React – Applying Redux

• What is Redux?
ans.
- Redux is a predictable state container for JavaScript apps.

It helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test. On top of that, it provides a great developer experience, such as live code editing combined with a time traveling debugger.

You can use Redux together with React, or with any other view library. It is tiny (2kB, including dependencies), but has a large ecosystem of addons available.

Installation

Redux Toolkit

Redux Toolkit is our official standard approach for writing Redux logic. It wraps around the Redux core, and contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

RTK includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application, Redux Toolkit can help you make your Redux code better.

Redux Toolkit is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
npm install @reduxjs/toolkit

# Yarn
yarn add @reduxjs/toolkit
```

Create a React Redux App

The recommended way to start new apps with React and Redux is by using our official Redux+TS template for Vite, or by creating a new Next.js project using Next's with-redux template.

Both of these already have Redux Toolkit and React-Redux configured appropriately for that build tool, and come with a small example app that demonstrates how to use several of Redux Toolkit's features.

```
# Vite with our Redux+TS template
# (using the `degit` tool to clone and extract the template)
npx degit reduxjs/redux-templates/packages/vite-template-redux my-app

# Next.js using the `with-redux` template
npx create-next-app --example with-redux my-app
```

Redux Core

The Redux core library is available as a package on NPM for use with a module bundler or in a Node application:

```
# NPM
```

```
npm install redux

# Yarn
yarn add redux
```

It is also available as a precompiled UMD package that defines a window.Redux global variable. The UMD package can be used as a <script> tag directly.

• What is Redux Thunk used for?

The word "thunk" is a programming term that means "a piece of code that does some delayed work". Rather than execute some logic now, we can write a function body or code that can be used to perform the work later.

• What is Pure Component? When to use Pure Component over Component?

Based on the concept of purity in functional programming paradigms, a function is said to be pure if it meets the following two conditions:

Its return value is only determined by its input values
Its return value is always the same for the same input values
A React component is considered pure if it renders the same output for the same state and props. For this type of class component, React provides the PureComponent base class. Class components that extend the React.PureComponent class are treated as pure components.

Pure components have some performance improvements and render optimizations since React implements the shouldComponentUpdate() method for them with a shallow comparison for props and state.

How does a pure component work in React?
In practice, a React pure component looks like the following code:

```
import React from 'react';

class PercentageStat extends React.PureComponent {

    render() {
        const { label, score = 0, total = Math.max(1, score) }
= this.props;
```

```
        return (
            <div>
                <h6>{ label }</h6>
                <span>{ Math.round(score / total *
100) }%</span>
            </div>
        )
    }

}

export default PercentageStat;
```

Are React functional components pure?

Functional components are very useful in React, especially when you want to isolate state management from the component. That’s why they are often called stateless components.

However, functional components cannot leverage the performance improvements and render optimizations that come with React.PureComponent since by definition, they are not classes.

If you want React to treat a functional component as a pure component, you’ll have to convert the functional component to a class component that extends React.PureComponent.

Check out the simple example below:

```
// FUNCTIONAL COMPONENT
function PercentageStat({ label, score = 0, total =
Math.max(1, score) }) {
    return (
        <div>
            <h6>{ label }</h6>
            <span>{ Math.round(score / total *
100) }%</span>
        </div>
    )
}
```

What is the second argument that can optionally be passed tosetState and what is its purpose?

Initialisation
Before we can use state, we need to declare a default set of values for the initial state. This can be done by either creating a state object in the constructor or directly within the class.

```
class Counter extends React.Component {
    constructor(props, context) {
        super(props, context)
        this.state = {
            quantity: 1,
            counter: 0
        }
    }
}
class Counter extends React.Component {
    state = {
        quantity: 1,
        counter: 0
    }
}
```

Update

State can be updated in response to event handlers, server responses or prop changes. React provides a method called setState for this purpose.

setState() enqueues changes to the component state and tells React that this component and its children need to be re-rendered with the updated state.

```
this.setState({quantity: 2})
```

Here, we passed setState() an object containing part(s) of the state we wanted to update. The object passed would have keys corresponding to the keys in the component state, then setState() updates or sets the state by merging the object to the state.

setState and re-rendering
setState() will always lead to a re-render unless shouldComponentUpdate() returns false. To avoid unnecessary renders, calling setState() only when the new state differs from the previous state makes sense and can avoid calling setState() in an infinite loop within certain lifecycle methods like componentDidUpdate.

React 16 onwards, calling setState with null no longer triggers an update. This means we can decide if the state gets updated within our setState method itself!

Signature
setState(updater[, callback])
The first argument is an updater function with the signature:

(prevState, props) => stateChange
prevState is a reference to the previous state. It should not be directly mutated. Instead, changes should be

represented by building a new object based on the input from prevState and props. For example, to increment a value in state by props.step:

```
this.setState((prevState, props) => {
    return {counter: prevState.counter + props.step};
})
```

Due to the async nature of setState, it is not advisable to use this.state to get the previous state within setState. Instead, always rely on the above way. Both prevState and props received by the updater function are guaranteed to be up-to-date. The output of the updater is shallowly merged with prevState.

The second parameter to setState() is an optional callback function that will be executed once setState is completed and the component is re-rendered. componentDidUpdate should be used instead to apply such logic in most cases.

You may directly pass an object as the first argument to setState instead of a function. This performs a shallow merge of the state change into the new state.

```
this.setState({quantity: 2})
```

Batching state updates

In case multiple setState() calls are made, React may batch the state updates while respecting the order of updates. Currently (React 16 and earlier), only updates inside React event handlers are batched by default. Changes are always flushed together at the end of the event and you don’t see the intermediate state.

It doesn’t matter how many setState() calls in how many components you do inside a React event handler, they will produce only a single re-render at the end of the event. For example, if child and parent each call setState() when handling a click event, the child would only re-render once.

Till React 16, there is no batching by default outside of React event handlers. So, each setState() would be processed immediately as it happens if they lie outside any event handler. For example:

```
promise.then(() => {
    // We're not in an event handler, so these are flushed
separately.
    this.setState({a: true}); // Re-renders with {a: true, b:
false }
    this.setState({b: true}); // Re-renders with {a: true, b:
true }
})
```

However, ReactDOM provides an api which could be used to force batch updates.

```
promise.then(() => {
    // Forces batching
    ReactDOM.unstable_batchedUpdates(() => {
        this.setState({a: true}); // Doesn't re-render yet
        this.setState({b: true}); // Doesn't re-render yet
    });
    // When we exit unstable_batchedUpdates, re-renders
once
})
```

Internally React event handlers are all wrapped in unstable_batchedUpdates due to which they're batched by default. Wrapping an update in unstable_batchedUpdates twice has no effect. The updates are flushed when the outermost unstable_batchedUpdates call exits.

The API is “unstable” in the sense that it will be removed when batching is already enabled by default in the React core.