new Date(2021, 2, 28);

React.createElement()

style property in react requires Javascript object.

1. {ReactDom.createPortal(
2. <Backdrop onConfirm={props.onConfirm} />,
3. document.getElementById("backdrop-root")
4. )}
5. localStorage.setItem("isLoggedIn", "1");
6. localStorage.getItem("isLoggedIn");

function after 'return' is called cleanup function

-cleanup function will run before useEffect function or whenever there is change in DOM, before useEffect function.(except for the first time) or when component unmounts



useContext:

Step-1 : Create context

*const* CreatedContext = React.createContext({

  var1: "",

});

Step 2:

Make Provider of Context

*const* ProviderContext = (*props*) *=>* (

  <*CreatedContext.Provider* value={{ var1: "hello", var2: "hi" }}>

    {*props*.children}

  </*CreatedContext.Provider*>

);

Step:3

Wrap App component

ReactDOM.render(

  <*ProviderContext*>

    <*React.StrictMode*>

      <*App* />

    </*React.StrictMode*>

  </*ProviderContext*>,

  document.getElementById("root")

);

Step:4

Consume Context

*const* ConsumeContext = () *=>* {

*const* ctx = useContext(CreatedContext);

  console.log(ctx);

  return <div>HI</div>;

};

Notes:

- Whichever component consumes context (useContext) will re-render if context re-render due to state management in context.

- if context is holding data only & no state management then it will not re-render even if consuming components re-renders.

**Redux basics:**

*const* redux = require("redux");

//reducer function. it receives first argument as prev state

//initialize state

*const* reducerFn = (*state* = { counter: 0 }, *action*) *=>* {

  if (*action*.type === "increament") {

    console.log("entered");

    return { counter: *state*.counter + 1 };

  }

  if (*action*.type === "decrement") {

    return { counter: *state*.counter - 1 };

  }

  return *state*;

};

//creating store & pointing to reducer function

*const* store = redux.createStore(reducerFn);

console.log(store.getState());

//subscription Fn

*const* subscriptionFn = () *=>* {

*const* latestStore = store.getState();

  console.log(latestStore.counter);

};

//let redux aware about subscription

store.subscribe(subscriptionFn);

//dispatch action

store.dispatch({ type: "decrement" });

store.dispatch({ type: "increament" });

**Redux using with React:**

npm install redux react-redux

-create Strore, reducerFn, and point to reducerFn

//action.amount is called payload //always return new state object & never mutate the existing state

import { createStore } from "redux";

*const* reducerFn = (*state* = { counter: 0, show: true }, *action*) *=>* {

  if (*action*.type === "increament")

    return { counter: *state*.counter + *action*.amount, show: *state*.show };

  if (*action*.type === "decrement")

    return { ...*state*, counter: *state*.counter - 1 };

  if (*action*.type === "toggle") return { ...*state*, show: !*state*.show };

  return *state*;

};

*const* store = createStore(reducerFn);

export default store;

Wrap App component with Provider:

import { Provider } from "react-redux";

import store from "./components/store/store";

*const* root = ReactDOM.createRoot(document.getElementById("root"));

root.render(

  <*Provider* store={store}>

    <*App* />

  </*Provider*>

);

Subscribe using useSelector. Using this only part of state can be selected instead of whole state. And it also provides subscription. If component is unmounted, then subscription is removed. Multiple useSelector can be used.

Create dispatch function using useDispatch()// no argument

import { useSelector, useDispatch } from "react-redux";

*const* Counter = () *=>* {

*const* counter = useSelector((*state*) *=>* *state*.counter);

*const* show = useSelector((*state*) *=>* *state*.show);

*const* dispatch = useDispatch();

*const* increaseHandler = () *=>* {

    dispatch({ type: "increament", amount: 2 });

  };

*const* decreaseHandler = () *=>* {

    dispatch({ type: "decrement" });

  };

*const* toggleCounterHandler = () *=>* {

    dispatch({ type: "toggle" });

  };

  return (

    <main className={classes.counter}>

      <h1>Redux Counter</h1>

      {show && <div className={classes.value}>{counter}</div>}

      <div>

        <button onClick={increaseHandler}>Increase</button>

        <button onClick={decreaseHandler}>Decrease</button>

      </div>

      <button onClick={toggleCounterHandler}>Toggle Counter</button>

    </main>

  );

};

export default Counter;

**use redux- toolkit:**

npm install @reduxjs/toolkit;

createSlice in Store:

any parameter sent in dispatch can be accessed by unique payload key

import { createSlice, configureStore } from "@reduxjs/toolkit";

*const* initialCountState = { counter: 0, show: true };

*const* counterSlice = createSlice({

  name: "counter",

  initialState: initialCountState,

  reducers: {

    increament(*state*, *action*) {

*state*.counter = *state*.counter + *action*.payload.amount;

    },

    decrement(*state*) {

*state*.counter++;

    },

    toggle(*state*) {

*state*.show = !*state*.show;

    },

  },

});

Create another Slice:

*const* initialAuthState = { authStatus: false };

*const* authSlice = createSlice({

  name: "Auth",

  initialState: initialAuthState,

  reducers: {

    login(*state*) {

*state*.authStatus = true;

    },

    logout(*state*) {

*state*.authStatus = false;

    },

  },

});

Create global store which is imported in Provider and passed on as store:

Provide name for each slice to identify

*const* store = configureStore({

  reducer: { counter: counterSlice.reducer, auth: authSlice.reducer },

});

Access and output reducer functions:

export *const* counterActions = counterSlice.actions;

export *const* authActions = authSlice.actions;

Subscribe like below:

*const* authStatus = useSelector((*state*) *=>* *state*.auth.authStatus);

*const* counter = useSelector((*state*) *=>* *state*.counter.counter);

Imort Dispatch actions & dispatch actions:

import { counterActions } from "../components/store/store";

    dispatch(counterActions.increament({ amount: 3 }));

    dispatch(authActions.login());

Asynchronous code with redux. Never add async code in reducer function. So 2 approaches:

1. add async code in component. (not a preferred approach)
2. Use middleware (like thunk)

Redux Thunk is a middleware that lets you call action creators that return a function instead of an action object.

In slice file:

*const* addToCart = (*cart*) *=>* {

  return async (*dispatch*) *=>* {

    dispatch(cartActions.additioninCart(*cart*));

*const* addCart = async () *=>* {

*const* response = await fetch("API");

      if (!response.ok) throw new *Error*("error");

    };

    try {

      await addCart();

    } catch (err) {

      console.log(err);

    }

  };

};

dispatch(addToCart(cart));

**Testing react App:**

Manual testing & automated testing

Types of automated testing:

1. Unit tests: test each function, components
2. Integrated tests: test with integration of some components
3. End to end testing: Overall app what users going to experience (Can be done manually also – partially)

Basic default test.js:

import { render, screen } from '@testing-library/react';

import App from './App';

test('renders learn react link', () *=>* {

  render(<*App* />);

*const* linkElement = screen.getByText(/learn react/i);

  expect(linkElement).toBeInTheDocument();

});

3As in writings tests: 1. Arrange (arrange test env) 2. Act 3. Assert

screen.get (functions starting with get) ==> throw an error if element is not found

screen.find(functions starting with find) ==> will return as promise

screen.query(function starting with query) ==> wont return error

test suites are tests grouped together using describe function:

1. describe("Greeting Componenet", () => {
2. test1;
3. test2;
4. });

test, expect, describe are global functions. Render & screen are imported from @testing-library/react

For click event:

1. import userEvent from "@testing-library/user-event";
2. import { act } from "react-dom/test-utils";
3. test("renders 'its good to see you' if button is clicked", () => {
4. render(<Greeting />);
6. //action
7. act(()=>{ //act to be used if action is changing the state.
8. const buttonElement = screen.getByRole("button");
9. userEvent.click(buttonElement);
10. }
12. const checkRendering = screen.queryByText("Its good to see you");
13. expect(checkRendering).toBeNull();
14. });

Or instead of using act, we can use fireEvent.click()

Greetings component is rendered with children.

Mock function to replace fetch function as it will hammer the server. Or if there is post request then it will create unnecessary data on server.

import { render, screen } from "@testing-library/react";

import { Async } from "./Async";

describe("Async code check", () *=>* {

  test("check if list is rendered", async () *=>* {

    window.fetch = jest.fn();

    window.fetch.mockResolvedValueOnce({

      json: async () *=>* [{ id: 0, title: "listName" }],

    });

    render(<*Async* />);

*const* listItems = await screen.findAllByRole("listitem"); //will return array

    expect(listItems).not.toHaveLength(0);

  });

});

beforeEach, afterEach, beforeAll, afterAll, these are hooks of jest which helps in tests configuration.

Top level code runs first even in inside Describe. Then tests are executed line by line.

Above hooks are describe scoped if defined inside describe scope. Else it is global scope.

test.afterEach(cleanup) cleanup function: **cleanup** clears the DOM ([use with afterEach](https://testing-library.com/docs/react-testing-library/api#cleanup) to reset DOM between tests)

Methods of testing



Async Methods: 1. findQueries 2. WaitFor methods

userEvent: fireEvent.click

**TypeScript:**

Javascript is dynamically typed but typescript is static typed. It gives error in IDE before running it. During writing phase it detects the issue.

*interface* store {

  removeFunction: (*id*: *string*) *=>* *void*; // function is not returning anything

  listData: Todo[]; // array of Todo interface

  submitHandler: (*data*: *string*) *=>* *number*;  // function returning number

}

Class can act as interface

export *class* Todo {

  item: *string*;

  id: *string*;

*constructor*(*itemText*: *string*) {

    this.item = *itemText*;

    this.id = new *Date*().toISOString();

  }

}

Submit Handler of <form>

*const* submitHandler = (*event*: React.FormEvent) *=>* {

*event*.preventDefault();

*const* data = todoTextInputRef.current!.value;

Function component type & useref to be attached to HTML Input element

*const* Form: React.FC = (*props*) *=>* {

*const* todoTextInputRef = useRef<*HTMLInputElement*>(null);

interface ProvideContextProps {

children: React.ReactNode;

}

Add props structure

*const* List: React.FC<{

  item: *string*;

  id: *string*;

}> = (*props*) *=>* {

useState: (This is also example of generic type)

*const* [listData, setListData] = useState<Todo[]>([]);

type & interface are similar

1. Declaration Merging: **interface** supports declaration merging, which allows you to split an interface definition into multiple parts that automatically merge together. **type** does not support declaration merging.
2. Extending and Implementing: **interface** can extend other interfaces using the **extends** keyword and can be implemented by classes using the **implements** keyword. **type** can use intersections to combine types but cannot be implemented by classes.
3. Type Alias vs Interface: **type** can create aliases for primitive types, union types, intersection types, mapped types, and more. **interface** is focused on defining object shapes and is not as versatile as **type**

type is versatile like below, but interface are used for object types:

Ex: type stringornumber = string | number

Ex:

Interface{

appCode: string

}

export type AppAction = {type: “SET\_APP\_DETAILS”, appPayload: AppPayload}

Here type in AppAction object is set as string literal “SET\_APP\_DETAILS”, it means it cannot be anything else.

**Redux-saga:**

Generator functions:

function\* idGenerator(val=0) {

console.log('fun') //only once during first next()

let id = 1;

while (true) {

console.log('before yield') // will be called first time next()

yield val+id++;

console.log('yiels') // will be called in 2nd next()

}

}

const gen = idGenerator();

console.log(idGenerator().next()) //{ value: 1, done: false }

console.log(idGenerator().next().value); // 2

create saga:

function\* fetchUser(){

const jsondata = yield call(fetch, url); //calls the async code

const data = yield jsondata.json();

yield put('ADD\_USER', data); //send the action to reducer.

}

function\* watcherMySaga (){

yield takeEvery('FETCH\_USER', fetchUser) // for each request. takeLatest for latest request.

}

In Store:

Const sagaMiddleware = CreateSagaMiddleware()

Const store = createStore(rootreducer, applyMiddleware(sagaMiddleware))

sagaMiddleware.run(watcherMySaga)

yield all() //to combine

**WebPack:**

CSS loader converts css to javascript code and style loader converts add that converted css to js file as style tag

Contenthash is provided so that everytime code is changed, new hash Is generated means new file name is generate. Otherwise browser will cache the file and changes will not get reflected.

But due to above, we canot keep manually changing the script name in the index.html file. SO this html file also should be created dynamically by webpack. That’s where plugins come into the picture.

[HtmlWebpackPlugin](https://webpack.js.org/plugins/html-webpack-plugin) solves above problem but it creates its own html. Doesn’t consider our html file. So to resolve that, we can provide template to its argument, so it will keep our html code also intact.

For third party library like bootstrap we can create new entry point and its separate file will be outputted. Basically this will change only if version of vendor lib is changed. So its easy to cache.

But even after this, scripts are loaded after html parsing and these are having css. So when page is loaded, html content is flashed then css styling is applied which is not a good user experience.

So, to avoid this, seaparte css file to be spitted out which can be loaded during html parsing start. But this is a time consuming process so it is to be used in production only, not in dev server. (MiniCssExtractPluging)

Webpack.common.js:

const path = require("path");

var HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

entry: {

main: "./src/index.js",

vendor: "./src/vendor.js"

},

module: {

rules: [

{

test: /\.html$/,

use: ["html-loader"]

},

{

test: /\.(svg|png|jpg|gif)$/,

use: {

loader: "file-loader",

options: {

name: "[name].[hash].[ext]",

outputPath: "imgs"

}

}

}

]

}

};

Webpack.dev.js:

const path = require("path");

const common = require("./webpack.common");

const merge = require("webpack-merge");

var HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = merge(common, {

mode: "development",

output: {

filename: "[name].bundle.js",

path: path.resolve(\_\_dirname, "dist")

},

plugins: [

new HtmlWebpackPlugin({

template: "./src/template.html"

})

],

module: {

rules: [

{

test: /\.scss$/,

use: [

"style-loader", //3. Inject styles into DOM

"css-loader", //2. Turns css into commonjs

"sass-loader" //1. Turns sass into css

]

}

]

}

});

webpack.dev.js:

const path = require("path");

const common = require("./webpack.common");

const merge = require("webpack-merge");

const CleanWebpackPlugin = require("clean-webpack-plugin");

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

const OptimizeCssAssetsPlugin = require("optimize-css-assets-webpack-plugin");

const TerserPlugin = require("terser-webpack-plugin");

var HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = merge(common, {

mode: "production",

output: {

filename: "[name].[contentHash].bundle.js",

path: path.resolve(\_\_dirname, "dist")

},

optimization: {

minimizer: [

new OptimizeCssAssetsPlugin(),

new TerserPlugin(),

new HtmlWebpackPlugin({

template: "./src/template.html",

minify: {

removeAttributeQuotes: true,

collapseWhitespace: true,

removeComments: true

}

})

]

},

plugins: [

new MiniCssExtractPlugin({ filename: "[name].[contentHash].css" }),

new CleanWebpackPlugin()

],

module: {

rules: [

{

test: /\.scss$/,

use: [

MiniCssExtractPlugin.loader, //3. Extract css into files

"css-loader", //2. Turns css into commonjs

"sass-loader" //1. Turns sass into css

]

}

]

}

});

Package.json:

"name": "code",

"version": "1.0.0",

"private": true,

"description": "",

"main": "index.js",

"scripts": {

"start": "webpack-dev-server --config webpack.dev.js --open",

"build": "webpack --config webpack.prod.js"

},

"keywords": [],

"author": "",

"license": "ISC",

"devDependencies": {

"bootstrap": "^4.3.1",

"css-loader": "^2.1.0",

"file-loader": "^3.0.1",

"html-loader": "^0.5.5",

"html-webpack-plugin": "^3.2.0",

"jquery": "^3.3.1",

"mini-css-extract-plugin": "^0.5.0",

"node-sass": "^4.11.0",

"optimize-css-assets-webpack-plugin": "^5.0.1",

"popper.js": "^1.14.7",

"sass-loader": "^7.1.0",

"style-loader": "^0.23.1",

"webpack": "^4.29.6",

"webpack-cli": "^3.2.3",

"webpack-dev-server": "^3.2.1",

"webpack-merge": "^4.2.1"

},

"dependencies": {

"clean-webpack-plugin": "^2.0.0"

}

}

**D3.js**

D3: Data-Driven Documents

Scalable Vector Graphics (SVG): SVG is a way to render images on the webpage. SVG is not a direct image, but is just a way to create images using text. As its name suggests, it is a Scalable Vector. It scales itself according to the size of the browser, so resizing your browser will not distort the image

<svg width = "500" height = "500">

<rect x = "0" y = "0" width = "300" height = "200" fill = "yellow"></rect>

</svg>

select() − Selects only one DOM element (first)

selectAll() − Selects all DOM elements by matching the given CSS selector.

The append() Method: The append() method appends a new element as the last child of the element in the current selection.

The text() Method: The text() method is used to set the content of the selected / appended elements

The html() Method

The html() method is used to set the html content of the selected / appended elements.

Attr() method

<div class = "myclass">

Hello World!

</div>

<script>

d3.select(".myclass").attr("style", "color: red");

</script>

Style() method:

<script>

d3.select(".myclass").style("color", "red");

</script>

Classed method

d3.select(".myclass").classed("myanotherclass", true);

d3.select(".myclass").classed("myanotherclass", false);

tansform

<g transform = "translate(60,60) rotate(30)">

<rect x = "20"

y = "20"

width = "60"

height = "60"

fill = "green">

</rect>

<circle cx = "0"

cy = "0"

r = "30"

fill = "red"/>

</g>

transition:

d3.selectAll("h3").transition()

.style("font-size","28px").delay(2000).duration(2000);

Barchart:

<html>

<head>

<script type = "text/javascript" src = "https://d3js.org/d3.v4.min.js"></script>

<style>

svg rect {

fill: gray;

}

svg text {

fill: yellow;

font: 12px sans-serif;

text-anchor: end;

}

</style>

</head>

<body>

<script>

var data = [10, 5, 12, 15];

var width = 300

scaleFactor = 20,

barHeight = 30;

var graph = d3.select("body")

.append("svg")

.attr("width", width)

.attr("height", barHeight \* data.length);

var bar = graph.selectAll("g")

.data(data)

.enter()

.append("g")

.attr("transform", function(d, i) {

return "translate(0," + i \* barHeight + ")";

});

bar.append("rect").attr("width", function(d) {

return d \* scaleFactor;

})

.attr("height", barHeight - 1);

bar.append("text")

.attr("x", function(d) { return (d\*scaleFactor); })

.attr("y", barHeight / 2)

.attr("dy", ".35em")

.text(function(d) { return d; });

</script>

</body>

</html>

Line Chart:

<!DOCTYPE html>

<html>

<head>

<script type = "text/javascript" src = "https://d3js.org/d3.v4.min.js"></script>

<style>

.line {

fill: none;

stroke: green;

stroke-width: 5px;

}

</style>

</head>

<body>

<script>

// set the dimensions and margins of the graph

var margin = {top: 20, right: 20, bottom: 30, left: 50},

width = 960 - margin.left - margin.right,

height = 500 - margin.top - margin.bottom;

// set the ranges

var x = d3.scaleTime().range([0, width]);

var y = d3.scaleLinear().range([height, 0]);

// define the line

var valueline = d3.line()

.x(function(d) { return x(d.year); })

.y(function(d) { return y(d.population); });

// append the svg obgect to the body of the page

// appends a 'group' element to 'svg'

// moves the 'group' element to the top left margin

var svg = d3.select("body").append("svg")

.attr("width", width + margin.left + margin.right)

.attr("height", height + margin.top + margin.bottom)

.append("g").attr("transform",

"translate(" + margin.left + "," + margin.top + ")");

// Get the data

d3.csv("data.csv", function(error, data) {

if (error) throw error;

// format the data

data.forEach(function(d) {

d.year = d.year;

d.population = +d.population;

});

// Scale the range of the data

x.domain(d3.extent(data, function(d) { return d.year; }));

y.domain([0, d3.max(data, function(d) { return d.population; })]);

// Add the valueline path.

svg.append("path")

.data([data])

.attr("class", "line")

.attr("d", valueline);

// Add the X Axis

svg.append("g")

.attr("transform", "translate(0," + height + ")")

.call(d3.axisBottom(x));

// Add the Y Axis

svg.append("g")

.call(d3.axisLeft(y));

});

</script>

</body>

</html>

Lighthouse:

LCP optimization: <https://www.youtube.com/watch?v=fWoI9DXmpdk>

-Ensure that LCP resource starts loading as early as possible (You can use pre-load attribute on html directly)

-Ensure the LCP resource can render as soon as its resource finishes loading (can be server side rendered on html directly)

-Reduce the load time of LCP as small as possible without compromising the quality (e.g. webp images)  
-Deliver the initial HTML document as fast as possible (e.g. use CDN, )

CSS:

clip-path: ellipse(95% 105% at 50% -5%); //Used for drawing the ellipse for cutting the div.

ellipse(rx ry at cx cy) At cx and cy, center is located and with rx and ry radius of elipse, one ellipse is made and its intersection with div is cut