org/api/createSlice)

Automatic create this action creator for the reducer

store.dispatch(counter.actions.increment())

[configureStore()](https://www.youtube.com/watch?v=9lCmbth63k0&t=13s&ab_channel=JustinKim)

Better default for configuration

Reducer : rootReducer

getDefaultMiddleware [#](https://redux-toolkit.js.org/api/getDefaultMiddleware)

**Hook** [**#**](https://reactjs.org/docs/hooks-intro.html)

useState()

Replace the setState()

[useEffect()](https://reactjs.org/docs/hooks-effect.html)

**Effects are consistent by default**

**And declare the state in the scope of above useState function**

const [count, setCount] = useState(0);

useEffect(() => {

// Update the document title using the browser API

document.title = `You clicked ${count} times`;

});

**Replace**

componentDidMount() + componentDidUpdate() + componentWillMount()

The *Effect Hook* lets you perform **side effects** in function components:

The react what to do after it has flushed our components to the Dom

**Data fetching, setting up a subscription, and manually changing the DOM** in React components are all examples of side effects.

Another example

Adding and removing a subscription is so tightly related that useEffect is designed to keep it together. If your effect returns a function, React will run it when it is time to clean up:

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

Another Example:

The subscription logic is also spread between componentDidMount and componentWillUnmount.

[**you can use the *State* Hook more than once**](https://reactjs.org/docs/hooks-state.html#tip-using-multiple-state-variables)**, you can also use several effects. This lets us separate unrelated logic into different effects:**

const [count, setCount] = useState(0);

useEffect(() => {

document.title = `You clicked ${count} times`;

});

const [isOnline, setIsOnline] = useState(null);

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

**Explanation**

[Earlier on this page](https://reactjs.org/docs/hooks-effect.html#example-using-classes-1), we introduced an example FriendStatus component that displays whether a friend is online or not.

Our class reads friend.id from this.props, subscribes to the friend status after the component mounts, and unsubscribes during unmounting:

componentDidMount() {

ChatAPI.subscribeToFriendStatus(

this.props.friend.id,

this.handleStatusChange

);

}

componentWillUnmount() {

ChatAPI.unsubscribeFromFriendStatus(

this.props.friend.id,

this.handleStatusChange

);

}

**If the friend prop changes** while the component is on the screen?

Our component would continue displaying the online status of a different friend.

This is a bug. We would also cause a memory leak or crash when unmounting since the unsubscribe call would use the wrong friend ID.

In a class component, we would need to add componentDidUpdate to handle this case:

componentDidMount() {

ChatAPI.subscribeToFriendStatus(

this.props.friend.id,

this.handleStatusChange

);

}

componentDidUpdate(prevProps) {

// Unsubscribe from the previous friend.id

ChatAPI.unsubscribeFromFriendStatus(

prevProps.friend.id,

this.handleStatusChange

);

// Subscribe to the next friend.id

ChatAPI.subscribeToFriendStatus(

this.props.friend.id,

this.handleStatusChange

);

}

componentWillUnmount() {

ChatAPI.unsubscribeFromFriendStatus(

this.props.friend.id,

this.handleStatusChange

);

}

**UseEffect Solution**

function FriendStatus(props) {

// ...

useEffect(() => {

// ...

ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

};

});

**Extracting a Custom Hook** [**#**](https://reactjs.org/docs/hooks-custom.html)

**A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.**

For example, **useFriendStatus** below is our first custom Hook:

**Using a Custom Hook**

In the beginning, our stated goal was to remove the duplicated logic from the FriendStatus and FriendListItem components. Both of them want to know whether a friend is online.

**Do two components using the same Hook share state?** No.

Now that we’ve extracted this logic to a useFriendStatus hook, we can *just use it:*

function FriendStatus(props) {

const isOnline = useFriendStatus(props.friend.id);

if (isOnline === null) {

return 'Loading...';

}

return isOnline ? 'Online' : 'Offline';

}

function FriendListItem(props) {

const isOnline = useFriendStatus(props.friend.id);

return (

<li style={{ color: isOnline ? 'green' : 'black' }}>

{props.friend.name}

</li>

);

}

**Pass Information Between Hooks**

const [recipientID, setRecipientID] = useState(1);

const isRecipientOnline = useFriendStatus(recipientID);

Because the useState Hook call gives us the latest value of the recipientID state variable, we can pass it to our custom useFriendStatus Hook as an argument:

[useCallback()](https://react-redux.js.org/api/hooks#usedispatch)

when passing a callback using dispatch to a child component, you should memoize it with [useCallback](https://reactjs.org/docs/hooks-reference.html" \l "usecallback), just like you should memoize any passed callback.

This avoids unnecessary rendering of child components due to the changed callback reference.

You can safely pass [dispatch] in the dependency array for the useCallback call - since dispatch won't change, the callback will be reused properly (as it should). For example:

export const CounterComponent = ({ value }) => {

const dispatch = useDispatch()

const incrementCounter = useCallback(

() => dispatch({ type: 'increment-counter' }),

[dispatch]

)

[Context](https://reactjs.org/docs/context.html)

In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application.

**Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.**

**Only want to avoid passing some props through many levels,**[**component composition**](https://reactjs.org/docs/composition-vs-inheritance.html)**is often a simpler solution than context.**

**Dialog composition for the props.children pass in** [**#**](https://codepen.io/gaearon/pen/gwZbYa?editors=0010)

**Pass down** the Avatar component itself (**the user props and link)** [**#**](https://reactjs.org/docs/context.html#before-you-use-context)

With this change, only the top-most Page component needs to know about the Link and Avatar components’ use of user and avatarSize.

function Page(props) {

const user = props.user;

const userLink = (

<Link href={user.permalink}>

<Avatar user={user} size={props.avatarSize} />

</Link>

);

return <PageLayout userLink={userLink} />;

}

// Now, we have:

<Page user={user} avatarSize={avatarSize} />

// ... which renders ...

<PageLayout userLink={...} />

// ... which renders ...

<NavigationBar userLink={...} />

// ... which renders ...

{props.userLink}

You’re not limited to a single child for a component. You may pass multiple children, or even have multiple separate “slots” for children, [as documented here](https://reactjs.org/docs/composition-vs-inheritance.html#containment):

[useSelector()](https://react-redux.js.org/api/hooks)

useSelector with an inline a new instance of the selector is created whenever the component is rendered

When the selector depend on the state, declared outside the component

[useDispatch()](https://react-redux.js.org/api/hooks#usedispatch)

This hook returns a reference to the dispatch function from the Redux store. You may use it to dispatch actions as needed.

import React from 'react'

import { useDispatch } from 'react-redux'

export const CounterComponent = ({ value }) => {

const dispatch = useDispatch()

return (

<div>

<span>{value}</span>

<button onClick={() => dispatch({ type: 'increment-counter' })}>

Increment counter

</button>

</div>

)

}

**Context**

[SettingsContext.Provider](https://reactjs.org/docs/context.html#contextprovider) - allows consuming components to subscribe to context changes.