**React Redux**

1. **What is flux?**

*Flux* is an *application design paradigm* used as a replacement for the more traditional MVC pattern. It is not a framework or a library but a new kind of architecture that complements React and the concept of Unidirectional Data Flow. Facebook uses this pattern internally when working with React.

The workflow between dispatcher, stores and views components with distinct inputs and outputs as follows:

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/flux.png)

1. **What is Redux?**

*Redux* is a predictable state container for JavaScript apps based on the *Flux design pattern*. Redux can be used together with React, or with any other view library. It is tiny (about 2kB) and has no dependencies.

1. **What are the core principles of Redux?**

Redux follows three fundamental principles:

* 1. **Single source of truth:** The state of your whole application is stored in an object tree within a single store. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.
  2. **State is read-only:** The only way to change the state is to emit an action, an object describing what happened. This ensures that neither the views nor the network callbacks will ever write directly to the state.
  3. **Changes are made with pure functions:** To specify how the state tree is transformed by actions, you write reducers. Reducers are just pure functions that take the previous state and an action as parameters, and return the next state.

1. **What are the downsides of Redux compared to Flux?**

Instead of saying downsides we can say that there are few compromises of using Redux over Flux. Those are as follows:

* 1. **You will need to learn to avoid mutations:** Flux is un-opinionated about mutating data, but Redux doesn't like mutations and many packages complementary to Redux assume you never mutate the state. You can enforce this with dev-only packages like redux-immutable-state-invariant, Immutable.js, or instructing your team to write non-mutating code.
  2. **You're going to have to carefully pick your packages:** While Flux explicitly doesn't try to solve problems such as undo/redo, persistence, or forms, Redux has extension points such as middleware and store enhancers, and it has spawned a rich ecosystem.
  3. **There is no nice Flow integration yet:** Flux currently lets you do very impressive static type checks which Redux doesn't support yet.

1. **What is the difference between mapStateToProps() and mapDispatchToProps()?**

mapStateToProps() is a utility which helps your component get updated state (which is updated by some other components):

const mapStateToProps = (state) => {

return {

todos: getVisibleTodos(state.todos, state.visibilityFilter)

}

}

mapDispatchToProps() is a utility which will help your component to fire an action event (dispatching action which may cause change of application state):

const mapDispatchToProps = (dispatch) => {

return {

onTodoClick: (id) => {

dispatch(toggleTodo(id))

}

}

}

Recommend always using the “object shorthand” form for the mapDispatchToProps

Redux wrap it in another function that looks like (…args) => dispatch(onTodoClick(…args)), and pass that wrapper function as a prop to your component.

const mapDispatchToProps = ({

onTodoClick

})

1. **Can I dispatch an action in reducer?**

Dispatching an action within a reducer is an **anti-pattern**. Your reducer should be *without side effects*, simply digesting the action payload and returning a new state object. Adding listeners and dispatching actions within the reducer can lead to chained actions and other side effects.

1. **How to access Redux store outside a component?**

You just need to export the store from the module where it created with createStore(). Also, it shouldn't pollute the global window object.

store = createStore(myReducer)

export default store

1. **What are the drawbacks of MVW pattern?**
   1. DOM manipulation is very expensive which causes applications to behave slow and inefficient.
   2. Due to circular dependencies, a complicated model was created around models and views.
   3. Lot of data changes happens for collaborative applications(like Google Docs).
   4. No way to do undo (travel back in time) easily without adding so much extra code.
2. **Are there any similarities between Redux and RxJS?**

These libraries are very different for very different purposes, but there are some vague similarities.

Redux is a tool for managing state throughout the application. It is usually used as an architecture for UIs. Think of it as an alternative to (half of) Angular. RxJS is a reactive programming library. It is usually used as a tool to accomplish asynchronous tasks in JavaScript. Think of it as an alternative to Promises. Redux uses the Reactive paradigm because the Store is reactive. The Store observes actions from a distance, and changes itself. RxJS also uses the Reactive paradigm, but instead of being an architecture, it gives you basic building blocks, Observables, to accomplish this pattern.

1. **How to dispatch an action on load?**

You can dispatch an action in componentDidMount() method and in render() method you can verify the data.

class App extends Component {

componentDidMount() {

this.props.fetchData()

}

render() {

return this.props.isLoaded

? <div>{'Loaded'}</div>

: <div>{'Not Loaded'}</div>

}

}

const mapStateToProps = (state) => ({

isLoaded: state.isLoaded

})

const mapDispatchToProps = { fetchData }

export default connect(mapStateToProps, mapDispatchToProps)(App)

1. **How to use connect() from React Redux?**

You need to follow two steps to use your store in your container:

* 1. **Use mapStateToProps():** It maps the state variables from your store to the props that you specify.
  2. **Connect the above props to your container:** The object returned by the mapStateToProps function is connected to the container. You can import connect() from react-redux.
  3. import React from 'react'
  4. import { connect } from 'react-redux'
  5. class App extends React.Component {
  6. render() {
  7. return <div>{this.props.containerData}</div>
  8. }
  9. }
  10. function mapStateToProps(state) {
  11. return { containerData: state.data }
  12. }

export default connect(mapStateToProps)(App)

1. **How to reset state in Redux?**

You need to write a *root reducer* in your application which delegate handling the action to the reducer generated by combineReducers().

For example, let us take rootReducer() to return the initial state after USER\_LOGOUT action. As we know, reducers are supposed to return the initial state when they are called with undefined as the first argument, no matter the action.

const appReducer = combineReducers({

/\* your app's top-level reducers \*/

})

const rootReducer = (state, action) => {

if (action.type === 'USER\_LOGOUT') {

state = undefined

}

return appReducer(state, action)

}

In case of using redux-persist, you may also need to clean your storage. redux-persist keeps a copy of your state in a storage engine. First, you need to import the appropriate storage engine and then, to parse the state before setting it to undefined and clean each storage state key.

const appReducer = combineReducers({

/\* your app's top-level reducers \*/

})

const rootReducer = (state, action) => {

if (action.type === 'USER\_LOGOUT') {

Object.keys(state).forEach(key => {

storage.removeItem(`persist:${key}`)

})

state = undefined

}

return appReducer(state, action)

}

1. **Whats the purpose of at symbol in the Redux connect decorator?**

The **@** symbol is in fact a JavaScript expression used to signify decorators. *Decorators* make it possible to annotate and modify classes and properties at design time.

Let's take an example setting up Redux without and with a decorator.

* + **Without decorator:**
  + import React from 'react'
  + import \* as actionCreators from './actionCreators'
  + import { bindActionCreators } from 'redux'
  + import { connect } from 'react-redux'
  + function mapStateToProps(state) {
  + return { todos: state.todos }
  + }
  + function mapDispatchToProps(dispatch) {
  + return { actions: bindActionCreators(actionCreators, dispatch) }
  + }
  + class MyApp extends React.Component {
  + // ...define your main app here
  + }

export default connect(mapStateToProps, mapDispatchToProps)(MyApp)

* + **With decorator:**
  + import React from 'react'
  + import \* as actionCreators from './actionCreators'
  + import { bindActionCreators } from 'redux'
  + import { connect } from 'react-redux'
  + function mapStateToProps(state) {
  + return { todos: state.todos }
  + }
  + function mapDispatchToProps(dispatch) {
  + return { actions: bindActionCreators(actionCreators, dispatch) }
  + }
  + @connect(mapStateToProps, mapDispatchToProps)
  + export default class MyApp extends React.Component {
  + // ...define your main app here

}

The above examples are almost similar except the usage of decorator. The decorator syntax isn't built into any JavaScript runtimes yet, and is still experimental and subject to change. You can use babel for the decorators support.

1. **What is the difference between React context and React Redux?**

You can use **Context** in your application directly and is going to be great for passing down data to deeply nested components which what it was designed for.

Whereas **Redux** is much more powerful and provides a large number of features that the Context API doesn't provide. Also, React Redux uses context internally but it doesn't expose this fact in the public API.

1. **Why are Redux state functions called reducers?**

Reducers always return the accumulation of the state (based on all previous and current actions). Therefore, they act as a reducer of state. Each time a Redux reducer is called, the state and action are passed as parameters. This state is then reduced (or accumulated) based on the action, and then the next state is returned. You could *reduce* a collection of actions and an initial state (of the store) on which to perform these actions to get the resulting final state.

1. **How to make AJAX request in Redux?**

You can use redux-thunk middleware which allows you to define async actions.

Let's take an example of fetching specific account as an AJAX call using *fetch API*:

export function fetchAccount(id) {

return dispatch => {

dispatch(setLoadingAccountState()) // Show a loading spinner

fetch(`/account/${id}`, (response) => {

dispatch(doneFetchingAccount()) // Hide loading spinner

if (response.status === 200) {

dispatch(setAccount(response.json)) // Use a normal function to set the received state

} else {

dispatch(someError)

}

})

}

}

function setAccount(data) {

return { type: 'SET\_Account', data: data }

}

1. **Should I keep all component's state in Redux store?**

Keep your data in the Redux store, and the UI related state internally in the component.

1. **What is the proper way to access Redux store?**

The best way to access your store in a component is to use the connect() function, that creates a new component that wraps around your existing one. This pattern is called *Higher-Order Components*, and is generally the preferred way of extending a component's functionality in React. This allows you to map state and action creators to your component, and have them passed in automatically as your store updates.

Let's take an example of <FilterLink> component using connect:

import { connect } from 'react-redux'

import { setVisibilityFilter } from '../actions'

import Link from '../components/Link'

const mapStateToProps = (state, ownProps) => ({

active: ownProps.filter === state.visibilityFilter

})

const mapDispatchToProps = (dispatch, ownProps) => ({

onClick: () => dispatch(setVisibilityFilter(ownProps.filter))

})

const FilterLink = connect(

mapStateToProps,

mapDispatchToProps

)(Link)

export default FilterLink

Due to it having quite a few performance optimizations and generally being less likely to cause bugs, the Redux developers almost always recommend using connect() over accessing the store directly (using context API).

class MyComponent {

someMethod() {

doSomethingWith(this.context.store)

}

}

1. **What is the difference between component and container in React Redux?**

**Component** is a class or function component that describes the presentational part of your application.

**Container** is an informal term for a component that is connected to a Redux store. Containers *subscribe* to Redux state updates and *dispatch* actions, and they usually don't render DOM elements; they delegate rendering to presentational child components.

1. **What is the purpose of the constants in Redux?**

Constants allows you to easily find all usages of that specific functionality across the project when you use an IDE. It also prevents you from introducing silly bugs caused by typos – in which case, you will get a ReferenceError immediately.

Normally we will save them in a single file (constants.js or actionTypes.js).

export const ADD\_TODO = 'ADD\_TODO'

export const DELETE\_TODO = 'DELETE\_TODO'

export const EDIT\_TODO = 'EDIT\_TODO'

export const COMPLETE\_TODO = 'COMPLETE\_TODO'

export const COMPLETE\_ALL = 'COMPLETE\_ALL'

export const CLEAR\_COMPLETED = 'CLEAR\_COMPLETED'

In Redux, you use them in two places:

* 1. **During action creation:**

Let's take actions.js:

import { ADD\_TODO } from './actionTypes';

export function addTodo(text) {

return { type: ADD\_TODO, text }

}

* 1. **In reducers:**

Let's create reducer.js:

import { ADD\_TODO } from './actionTypes'

export default (state = [], action) => {

switch (action.type) {

case ADD\_TODO:

return [

...state,

{

text: action.text,

completed: false

}

];

default:

return state

}

}

1. **What are the different ways to write mapDispatchToProps()?**

There are a few ways of binding *action creators* to dispatch() in mapDispatchToProps().

Below are the possible options:

const mapDispatchToProps = (dispatch) => ({

action: () => dispatch(action())

})

const mapDispatchToProps = (dispatch) => ({

action: bindActionCreators(action, dispatch)

})

const mapDispatchToProps = { action }

The third option is just a shorthand for the first one.

1. **What is the use of the ownProps parameter in mapStateToProps() and mapDispatchToProps()?**

If the ownProps parameter is specified, React Redux will pass the props that were passed to the component into your *connect* functions. So, if you use a connected component:

import ConnectedComponent from './containers/ConnectedComponent';

<ConnectedComponent user={'john'} />

The ownProps inside your mapStateToProps() and mapDispatchToProps() functions will be an object:

{ user: 'john' }

You can use this object to decide what to return from those functions.

1. **How to structure Redux top level directories?**

Most of the applications has several top-level directories as below:

* 1. **Components**: Used for *dumb* components unaware of Redux.
  2. **Containers**: Used for *smart* components connected to Redux.
  3. **Actions**: Used for all action creators, where file names correspond to part of the app.
  4. **Reducers**: Used for all reducers, where files name correspond to state key.
  5. **Store**: Used for store initialization.

This structure works well for small and medium size apps.

1. **What is redux-saga?**

redux-saga is a library that aims to make side effects (asynchronous things like data fetching and impure things like accessing the browser cache) in React/Redux applications easier and better.

It is available in NPM:

$ npm install --save redux-saga

1. **What is the mental model of redux-saga?**

*Saga* is like a separate thread in your application, that's solely responsible for side effects. redux-saga is a redux *middleware*, which means this thread can be started, paused and cancelled from the main application with normal Redux actions, it has access to the full Redux application state and it can dispatch Redux actions as well.

1. **What are the differences between call() and put() in redux-saga?**

Both call() and put() are effect creator functions. call() function is used to create effect description, which instructs middleware to call the promise. put() function creates an effect, which instructs middleware to dispatch an action to the store.

Let's take example of how these effects work for fetching particular user data.

function\* fetchUserSaga(action) {

// `call` function accepts rest arguments, which will be passed to `api.fetchUser` function.

// Instructing middleware to call promise, it resolved value will be assigned to `userData` variable

const userData = yield call(api.fetchUser, action.userId)

// Instructing middleware to dispatch corresponding action.

yield put({

type: 'FETCH\_USER\_SUCCESS',

userData

})

}

1. **What is Redux Thunk?**

*Redux Thunk* middleware allows you to write action creators that return a function instead of an action. The thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met. The inner function receives the store methods dispatch() and getState() as parameters.

1. **What are the differences between redux-saga and redux-thunk?**

Both *Redux Thunk* and *Redux Saga* take care of dealing with side effects. In most of the scenarios, Thunk uses *Promises* to deal with them, whereas Saga uses *Generators*. Thunk is simple to use and Promises are familiar to many developers, Sagas/Generators are more powerful but you will need to learn them. But both middleware can coexist, so you can start with Thunks and introduce Sagas when/if you need them.

1. **What is Redux DevTools?**

*Redux DevTools* is a live-editing time travel environment for Redux with hot reloading, action replay, and customizable UI. If you don't want to bother with installing Redux DevTools and integrating it into your project, consider using Redux DevTools Extension for Chrome and Firefox.

1. **What are the features of Redux DevTools?**

Some of the main features of Redux DevTools are below,

* 1. Lets you inspect every state and action payload.
  2. Lets you go back in time by *cancelling* actions.
  3. If you change the reducer code, each *staged* action will be re-evaluated.
  4. If the reducers throw, you will see during which action this happened, and what the error was.
  5. With persistState() store enhancer, you can persist debug sessions across page reloads.

1. **What are Redux selectors and why to use them?**

*Selectors* are functions that take Redux state as an argument and return some data to pass to the component.

For example, to get user details from the state:

const getUserData = state => state.user.data

These selectors have two main benefits,

* 1. The selector can compute derived data, allowing Redux to store the minimal possible state
  2. The selector is not recomputed unless one of its arguments changes

1. **What is Redux Form?**

*Redux Form* works with React and Redux to enable a form in React to use Redux to store all of its state. Redux Form can be used with raw HTML5 inputs, but it also works very well with common UI frameworks like Material UI, React Widgets and React Bootstrap.

1. **What are the main features of Redux Form?**

Some of the main features of Redux Form are:

* 1. Field values persistence via Redux store.
  2. Validation (sync/async) and submission.
  3. Formatting, parsing and normalization of field values.

1. **How to add multiple middlewares to Redux?**

You can use applyMiddleware().

For example, you can add redux-thunk and logger passing them as arguments to applyMiddleware():

import { createStore, applyMiddleware } from 'redux'

const createStoreWithMiddleware = applyMiddleware(ReduxThunk, logger)(createStore)

1. **How to set initial state in Redux?**

You need to pass initial state as second argument to createStore:

const rootReducer = combineReducers({

todos: todos,

visibilityFilter: visibilityFilter

})

const initialState = {

todos: [{ id: 123, name: 'example', completed: false }]

}

const store = createStore(

rootReducer,

initialState

)

1. **How Relay is different from Redux?**

Relay is similar to Redux in that they both use a single store. The main difference is that relay only manages state originated from the server, and all access to the state is used via *GraphQL* queries (for reading data) and mutations (for changing data). Relay caches the data for you and optimizes data fetching for you, by fetching only changed data and nothing more.

1. **What is an action in Redux?**

*Actions* are plain JavaScript objects or payloads of information that send data from your application to your store. They are the only source of information for the store. Actions must have a type property that indicates the type of action being performed.

For example, let's take an action which represents adding a new todo item:

{

type: ADD\_TODO,

text: 'Add todo item'

}

**React Native**

1. **What is the difference between React Native and React?**

**React** is a JavaScript library, supporting both front end web and being run on the server, for building user interfaces and web applications.

**React Native** is a mobile framework that compiles to native app components, allowing you to build native mobile applications (iOS, Android, and Windows) in JavaScript that allows you to use React to build your components, and implements React under the hood.

1. **How to test React Native apps?**

React Native can be tested only in mobile simulators like iOS and Android. You can run the app in your mobile using expo app ([https://expo.io](https://expo.io/)) Where it syncs using QR code, your mobile and computer should be in same wireless network.

1. **How to do logging in React Native?**

You can use console.log, console.warn, etc. As of React Native v0.29 you can simply run the following to see logs in the console:

$ react-native log-ios

$ react-native log-android

1. **How to debug your React Native?**

Follow the below steps to debug React Native app:

* 1. Run your application in the iOS simulator.
  2. Press Command + D and a webpage should open up at http://localhost:8081/debugger-ui.
  3. Enable *Pause On Caught Exceptions* for a better debugging experience.
  4. Press Command + Option + I to open the Chrome Developer tools, or open it via View -> Developer -> Developer Tools.
  5. You should now be able to debug as you normally would.

**React supported libraries & Integration**

1. **What is reselect and how it works?**

*Reselect* is a **selector library** (for Redux) which uses *memoization* concept. It was originally written to compute derived data from Redux-like applications state, but it can't be tied to any architecture or library.

Reselect keeps a copy of the last inputs/outputs of the last call, and recomputes the result only if one of the inputs changes. If the the same inputs are provided twice in a row, Reselect returns the cached output. It's memoization and cache are fully customizable.

1. **What is Flow?**

*Flow* is a *static type checker* designed to find type errors in JavaScript. Flow types can express much more fine-grained distinctions than traditional type systems. For example, Flow helps you catch errors involving null, unlike most type systems.

1. **What is the difference between Flow and PropTypes?**

Flow is a *static analysis tool* (static checker) which uses a superset of the language, allowing you to add type annotations to all of your code and catch an entire class of bugs at compile time.

PropTypes is a *basic type checker* (runtime checker) which has been patched onto React. It can't check anything other than the types of the props being passed to a given component. If you want more flexible typechecking for your entire project Flow/TypeScript are appropriate choices.

1. **How to use Font Awesome icons in React?**

The below steps followed to include Font Awesome in React:

* 1. Install font-awesome:

$ npm install --save font-awesome

* 1. Import font-awesome in your index.js file:

import 'font-awesome/css/font-awesome.min.css'

* 1. Add Font Awesome classes in className:
  2. render() {
  3. return <div><i className={'fa fa-spinner'} /></div>

}

1. **What is React Dev Tools?**

*React Developer Tools* let you inspect the component hierarchy, including component props and state. It exists both as a browser extension (for Chrome and Firefox), and as a standalone app (works with other environments including Safari, IE, and React Native).

The official extensions available for different browsers or environments.

* 1. **Chrome extension**
  2. **Firefox extension**
  3. **Standalone app** (Safari, React Native, etc)

1. **Why is DevTools not loading in Chrome for local files?**

If you opened a local HTML file in your browser (file://...) then you must first open *Chrome Extensions* and check Allow access to file URLs.

1. **How to use Polymer in React?**

You need to follow below steps to use Polymer in React,

* 1. Create a Polymer element:
  2. <link rel='import' href='../../bower\_components/polymer/polymer.html' />
  3. Polymer({
  4. is: 'calender-element',
  5. ready: function() {
  6. this.textContent = 'I am a calender'
  7. }

})

* 1. Create the Polymer component HTML tag by importing it in a HTML document, e.g. import it in the index.html of your React application:

<link rel='import' href='./src/polymer-components/calender-element.html'>

* 1. Use that element in the JSX file:
  2. import React from 'react'
  3. class MyComponent extends React.Component {
  4. render() {
  5. return (
  6. <calender-element />
  7. )
  8. }
  9. }

export default MyComponent

1. **What are the advantages of React over Vue.js?**

React has the following advantages over Vue.js:

* 1. Gives more flexibility in large apps developing.
  2. Easier to test.
  3. Suitable for mobile apps creating.
  4. More information and solutions available.

**Note:** The above list of advantages are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

1. **What is the difference between React and Angular?**

Let's see the difference between React and Angular in a table format.

| **React** | **Angular** |
| --- | --- |
| React is a library and has only the View layer | Angular is a framework and has complete MVC functionality |
| React handles rendering on the server side | AngularJS renders only on the client side but Angular 2 and above renders on the server side |
| React uses JSX that looks like HTML in JS which can be confusing | Angular follows the template approach for HTML, which makes code shorter and easy to understand |
| React Native, which is a React type to build mobile applications are faster and more stable | Ionic, Angular's mobile native app is relatively less stable and slower |
| In React, data flows only in one way and hence debugging is easy | In Angular, data flows both way i.e it has two-way data binding between children and parent and hence debugging is often difficult |

**Note:** The above list of differences are purely opinionated and it vary based on the professional experience. But they are helpful as base parameters.

1. **Why React tab is not showing up in DevTools?**

When the page loads, *React DevTools* sets a global named \_\_REACT\_DEVTOOLS\_GLOBAL\_HOOK\_\_, then React communicates with that hook during initialization. If the website is not using React or if React fails to communicate with DevTools then it won't show up the tab.

1. **What are Styled Components?**

styled-components is a JavaScript library for styling React applications. It removes the mapping between styles and components, and lets you write actual CSS augmented with JavaScript.

1. **Give an example of Styled Components?**

Lets create <Title> and <Wrapper> components with specific styles for each.

import React from 'react'

import styled from 'styled-components'

// Create a <Title> component that renders an <h1> which is centered, red and sized at 1.5em

const Title = styled.h1`

font-size: 1.5em;

text-align: center;

color: palevioletred;

`

// Create a <Wrapper> component that renders a <section> with some padding and a papayawhip background

const Wrapper = styled.section`

padding: 4em;

background: papayawhip;

`

These two variables, Title and Wrapper, are now components that you can render just like any other react component.

<Wrapper>

<Title>{'Lets start first styled component!'}</Title>

</Wrapper>

1. **What is Relay?**

Relay is a JavaScript framework for providing a data layer and client-server communication to web applications using the React view layer.

1. **How to use TypeScript in create-react-app application?**

Starting from react-scripts@2.1.0 or higher, there is a built-in support for typescript. i.e, create-react-app now supports typescript natively. You can just pass --typescript option as below

npx create-react-app my-app --typescript

# or

yarn create react-app my-app --typescript

But for lower versions of react scripts, just supply --scripts-version option as react-scripts-ts while you create a new project. react-scripts-ts is a set of adjustments to take the standard create-react-app project pipeline and bring TypeScript into the mix.

Now the project layout should look like the following:

my-app/

├─ .gitignore

├─ images.d.ts

├─ node\_modules/

├─ public/

├─ src/

│ └─ ...

├─ package.json

├─ tsconfig.json

├─ tsconfig.prod.json

├─ tsconfig.test.json

└─ tslint.json

**Miscellaneous**

1. **What are the main features of Reselect library?**

Let's see the main features of Reselect library,

* 1. Selectors can compute derived data, allowing Redux to store the minimal possible state.
  2. Selectors are efficient. A selector is not recomputed unless one of its arguments changes.
  3. Selectors are composable. They can be used as input to other selectors.

1. **Give an example of Reselect usage?**

Let's take calculations and different amounts of a shipment order with the simplified usage of Reselect:

import { createSelector } from 'reselect'

const shopItemsSelector = state => state.shop.items

const taxPercentSelector = state => state.shop.taxPercent

const subtotalSelector = createSelector(

shopItemsSelector,

items => items.reduce((acc, item) => acc + item.value, 0)

)

const taxSelector = createSelector(

subtotalSelector,

taxPercentSelector,

(subtotal, taxPercent) => subtotal \* (taxPercent / 100)

)

export const totalSelector = createSelector(

subtotalSelector,

taxSelector,

(subtotal, tax) => ({ total: subtotal + tax })

)

let exampleState = {

shop: {

taxPercent: 8,

items: [

{ name: 'apple', value: 1.20 },

{ name: 'orange', value: 0.95 },

]

}

}

console.log(subtotalSelector(exampleState)) // 2.15

console.log(taxSelector(exampleState)) // 0.172

console.log(totalSelector(exampleState)) // { total: 2.322 }

1. **Does the statics object work with ES6 classes in React?**

No, statics only works with React.createClass():

someComponent= React.createClass({

statics: {

someMethod: function() {

// ..

}

}

})

But you can write statics inside ES6+ classes as below,

class Component extends React.Component {

static propTypes = {

// ...

}

static someMethod() {

// ...

}

}

or writing them outside class as below,

class Component extends React.Component {

....

}

Component.propTypes = {...}

Component.someMethod = function(){....}

1. **Can Redux only be used with React?**

Redux can be used as a data store for any UI layer. The most common usage is with React and React Native, but there are bindings available for Angular, Angular 2, Vue, Mithril, and more. Redux simply provides a subscription mechanism which can be used by any other code.

1. **Do you need to have a particular build tool to use Redux?**

Redux is originally written in ES6 and transpiled for production into ES5 with Webpack and Babel. You should be able to use it regardless of your JavaScript build process. Redux also offers a UMD build that can be used directly without any build process at all.

1. **How Redux Form initialValues get updated from state?**

You need to add enableReinitialize : true setting.

const InitializeFromStateForm = reduxForm({

form: 'initializeFromState',

enableReinitialize : true

})(UserEdit)

If your initialValues prop gets updated, your form will update too.

1. **How React PropTypes allow different types for one prop?**

You can use oneOfType() method of PropTypes.

For example, the height property can be defined with either string or number type as below:

Component.PropTypes = {

size: PropTypes.oneOfType([

PropTypes.string,

PropTypes.number

])

}

1. **Can I import an SVG file as react component?**

You can import SVG directly as component instead of loading it as a file. This feature is available with react-scripts@2.0.0 and higher.

import { ReactComponent as Logo } from './logo.svg'

const App = () => (

<div>

{/\* Logo is an actual react component \*/}

<Logo />

</div>

)

**Note**: Don't forget about the curly braces in the import.

1. **Why are inline ref callbacks or functions not recommended?**

If the ref callback is defined as an inline function, it will get called twice during updates, first with null and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one.

class UserForm extends Component {

handleSubmit = () => {

console.log("Input Value is: ", this.input.value)

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={(input) => this.input = input} /> // Access DOM input in handle submit

<button type='submit'>Submit</button>

</form>

)

}

}

But our expectation is for the ref callback to get called once, when the component mounts. One quick fix is to use the ES7 class property syntax to define the function

class UserForm extends Component {

handleSubmit = () => {

console.log("Input Value is: ", this.input.value)

}

setSearchInput = (input) => {

this.input = input

}

render () {

return (

<form onSubmit={this.handleSubmit}>

<input

type='text'

ref={this.setSearchInput} /> // Access DOM input in handle submit

<button type='submit'>Submit</button>

</form>

)

}

}

\*\*Note:\*\* In React v16.3,

1. **What is render hijacking in react?**

The concept of render hijacking is the ability to control what a component will output from another component. It actually means that you decorate your component by wrapping it into a Higher-Order component. By wrapping you can inject additional props or make other changes, which can cause changing logic of rendering. It does not actually enables hijacking, but by using HOC you make your component behave in different way.

1. **What are HOC factory implementations?**

There are two main ways of implementing HOCs in React.

* 1. Props Proxy (PP) and
  2. Inheritance Inversion (II).

But they follow different approaches for manipulating the *WrappedComponent*.

**Props Proxy**

In this approach, the render method of the HOC returns a React Element of the type of the WrappedComponent. We also pass through the props that the HOC receives, hence the name **Props Proxy**.

function ppHOC(WrappedComponent) {

return class PP extends React.Component {

render() {

return <WrappedComponent {...this.props}/>

}

}

}

**Inheritance Inversion**

In this approach, the returned HOC class (Enhancer) extends the WrappedComponent. It is called Inheritance Inversion because instead of the WrappedComponent extending some Enhancer class, it is passively extended by the Enhancer. In this way the relationship between them seems **inverse**.

function iiHOC(WrappedComponent) {

return class Enhancer extends WrappedComponent {

render() {

return super.render()

}

}

}

1. **How to pass numbers to React component?**

You should be passing the numbers via curly braces({}) where as strings in quotes

React.render(<User age={30} department={"IT"} />, document.getElementById('container'));

1. **Do I need to keep all my state into Redux? Should I ever use react internal state?**

It is up to developer decision. i.e, It is developer job to determine what kinds of state make up your application, and where each piece of state should live. Some users prefer to keep every single piece of data in Redux, to maintain a fully serializable and controlled version of their application at all times. Others prefer to keep non-critical or UI state, such as “is this dropdown currently open”, inside a component's internal state.

Below are the thumb rules to determine what kind of data should be put into Redux

* 1. Do other parts of the application care about this data?
  2. Do you need to be able to create further derived data based on this original data?
  3. Is the same data being used to drive multiple components?
  4. Is there value to you in being able to restore this state to a given point in time (ie, time travel debugging)?
  5. Do you want to cache the data (i.e, use what's in state if it's already there instead of re-requesting it)?

1. **What is the purpose of registerServiceWorker in React?**

React creates a service worker for you without any configuration by default. The service worker is a web API that helps you cache your assets and other files so that when the user is offline or on slow network, he/she can still see results on the screen, as such, it helps you build a better user experience, that's what you should know about service worker's for now. It's all about adding offline capabilities to your site.

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import registerServiceWorker from './registerServiceWorker';

ReactDOM.render(<App />, document.getElementById('root'));

registerServiceWorker();

1. **What is React memo function?**

Class components can be restricted from rendering when their input props are the same using **PureComponent or shouldComponentUpdate**. Now you can do the same with function components by wrapping them in **React.memo**.

const MyComponent = React.memo(function MyComponent(props) {

/\* only rerenders if props change \*/

});

1. **What is React lazy function?**

The React.lazy function lets you render an dynamic import as a regular component. It will automatically load the bundle containing the OtherComponent when the component gets rendered. This must return a Promise which resolves to a module with a default export containing a React component.

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {

return (

<div>

<OtherComponent />

</div>

);

}

**Note:** React.lazy and Suspense is not yet available for server-side rendering. If you want to do code-splitting in a server rendered app, we still recommend React Loadable.

1. **How to prevent unnecessary updates using setState?**

You can compare current value of the state with an existing state value and decide whether to rerender the page or not. If the values are same then you need to return **null** to stop re-rendering otherwise return the latest state value.

For example, the user profile information is conditionally rendered as follows,

getUserProfile = user => {

const latestAddress = user.address;

this.setState(state => {

if (state.address === latestAddress) {

return null;

} else {

return { title: latestAddress };

}

});

};

1. **How do you render Array, Strings and Numbers in React 16 Version?**

**Arrays**: Unlike older releases, you don't need to make sure **render** method return a single element in React16. You are able to return multiple sibling elements without a wrapping element by returning an array.

For example, let us take the below list of developers,

const ReactJSDevs = () => {

return [

<li key="1">John</li>,

<li key="2">Jackie</li>,

<li key="3">Jordan</li>

];

}

You can also merge this array of items in another array component.

const JSDevs = () => {

return (

<ul>

<li>Brad</li>

<li>Brodge</li>

<ReactJSDevs/>

<li>Brandon</li>

</ul>

);

}

**Strings and Numbers:** You can also return string and number type from the render method.

render() {

return 'Welcome to ReactJS questions';

}

// Number

render() {

return 2018;

}

1. **How to use class field declarations syntax in React classes?**

React Class Components can be made much more concise using the class field declarations. You can initialize local state without using the constructor and declare class methods by using arrow functions without the extra need to bind them.

Let's take a counter example to demonstrate class field declarations for state without using constructor and methods without binding,

class Counter extends Component {

state = { value: 0 };

handleIncrement = () => {

this.setState(prevState => ({

value: prevState.value + 1

}));

};

handleDecrement = () => {

this.setState(prevState => ({

value: prevState.value - 1

}));

};

render() {

return (

<div>

{this.state.value}

<button onClick={this.handleIncrement}>+</button>

<button onClick={this.handleDecrement}>-</button>

</div>

)

}

}

1. **What are hooks?**

Hooks is a new feature(React 16.8) that lets you use state and other React features without writing a class.

Let's see an example of useState hook example,

import { useState } from 'react';

function Example() {

// Declare a new state variable, which we'll call "count"

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

return (

<div>

<p>You clicked {count} times</p>

<button onClick={() => setCount(count + 1)}>

Click me

</button>

</div>

);

}

1. **What are the rules needs to follow for hooks?**

You need to follow two rules in order to use hooks,

* 1. Call Hooks only at the top level of your react functions. i.e, You shouldn’t call Hooks inside loops, conditions, or nested functions. This will ensure that Hooks are called in the same order each time a component renders and it preserves the state of Hooks between multiple useState and useEffect calls.
  2. Call Hooks from React Functions only. i.e, You shouldn’t call Hooks from regular JavaScript functions.

1. **How to ensure hooks followed the rules in your project?**

React team released an ESLint plugin called **eslint-plugin-react-hooks** that enforces these two rules. You can add this plugin to your project using the below command,

npm install eslint-plugin-react-hooks@next

And apply the below config in your ESLint config file,

// Your ESLint configuration

{

"plugins": [

// ...

"react-hooks"

],

"rules": {

// ...

"react-hooks/rules-of-hooks": "error"

}

}

**Note:** This plugin is intended to use in Create React App by default.

1. **What are the differences between Flux and Redux?**

Below are the major differences between Flux and Redux

| **Flux** | **Redux** |
| --- | --- |
| State is mutable | State is immutable |
| The Store contains both state and change logic | The Store and change logic are separate |
| There are multiple stores exist | There is only one store exist |
| All the stores are disconnected and flat | Single store with hierarchical reducers |
| It has a singleton dispatcher | There is no concept of dispatcher |
| React components subscribe to the store | Container components uses connect function |

1. **What are the benefits of React Router V4?**

Below are the main benefits of React Router V4 module,

* 1. In React Router v4(version 4), the API is completely about components. A router can be visualized as a single component(<BrowserRouter>) which wraps specific child router components(<Route>).
  2. You don't need to manually set history. The router module will take care history by wrapping routes with <BrowserRouter> component.
  3. The application size is reduced by adding only the specific router module(Web, core, or native)

1. **Can you describe about componentDidCatch lifecycle method signature?**

The **componentDidCatch** lifecycle method is invoked after an error has been thrown by a descendant component. The method receives two parameters,

* 1. error: - The error object which was thrown
  2. info: - An object with a componentStack key contains the information about which component threw the error.

The method structure would be as follows

componentDidCatch(error, info)

1. **In which scenarios error boundaries do not catch errors?**

Below are the cases in which error boundaries doesn't work,

* 1. Inside Event handlers
  2. Asynchronous code using **setTimeout or requestAnimationFrame** callbacks
  3. During Server side rendering
  4. When errors thrown in the error boundary code itself

1. **Why do you not need error boundaries for event handlers?**

Error boundaries do not catch errors inside event handlers.

React doesn’t need error boundaries to recover from errors in event handlers. Unlike the render method and lifecycle methods, the event handlers don’t happen during rendering. So if they throw, React still knows what to display on the screen.

If you need to catch an error inside an event handler, use the regular JavaScript try / catch statement:

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = { error: null };

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

try {

// Do something that could throw

} catch (error) {

this.setState({ error });

}

}

render() {

if (this.state.error) {

return <h1>Caught an error.</h1>

}

return <button onClick={this.handleClick}>Click Me</button>

}

}

Note that the above example is demonstrating regular JavaScript behavior and doesn’t use error boundaries.

1. **What is the difference between try catch block and error boundaries?**

Try catch block works with imperative code whereas error boundaries are meant for declarative code to render on the screen.

For example, the try catch block used for below imperative code

try {

showButton();

} catch (error) {

// ...

}

Whereas error boundaries wrap declarative code as below,

<ErrorBoundary>

<MyComponent />

</ErrorBoundary>

So if an error occurs in a **componentDidUpdate** method caused by a **setState** somewhere deep in the tree, it will still correctly propagate to the closest error boundary.

1. **What is the behavior of uncaught errors in react 16?**

In React 16, errors that were not caught by any error boundary will result in unmounting of the whole React component tree. The reason behind this decision is that it is worse to leave corrupted UI in place than to completely remove it. For example, it is worse for a payments app to display a wrong amount than to render nothing.

1. **What is the proper placement for error boundaries?**

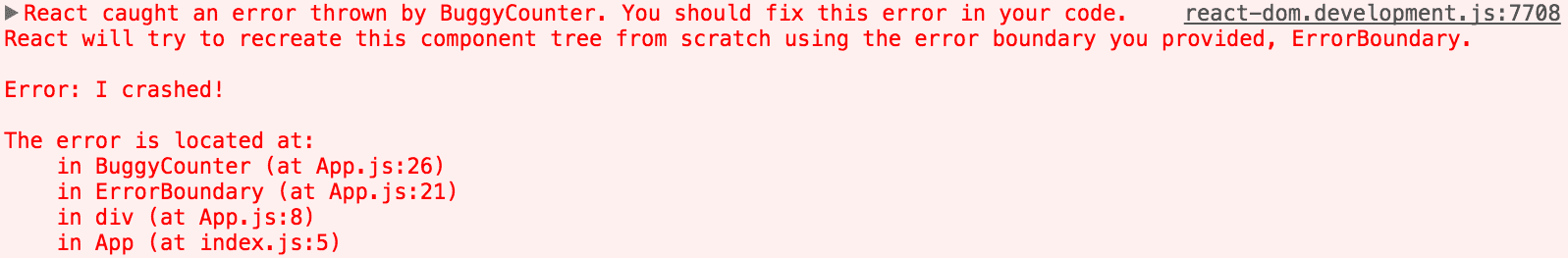
The granularity of error boundaries usage is up to the developer based on project needs. You can follow either of these approaches,

* 1. You can wrap top-level route components to display a generic error message for the entire application.
  2. You can also wrap individual components in an error boundary to protect them from crashing the rest of the application.

1. **What is the benefit of component stack trace from error boundary?**

Apart from error messages and javascript stack, React16 will display the component stack trace with file names and line numbers using error boundary concept.

For example, BuggyCounter component displays the component stack trace as below,

[](https://github.com/sudheerj/reactjs-interview-questions/blob/master/images/error_boundary.png)

1. **What is the required method to be defined for a class component?**

The render() method is the only required method in a class component. i.e, All methods other than render method are optional for a class component.

1. **What are the possible return types of render method?**

Below are the list of following types used and return from render method,

* 1. **React elements:** Elements that instruct React to render a DOM node. It includes html elements such as <div/> and user defined elements.
  2. **Arrays and fragments:** Return multiple elements to render as Arrays and Fragments to wrap multiple elements
  3. **Portals:** Render children into a different DOM subtree.
  4. **String and numbers:** Render both Strings and Numbers as text nodes in the DOM
  5. **Booleans or null:** Doesn't render anything but these types are used to conditionally render content.

1. **What is the main purpose of constructor?**

The constructor is mainly used for two purposes,

* 1. To initialize local state by assigning object to this.state
  2. For binding event handler methods to the instance For example, the below code covers both the above cases,

constructor(props) {

super(props);

// Don't call this.setState() here!

this.state = { counter: 0 };

this.handleClick = this.handleClick.bind(this);

}

1. **Is it mandatory to define constructor for React component?**

No, it is not mandatory. i.e, If you don’t initialize state and you don’t bind methods, you don’t need to implement a constructor for your React component.

1. **What are default props?**

The defaultProps are defined as a property on the component class to set the default props for the class. This is used for undefined props, but not for null props.

For example, let us create color default prop for the button component,

class MyButton extends React.Component {

// ...

}

MyButton.defaultProps = {

color: 'red'

};

If props.color is not provided then it will set the default value to 'red'. i.e, Whenever you try to access the color prop it uses default value

render() {

return <MyButton /> ; // props.color will be set to red

}

**Note:** If you provide null value then it remains null value.

1. **Why should not call setState in componentWillUnmount?**

You should not call setState() in componentWillUnmount() because once a component instance is unmounted, it will never be mounted again.

1. **What is the purpose of getDerivedStateFromError?**

This lifecycle method is invoked after an error has been thrown by a descendant component. It receives the error that was thrown as a parameter and should return a value to update state.

The signature of the lifecycle method is as follows,

static getDerivedStateFromError(error)

Let us take error boundary use case with the above lifecycle method for demonstration purpose,

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

static getDerivedStateFromError(error) {

// Update state so the next render will show the fallback UI.

return { hasError: true };

}

render() {

if (this.state.hasError) {

// You can render any custom fallback UI

return <h1>Something went wrong.</h1>;

}

return this.props.children;

}

}

1. **What is the methods order when component re-rendered?**

An update can be caused by changes to props or state. The below methods are called in the following order when a component is being re-rendered.

* 1. static getDerivedStateFromProps()
  2. shouldComponentUpdate()
  3. render()
  4. getSnapshotBeforeUpdate()
  5. componentDidUpdate()

1. **What are the methods invoked during error handling?**

Below methods are called when there is an error during rendering, in a lifecycle method, or in the constructor of any child component.

* 1. static getDerivedStateFromError()
  2. componentDidCatch()

1. **What is the purpose of displayName class property?**

The displayName string is used in debugging messages. Usually, you don’t need to set it explicitly because it’s inferred from the name of the function or class that defines the component. You might want to set it explicitly if you want to display a different name for debugging purposes or when you create a higher-order component.

For example, To ease debugging, choose a display name that communicates that it’s the result of a withSubscription HOC.

function withSubscription(WrappedComponent) {

class WithSubscription extends React.Component {/\* ... \*/}

WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;

return WithSubscription;

}

function getDisplayName(WrappedComponent) {

return WrappedComponent.displayName || WrappedComponent.name || 'Component';

}

1. **What is the browser support for react applications?**

React supports all popular browsers, including Internet Explorer 9 and above, although some polyfills are required for older browsers such as IE 9 and IE 10. If you use **es5-shim and es5-sham** polyfill then it even support old browsers that doesn't support ES5 methods.

1. **What is the purpose of unmountComponentAtNode method?**

This method is available from react-dom package and it removes a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns true if a component was unmounted and false if there was no component to unmount.

The method signature would be as follows,

ReactDOM.unmountComponentAtNode(container)

1. **What is code-splitting?**

Code-Splitting is a feature supported by bundlers like Webpack and Browserify which can create multiple bundles that can be dynamically loaded at runtime. The react project supports code splitting via dynamic import() feature.

For example, in the below code snippets, it will make moduleA.js and all its unique dependencies as a separate chunk that only loads after the user clicks the 'Load' button. **moduleA.js**

const moduleA = 'Hello';

export { moduleA };

**App.js**

import React, { Component } from 'react';

class App extends Component {

handleClick = () => {

import('./moduleA')

.then(({ moduleA }) => {

// Use moduleA

})

.catch(err => {

// Handle failure

});

};

render() {

return (

<div>

<button onClick={this.handleClick}>Load</button>

</div>

);

}

}

export default App;

1. **What is the benefit of strict mode?**

The will be helpful in the below cases

* 1. Identifying components with **unsafe lifecycle methods**.
  2. Warning about **legacy string ref** API usage.
  3. Detecting unexpected **side effects**.
  4. Detecting **legacy context** API.
  5. Warning about deprecated findDOMNode usage

1. **What are Keyed Fragments?**

The Fragments declared with the explicit <React.Fragment> syntax may have keys. The general use case is mapping a collection to an array of fragments as below,

function Glossary(props) {

return (

<dl>

{props.items.map(item => (

// Without the `key`, React will fire a key warning

<React.Fragment key={item.id}>

<dt>{item.term}</dt>

<dd>{item.description}</dd>

</React.Fragment>

))}

</dl>

);

}

**Note:** key is the only attribute that can be passed to Fragment. In the future, there might be a support for additional attributes, such as event handlers.

1. **Does React support all HTML attributes?**

As of React 16, both standard or custom DOM attributes are fully supported. Since React components often take both custom and DOM-related props, React uses the camelCase convention just like the DOM APIs.

Let us take few props with respect to standard HTML attributes,

<div tabIndex="-1" /> // Just like node.tabIndex DOM API

<div className="Button" /> // Just like node.className DOM API

<input readOnly={true} /> // Just like node.readOnly DOM API

These props work similarly to the corresponding HTML attributes, with the exception of the special cases. It also support all SVG attributes.

1. **What are the limitations with HOCs?**

Higher-order components come with a few caveats apart from its benefits. Below are the few listed in an order,

* 1. **Don’t use HOCs inside the render method:** It is not recommended to apply a HOC to a component within the render method of a component.
  2. render() {
  3. // A new version of EnhancedComponent is created on every render
  4. // EnhancedComponent1 !== EnhancedComponent2
  5. const EnhancedComponent = enhance(MyComponent);
  6. // That causes the entire subtree to unmount/remount each time!
  7. return <EnhancedComponent />;

}

The above code impact performance by remounting a component that causes the state of that component and all of its children to be lost. Instead, apply HOCs outside the component definition so that the resulting component is created only once.

* 1. **Static methods must be copied over:** When you apply a HOC to a component the new component does not have any of the static methods of the original component
  2. // Define a static method
  3. WrappedComponent.staticMethod = function() {/\*...\*/}
  4. // Now apply a HOC
  5. const EnhancedComponent = enhance(WrappedComponent);
  6. // The enhanced component has no static method

typeof EnhancedComponent.staticMethod === 'undefined' // true

You can overcome this by copying the methods onto the container before returning it,

function enhance(WrappedComponent) {

class Enhance extends React.Component {/\*...\*/}

// Must know exactly which method(s) to copy :(

Enhance.staticMethod = WrappedComponent.staticMethod;

return Enhance;

}

* 1. **Refs aren’t passed through:** For HOCs you need to pass through all props to the wrapped component but this does not work for refs. This is because ref is not really a prop similar to key. In this case you need to use the React.forwardRef API

1. **How to debug forwardRefs in DevTools?**

**React.forwardRef** accepts a render function as parameter and DevTools uses this function to determine what to display for the ref forwarding component.

For example, If you don't name the render function or not using displayName property then it will appear as ”ForwardRef” in the DevTools,

const WrappedComponent = React.forwardRef((props, ref) => {

return <LogProps {...props} forwardedRef={ref} />;

});

But If you name the render function then it will appear as **”ForwardRef(myFunction)”**

const WrappedComponent = React.forwardRef(

function myFunction(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

);

As an alternative, You can also set displayName property for forwardRef function,

function logProps(Component) {

class LogProps extends React.Component {

// ...

}

function forwardRef(props, ref) {

return <LogProps {...props} forwardedRef={ref} />;

}

// Give this component a more helpful display name in DevTools.

// e.g. "ForwardRef(logProps(MyComponent))"

const name = Component.displayName || Component.name;

forwardRef.displayName = `logProps(${name})`;

return React.forwardRef(forwardRef);

}

1. **When component props defaults to true?**

If you pass no value for a prop, it defaults to true. This behavior is available so that it matches the behavior of HTML.

For example, below expressions are equivalent,

<MyInput autocomplete />

<MyInput autocomplete={true} />

**Note:** It is not recommended to use this approach because it can be confused with the ES6 object shorthand (example, {name} which is short for {name: name})

1. **What is NextJS and major features of it?**

Next.js is a popular and lightweight framework for static and server‑rendered applications built with React. It also provides styling and routing solutions. Below are the major features provided by NextJS,

* 1. Server-rendered by default
  2. Automatic code splitting for faster page loads
  3. Simple client-side routing (page based)
  4. Webpack-based dev environment which supports (HMR)
  5. Able to implement with Express or any other Node.js HTTP server
  6. Customizable with your own Babel and Webpack configurations

1. **How do you pass an event handler to a component?**

You can pass event handlers and other functions as props to child components. It can be used in child component as below,

<button onClick={this.handleClick}>

1. **Is it good to use arrow functions in render methods?**

Yes, You can use. It is often the easiest way to pass parameters to callback functions. But you need to optimize the performance while using it.

class Foo extends Component {

handleClick() {

console.log('Click happened');

}

render() {

return <button onClick={() => this.handleClick()}>Click Me</button>;

}

}

**Note:** Using an arrow function in render method creates a new function each time the component renders, which may have performance implications

1. **How to prevent a function from being called multiple times?**

If you use an event handler such as **onClick or onScroll** and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be achieved in the below possible ways,

* 1. **Throttling:** Changes based on a time based frequency. For example, it can be used using \_.throttle lodash function
  2. **Debouncing:** Publish changes after a period of inactivity. For example, it can be used using \_.debounce lodash function
  3. **RequestAnimationFrame throttling:** Changes based on requestAnimationFrame. For example, it can be used using raf-schd lodash function

1. **How JSX prevents Injection Attacks?**

React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that’s not explicitly written in your application. Everything is converted to a string before being rendered.

For example, you can embed user input as below,

const name = response.potentiallyMaliciousInput;

const element = <h1>{name}</h1>;

This way you can prevent XSS(Cross-site-scripting) attacks in the application.

1. **How do you update rendered elements?**

You can update UI(represented by rendered element) by passing the newly created element to ReactDOM's render method.

For example, lets take a ticking clock example, where it updates the time by calling render method multiple times,

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root'));

}

setInterval(tick, 1000);

1. **How do you say that props are read only?**

When you declare a component as a function or a class, it must never modify its own props.

Let us take a below capital function,

function capital(amount, interest) {

return amount + interest;

}

The above function is called “pure” because it does not attempt to change their inputs, and always return the same result for the same inputs. Hence, React has a single rule saying "All React components must act like pure functions with respect to their props."

1. **How do you say that state updates are merged?**

When you call setState() in the component, React merges the object you provide into the current state.

For example, let us take a facebook user with posts and comments details as state variables,

constructor(props) {

super(props);

this.state = {

posts: [],

comments: []

};

}

Now you can update them independently with separate setState() calls as below,

componentDidMount() {

fetchPosts().then(response => {

this.setState({

posts: response.posts

});

});

fetchComments().then(response => {

this.setState({

comments: response.comments

});

});

}

As mentioned in the above code snippets, this.setState({comments}) updates only comments variable without modifying or replacing posts variable.

1. **How do you pass arguments to an event handler?**

During iterations or loops, it is common to pass an extra parameter to an event handler. This can be achieved through arrow functions or bind method.

Let us take an example of user details updated in a grid,

<button onClick={(e) => this.updateUser(userId, e)}>Update User details</button>

<button onClick={this.updateUser.bind(this, userId)}>Update User details</button>

In both the approaches, the synthetic argument e is passed as a second argument. You need to pass it explicitly for arrow functions and it forwarded automatically for bind method.

1. **How to prevent component from rendering?**

You can prevent component from rendering by returning null based on specific condition. This way it can conditionally render component.

function Greeting(props) {

if (!props.loggedIn) {

return null;

}

return (

<div className="greeting">

welcome, {props.name}

</div>

);

}

class User extends React.Component {

constructor(props) {

super(props);

this.state = {loggedIn: false, name: 'John'};

}

render() {

return (

<div>

//Prevent component render if it is not loggedIn

<Greeting loggedIn={this.state.loggedIn} />

<UserDetails name={this.state.name}>

</div>

);

}

In the above example, the greeting component skips its rendering section by applying condition and returning null value.

1. **What are the conditions to safely use the index as a key?**

There are three conditions to make sure, it is safe use the index as a key.

* 1. The list and items are static– they are not computed and do not change
  2. The items in the list have no ids
  3. The list is never reordered or filtered.

1. **Is it keys should be globally unique?**

Keys used within arrays should be unique among their siblings but they don’t need to be globally unique. i.e, You can use the same keys with two different arrays.

For example, the below book component uses two arrays with different arrays,

function Book(props) {

const index = (

<ul>

{props.pages.map((page) =>

<li key={page.id}>

{page.title}

</li>

)}

</ul>

);

const content = props.pages.map((page) =>

<div key={page.id}>

<h3>{page.title}</h3>

<p>{page.content}</p>

<p>{page.pageNumber}</p>

</div>

);

return (

<div>

{index}

<hr />

{content}

</div>

);

}

1. **What is the popular choice for form handling?**

Formik is a form library for react which provides solutions such as validation, keeping track of the visited fields, and handling form submission.

In detail, You can categorize them as follows,

* 1. Getting values in and out of form state
  2. Validation and error messages
  3. Handling form submission

It is used to create a scalable, performant, form helper with a minimal API to solve annoying stuff.

1. **What are the advantages of formik over redux form library?**

Below are the main reasons to recommend formik over redux form library,

* 1. The form state is inherently short-term and local, so tracking it in Redux (or any kind of Flux library) is unnecessary.
  2. Redux-Form calls your entire top-level Redux reducer multiple times ON EVERY SINGLE KEYSTROKE. This way it increases input latency for large apps.
  3. Redux-Form is 22.5 kB minified gzipped whereas Formik is 12.7 kB

1. **Why do you not required to use inheritance?**

In React, it is recommend using composition instead of inheritance to reuse code between components. Both Props and composition give you all the flexibility you need to customize a component’s look and behavior in an explicit and safe way. Whereas, If you want to reuse non-UI functionality between components, it is suggested to extracting it into a separate JavaScript module. Later components import it and use that function, object, or a class, without extending it.

1. **Can I use web components in react application?**

Yes, you can use web components in a react application. Even though many developers won't use this combination, it may require especially if you are using third-party UI components that are written using Web Components.

For example, let us use Vaadin date picker web component as below,

import React, { Component } from 'react';

import './App.css';

import '@vaadin/vaadin-date-picker';

class App extends Component {

render() {

return (

<div className="App">

<vaadin-date-picker label="When were you born?"></vaadin-date-picker>

</div>

);

}

}

export default App;

1. **What is dynamic import?**

The dynamic import() syntax is a ECMAScript proposal not currently part of the language standard. It is expected to be accepted in the near future. You can achieve code-splitting into your app using dynamic import.

Let's take an example of addition,

* 1. **Normal Import**

import { add } from './math';

console.log(add(10, 20));

* 1. **Dynamic Import**

import("./math").then(math => {

console.log(math.add(10, 20));

});

1. **What are loadable components?**

If you want to do code-splitting in a server rendered app, it is recommend to use Loadable Components because React.lazy and Suspense is not yet available for server-side rendering. Loadable lets you render a dynamic import as a regular component.

Lets take an example,

import loadable from '@loadable/component'

const OtherComponent = loadable(() => import('./OtherComponent'))

function MyComponent() {

return (

<div>

<OtherComponent />

</div>

)

}

Now OtherComponent will be loaded in a separated bundle

1. **What is suspense component?**

If the module containing the dynamic import is not yet loaded by the time parent component renders, you must show some fallback content while you’re waiting for it to load using a loading indicator. This can be done using **Suspense** component.

For example, the below code uses suspense component,

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {

return (

<div>

<Suspense fallback={<div>Loading...</div>}>

<OtherComponent />

</Suspense>

</div>

);

}

As mentioned in the above code, Suspense is wrapped above the lazy component.

1. **What is route based code splitting?**

One of the best place to do code splitting is with routes. The entire page is going to re-render at once so users are unlikely to interact with other elements in the page at the same time. Due to this, the user experience won't be disturbed.

Let us take an example of route based website using libraries like React Router with React.lazy,

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import React, { Suspense, lazy } from 'react';

const Home = lazy(() => import('./routes/Home'));

const About = lazy(() => import('./routes/About'));

const App = () => (

<Router>

<Suspense fallback={<div>Loading...</div>}>

<Switch>

<Route exact path="/" component={Home}/>

<Route path="/about" component={About}/>

</Switch>

</Suspense>

</Router>

);

In the above code, the code splitting will happen at each route level.

1. **Give an example on How to use context?**

**Context** is designed to share data that can be considered **global** for a tree of React components.

For example, in the code below lets manually thread through a “theme” prop in order to style the Button component.

//Lets create a context with a default theme value "luna"

const ThemeContext = React.createContext('luna');

// Create App component where it uses provider to pass theme value in the tree

class App extends React.Component {

render() {

return (

<ThemeContext.Provider value="nova">

<Toolbar />

</ThemeContext.Provider>

);

}

}

// A middle component where you don't need to pass theme prop anymore

function Toolbar(props) {

return (

<div>

<ThemedButton />

</div>

);

}

// Lets read theme value in the button component to use

class ThemedButton extends React.Component {

static contextType = ThemeContext;

render() {

return <Button theme={this.context} />;

}

}

1. **What is the purpose of default value in context?**

The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them.

Below code snippet provides default theme value as Luna.

const MyContext = React.createContext(defaultValue);

1. **How do you use contextType?**

ContextType is used to consume the context object. The contextType property can be used in two ways,

* 1. **contextType as property of class:** The contextType property on a class can be assigned a Context object created by React.createContext(). After that, you can consume the nearest current value of that Context type using this.context in any of the lifecycle methods and render function.

Lets assign contextType property on MyClass as below,

class MyClass extends React.Component {

componentDidMount() {

let value = this.context;

/\* perform a side-effect at mount using the value of MyContext \*/

}

componentDidUpdate() {

let value = this.context;

/\* ... \*/

}

componentWillUnmount() {

let value = this.context;

/\* ... \*/

}

render() {

let value = this.context;

/\* render something based on the value of MyContext \*/

}

}

MyClass.contextType = MyContext;

* 1. **Static field** You can use a static class field to initialize your contextType using public class field syntax.
  2. class MyClass extends React.Component {
  3. static contextType = MyContext;
  4. render() {
  5. let value = this.context;
  6. /\* render something based on the value \*/
  7. }

}

1. **What is a consumer?**

A Consumer is a React component that subscribes to context changes. It requires a function as a child which receives current context value as argument and returns a react node. The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree.

Lets take a simple example,

<MyContext.Consumer>

{value => /\* render something based on the context value \*/}

</MyContext.Consumer>

1. **How do you solve performance corner cases while using context?**

The context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider’s parent re-renders.

For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for value.

class App extends React.Component {

render() {

return (

<Provider value={{something: 'something'}}>

<Toolbar />

</Provider>

);

}

}

This can be solved by lifting up the value to parent state,

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

value: {something: 'something'},

};

}

render() {

return (

<Provider value={this.state.value}>

<Toolbar />

</Provider>

);

}

}

1. **What is the purpose of forward ref in HOCs?**

Refs will not get passed through because ref is not a prop. It handled differently by React just like **key**. If you add a ref to a HOC, the ref will refer to the outermost container component, not the wrapped component. In this case, you can use Forward Ref API. For example, we can explicitly forward refs to the inner FancyButton component using the React.forwardRef API.

The below HOC logs all props,

function logProps(Component) {

class LogProps extends React.Component {

componentDidUpdate(prevProps) {

console.log('old props:', prevProps);

console.log('new props:', this.props);

}

render() {

const {forwardedRef, ...rest} = this.props;

// Assign the custom prop "forwardedRef" as a ref

return <Component ref={forwardedRef} {...rest} />;

}

}

return React.forwardRef((props, ref) => {

return <LogProps {...props} forwardedRef={ref} />;

});

}

Let's use this HOC to log all props that get passed to our “fancy button” component,

class FancyButton extends React.Component {

focus() {

// ...

}

// ...

}

export default logProps(FancyButton);

Now lets create a ref and pass it to FancyButton component. In this case, you can set focus to button element.

import FancyButton from './FancyButton';

const ref = React.createRef();

ref.current.focus();

<FancyButton

label="Click Me"

handleClick={handleClick}

ref={ref}

/>;

1. **Is it ref argument available for all functions or class components?**

Regular function or class components don’t receive the ref argument, and ref is not available in props either. The second ref argument only exists when you define a component with React.forwardRef call.

1. **Why do you need additional care for component libraries while using forward refs?**

When you start using forwardRef in a component library, you should treat it as a breaking change and release a new major version of your library. This is because your library likely has a different behavior such as what refs get assigned to, and what types are exported. These changes can break apps and other libraries that depend on the old behavior.

1. **How to create react class components without ES6?**

If you don’t use ES6 then you may need to use the create-react-class module instead. For default props, you need to define getDefaultProps() as a function on the passed object. Whereas for initial state, you have to provide a separate getInitialState method that returns the initial state.

var Greeting = createReactClass({

getDefaultProps: function() {

return {

name: 'Jhohn'

};

},

getInitialState: function() {

return {message: this.props.message};

},

handleClick: function() {

console.log(this.state.message);

},

render: function() {

return <h1>Hello, {this.props.name}</h1>;

}

});

**Note:** If you use createReactClass then auto binding is available for all methods. i.e, You don't need to use .bind(this) with in constructor for event handlers.

1. **Is it possible to use react without JSX?**

Yes, JSX is not mandatory for using React. Actually it is convenient when you don’t want to set up compilation in your build environment. Each JSX element is just syntactic sugar for calling React.createElement(component, props, ...children).

For example, let us take a greeting example with JSX,

class Greeting extends React.Component {

render() {

return <div>Hello {this.props.message}</div>;

}

}

ReactDOM.render(

<Greeting message="World" />,

document.getElementById('root')

);

You can write the same code without JSX as below,

class Greeting extends React.Component {

render() {

return React.createElement('div', null, `Hello ${this.props.message}`);

}

}

ReactDOM.render(

React.createElement(Greeting, {message: 'World'}, null),

document.getElementById('root')

);

1. **What is diffing algorithm?**

React needs to use algorithms to find out how to efficiently update the UI to match the most recent tree. The diffing algorithms is generating the minimum number of operations to transform one tree into another. However, the algorithms have a complexity in the order of O(n3) where n is the number of elements in the tree.

In this case, for displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic O(n) algorithm based on two assumptions:

* 1. Two elements of different types will produce different trees.
  2. The developer can hint at which child elements may be stable across different renders with a key prop.

1. **What are the rules covered by diffing algorithm?**

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements. It covers the below rules during reconciliation algorithm,

* 1. **Elements Of Different Types:** Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. For example, elements to , or from to of different types lead a full rebuild.
  2. **DOM Elements Of The Same Type:** When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. Lets take an example with same DOM elements except className attribute,
  3. <div className="show" title="ReactJS" />

<div className="hide" title="ReactJS" />

* 1. **Component Elements Of The Same Type:** When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls componentWillReceiveProps() and componentWillUpdate() on the underlying instance. After that, the render() method is called and the diff algorithm recurses on the previous result and the new result.
  2. **Recursing On Children:** when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there’s a difference. For example, when adding an element at the end of the children, converting between these two trees works well.
  3. <ul>
  4. <li>first</li>
  5. <li>second</li>
  6. </ul>
  7. <ul>
  8. <li>first</li>
  9. <li>second</li>
  10. <li>third</li>

</ul>

* 1. **Handling keys:** React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a key can make the tree conversion efficient,

<ul>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

<ul>

<li key="2014">Connecticut</li>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

1. **When do you need to use refs?**

There are few use cases to go for refs,

* 1. Managing focus, text selection, or media playback.
  2. Triggering imperative animations.
  3. Integrating with third-party DOM libraries.

1. **Is it prop must be named as render for render props?**

Even though the pattern named render props, you don’t have to use a prop named render to use this pattern. i.e, Any prop that is a function that a component uses to know what to render is technically a “render prop”. Lets take an example with the children prop for render props,

<Mouse children={mouse => (

<p>The mouse position is {mouse.x}, {mouse.y}</p>

)}/>

Actually children prop doesn’t need to be named in the list of “attributes” in JSX element. Instead, you can keep it directly inside element,

<Mouse>

{mouse => (

<p>The mouse position is {mouse.x}, {mouse.y}</p>

)}

</Mouse>

While using this above technique(without any name), explicitly state that children should be a function in your propTypes.

Mouse.propTypes = {

children: PropTypes.func.isRequired

};

1. **What are the problems of using render props with pure components?**

If you create a function inside a render method, it negates the purpose of pure component. Because the shallow prop comparison will always return false for new props, and each render in this case will generate a new value for the render prop. You can solve this issue by defining the render function as instance method.

1. **How do you create HOC using render props?**

You can implement most higher-order components (HOC) using a regular component with a render prop. For example, if you would prefer to have a withMouse HOC instead of a component, you could easily create one using a regular with a render prop.

function withMouse(Component) {

return class extends React.Component {

render() {

return (

<Mouse render={mouse => (

<Component {...this.props} mouse={mouse} />

)}/>

);

}

}

}

This way render props gives the flexibility of using either pattern.

1. **What is windowing technique?**

Windowing is a technique that only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created. If your application renders long lists of data then this technique is recommended. Both react-window and react-virtualized are popular windowing libraries which provides several reusable components for displaying lists, grids, and tabular data.

1. **How do you print falsy values in JSX?**

The falsy values such as false, null, undefined, and true are valid children but they don't render anything. If you still want to display them then you need to convert it to string. Let's take an example on how to convert to a string,

<div>

My JavaScript variable is {String(myVariable)}.

</div>

1. **What is the typical use case of portals?**

React portals are very useful when a parent component has overflow: hidden or has properties that affect the stacking context(z-index,position,opacity etc styles) and you need to visually “break out” of its container.

For example, dialogs, global message notifications, hovercards, and tooltips.

1. **How do you set default value for uncontrolled component?**

In React, the value attribute on form elements will override the value in the DOM. With an uncontrolled component, you might want React to specify the initial value, but leave subsequent updates uncontrolled. To handle this case, you can specify a **defaultValue** attribute instead of **value**.

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

User Name:

<input

defaultValue="John"

type="text"

ref={this.input} />

</label>

<input type="submit" value="Submit" />

</form>

);

}

The same applies for select and textArea inputs. But you need to use **defaultChecked** for checkbox and radio inputs.

1. **What is your favorite React stack?**

Even though the tech stack varies from developer to developer, the most popular stack is used in react boilerplate project code. It mainly uses Redux and redux-saga for state management and asynchronous side-effects, react-router for routing purpose, styled-components for styling react components, axios for invoking REST api, and other supported stack such as webpack, reselect, ESNext, Babel. You can clone the project <https://github.com/react-boilerplate/react-boilerplate> and start working on any new react project.

1. **What is the difference between Real DOM and Virtual DOM?**

Below are the main differences between Real DOM and Virtual DOM,

| **Real DOM** | **Virtual DOM** |
| --- | --- |
| Updates are slow | Updates are fast |
| DOM manipulation is very expensive. | DOM manipulation is very easy |
| You can update HTML directly. | You Can’t directly update HTML |
| It causes too much of memory wastage | There is no memory wastage |
| Creates a new DOM if element updates | It updates the JSX if element update |

1. **How to add Bootstrap to a react application?**

Bootstrap can be added to your React app in a three possible ways,

* 1. Using the Bootstrap CDN: This is the easiest way to add bootstrap. Add both bootstrap CSS and JS resources in a head tag.
  2. Bootstrap as Dependency: If you are using a build tool or a module bundler such as Webpack, then this is the preferred option for adding Bootstrap to your React application

npm install bootstrap

* 1. React Bootstrap Package: In this case, you can add Bootstrap to our React app is by using a package that has rebuilt Bootstrap components to work particularly as React components. Below packages are popular in this category,
     1. react-bootstrap
     2. reactstrap

1. **Can you list down top websites or applications using react as front end framework?**

Below are the top 10 websites using React as their front-end framework,

* 1. Facebook
  2. Uber
  3. Instagram
  4. WhatsApp
  5. Khan Academy
  6. Airbnb
  7. Dropbox
  8. Flipboard
  9. Netflix
  10. PayPal

1. **Is it recommended to use CSS In JS technique in React?**

React does not have any opinion about how styles are defined but if you are a beginner then good starting point is to define your styles in a separate \*.css file as usual and refer to them using className. This functionality is not part of React but came from third-party libraries. But If you want to try a different approach(CSS-In-JS) then styled-components library is a good option.

1. **Do I need to rewrite all my class components with hooks?**

No. But you can try Hooks in a few components(or new components) without rewriting any existing code. Because there are no plans to remove classes in ReactJS.

1. **How to fetch data with React Hooks?**

The effect hook called useEffect is used to fetch the data with axios from the API and to set the data in the local state of the component with the state hook’s update function.

Let's take an example in which it fetches list of react articles from the API

import React, { useState, useEffect } from 'react';

import axios from 'axios';

function App() {

const [data, setData] = useState({ hits: [] });

useEffect(async () => {

const result = await axios(

'http://hn.algolia.com/api/v1/search?query=react',

);

setData(result.data);

}, []);

return (

<ul>

{data.hits.map(item => (

<li key={item.objectID}>

<a href={item.url}>{item.title}</a>

</li>

))}

</ul>

);

}

export default App;

Remember we provided an empty array as second argument to the effect hook to avoid activating it on component updates but only for the mounting of the component. i.e, It fetches only for component mount.

1. **Is Hooks cover all use cases for classes?**

Hooks doesn't cover all use cases of classes but there is a plan to add them soon. Currently there are no Hook equivalents to the uncommon **getSnapshotBeforeUpdate** and **componentDidCatch** lifecycles yet.

1. **What is the stable release for hooks support?**

React includes a stable implementation of React Hooks in 16.8 release for below packages

* 1. React DOM
  2. React DOM Server
  3. React Test Renderer
  4. React Shallow Renderer

1. **Why do we use array destructuring (square brackets notation) in useState?**

When we declare a state variable with useState, it returns a pair — an array with two items. The first item is the current value, and the second is a function that updates the value. Using [0] and [1] to access them is a bit confusing because they have a specific meaning. This is why we use array destructuring instead.

For example, the array index access would look as follows:

var userStateVariable = useState('userProfile'); // Returns an array pair

var user = userStateVariable[0]; // Access first item

var setUser = userStateVariable[1]; // Access second item

Whereas with array destructuring the variables can be accessed as follows:

const [user, setUser] = useState('userProfile');

1. **What are the sources used for introducing hooks?**

Hooks got the ideas from several different sources. Below are some of them,

* 1. Previous experiments with functional APIs in the react-future repository
  2. Community experiments with render prop APIs such as Reactions Component
  3. State variables and state cells in DisplayScript.
  4. Subscriptions in Rx.
  5. Reducer components in ReasonReact.

1. **How do you access imperative API of web components?**

Web Components often expose an imperative API to implement its functions. You will need to use a **ref** to interact with the DOM node directly if you want to access imperative API of a web component. But if you are using third-party Web Components, the best solution is to write a React component that behaves as a **wrapper** for your Web Component.

1. **What is formik?**

Formik is a small react form library that helps you with the three major problems,

* 1. Getting values in and out of form state
  2. Validation and error messages
  3. Handling form submission

1. **What are typical middleware choices for handling asynchronous calls in Redux?**

Some of the popular middleware choices for handling asynchronous calls in Redux eco system are Redux Thunk, Redux Promise, Redux Saga.

1. **Do browsers understand JSX code?**

No, browsers can't understand JSX code. You need a transpiler to convert your JSX to regular Javascript that browsers can understand. The most widely used transpiler right now is Babel.

1. **Describe about data flow in react?**

React implements one-way reactive data flow using props which reduce boilerplate and is easier to understand than traditional two-way data binding.

1. **What is react scripts?**

The react-scripts package is a set of scripts from the create-react-app starter pack which helps you kick off projects without configuring. The react-scripts start command sets up the development environment and starts a server, as well as hot module reloading.

1. **What are the features of create react app?**

Below are the list of some of the features provided by create react app.

* 1. React, JSX, ES6, Typescript and Flow syntax support.
  2. Autoprefixed CSS
  3. CSS Reset/Normalize
  4. A live development server
  5. A fast interactive unit test runner with built-in support for coverage reporting
  6. A build script to bundle JS, CSS, and images for production, with hashes and sourcemaps
  7. An offline-first service worker and a web app manifest, meeting all the Progressive Web App criteria.

1. **What is the purpose of renderToNodeStream method?**

The ReactDOMServer#renderToNodeStream method is used to generate HTML on the server and send the markup down on the initial request for faster page loads. It also helps search engines to crawl your pages easily for SEO purposes. **Note:** Remember this method is not available in the browser but only server.

1. **What is MobX?**

MobX is a simple, scalable and battle tested state management solution for applying functional reactive programming (TFRP). For reactJs application, you need to install below packages,

npm install mobx --save

npm install mobx-react --save

1. **What are the differences between Redux and MobX?**

Below are the main differences between Redux and MobX,

| **Topic** | **Redux** | **MobX** |
| --- | --- | --- |
| Definition | It is a javascript library for managing the application state | It is a library for reactively managing the state of your applications |
| Programming | It is mainly written in ES6 | It is written in JavaScript(ES5) |
| Data Store | There is only one large store exist for data storage | There is more than one store for storage |
| Usage | Mainly used for large and complex applications | Used for simple applications |
| Performance | Need to be improved | Provides better performance |
| How it stores | Uses JS Object to store | Uses observable to store the data |

1. **Should I learn ES6 before learning ReactJS?**

No, you don’t have to learn es2015/es6 to learn react. But you may find many resources or React ecosystem uses ES6 extensively. Let's see some of the frequently used ES6 features,

* 1. **Destructuring:** To get props and use them in a component
  2. // in es 5
  3. var someData = this.props.someData
  4. var dispatch = this.props.dispatch
  5. // in es6

const { someData, dispatch } = this.props

* 1. Spread operator: Helps in passing props down into a component
  2. // in es 5
  3. <SomeComponent someData={this.props.someData} dispatch={this.props.dispatch} />
  4. // in es6

<SomeComponent {...this.props} />

* 1. Arrow functions: Makes compact syntax
  2. // es 5
  3. var users = usersList.map(function (user) {
  4. return <li>{user.name}</li>
  5. })
  6. // es 6

const users = usersList.map(user => <li>{user.name}</li>);

1. **What is Concurrent Rendering?**

The Concurrent rendering makes React apps to be more responsive by rendering component trees without blocking the main UI thread. It allows React to interrupt a long-running render to handle a high-priority event. i.e, When you enabled concurrent Mode, React will keep an eye on other tasks that need to be done, and if there's something with a higher priority it will pause what it is currently rendering and let the other task finish first. You can enable this in two ways,

// 1. Part of an app by wrapping with ConcurrentMode

<React.unstable\_ConcurrentMode>

<Something />

</React.unstable\_ConcurrentMode>

// 2. Whole app using createRoot

ReactDOM.unstable\_createRoot(domNode).render(<App />);

1. **What is the difference between async mode and concurrent mode?**

Both refers the same thing. Previously concurrent Mode being referred to as "Async Mode" by React team. The name has been changed to highlight React’s ability to perform work on different priority levels. So it avoids the confusion from other approaches to Async Rendering.

1. **Can I use javascript urls in react16.9?**

Yes, you can use javascript: URLs but it will log a warning in the console. Because URLs starting with javascript: are dangerous by including unsanitized output in a tag like <a href> and create a security hole.

const companyProfile = {

website: "javascript: alert('Your website is hacked')",

};

// It will log a warning

<a href={companyProfile.website}>More details</a>

Remember that the future versions will throw an error for javascript URLs.

1. **What is the purpose of eslint plugin for hooks?**

The ESLint plugin enforces rules of Hooks to avoid bugs. It assumes that any function starting with ”use” and a capital letter right after it is a Hook. In particular, the rule enforces that,

* 1. Calls to Hooks are either inside a PascalCase function (assumed to be a component) or another useSomething function (assumed to be a custom Hook).
  2. Hooks are called in the same order on every render.

1. **What is the difference between Imperative and Declarative in React?**

Imagine a simple UI component, such as a "Like" button. When you tap it, it turns blue if it was previously grey, and grey if it was previously blue.

The imperative way of doing this would be:

if( user.likes() ) {

if( hasBlue() ) {

removeBlue();

addGrey();

} else {

removeGrey();

addBlue();

}

}

Basically, you have to check what is currently on the screen and handle all the changes necessary to redraw it with the current state, including undoing the changes from the previous state. You can imagine how complex this could be in a real-world scenario.

In contrast, the declarative approach would be:

if( this.state.liked ) {

return <blueLike />;

} else {

return <greyLike />;

}

Because the declarative approach separates concerns, this part of it only needs to handle how the UI should look in a sepecific state, and is therefore much simpler to understand.

1. **What are the benefits of using typescript with reactjs?**

Below are some of the benefits of using typescript with Reactjs,

* 1. It is possible to use latest JavaScript features
  2. Use of interfaces for complex type definitions
  3. IDEs such as VS Code was made for TypeScript
  4. Avoid bugs with the ease of readability and Validation

1. **How do you make sure that user remains authenticated on page refresh while using Context API State Management?**

When a user logs in and reload, to persist the state generally we add the load user action in the useEffect hooks in the main App.js. While using Redux, loadUser action can be easily accessed.

**App.js**

import {loadUser} from '../actions/auth';

store.dispatch(loadUser());

* But while using **Context API**, to access context in App.js, wrap the AuthState in index.js so that App.js can access the auth context. Now whenever the page reloads, no matter what route you are on, the user will be authenticated as **loadUser** action will be triggered on each re-render.

**index.js**

import React from 'react';

import ReactDOM from 'react-dom';

import App from './App';

import AuthState from './context/auth/AuthState'

ReactDOM.render(

<React.StrictMode>

<AuthState>

<App />

</AuthState>

</React.StrictMode>,

document.getElementById('root')

);

**App.js**

const authContext = useContext(AuthContext);

const { loadUser } = authContext;

useEffect(() => {

loadUser();

},[])

**loadUser**

const loadUser = async () => {

const token = sessionStorage.getItem('token');

if(!token){

dispatch({

type: ERROR

})

}

setAuthToken(token);

try {

const res = await axios('/api/auth');

dispatch({

type: USER\_LOADED,

payload: res.data.data

})

} catch (err) {

console.error(err);

}

}

1. **What are the benefits of new JSX transform?**

There are three major benefits of new JSX transform,

* 1. It is possible to use JSX without importing React packages
  2. The compiled output might improve the bundle size in a small amount
  3. The future improvements provides the flexibility to reduce the number of concepts to learn React.

1. **How does new JSX transform different from old transform?**

The new JSX transform doesn’t require React to be in scope. i.e, You don't need to import React package for simple scenarios.

Let's take an example to look at the main differences between the old and the new transform,

**Old Transform:**

import React from 'react';

function App() {

return <h1>Good morning!!</h1>;

}

Now JSX transform convert the above code into regular JavaScript as below,

import React from 'react';

function App() {

return React.createElement('h1', null, 'Good morning!!');

}

**New Transform:**

The new JSX transform doesn't require any React imports

function App() {

return <h1>Good morning!!</h1>;

}

Under the hood JSX transform compiles to below code

import {jsx as \_jsx} from 'react/jsx-runtime';

function App() {

return \_jsx('h1', { children: 'Good morning!!' });

}

**Note:** You still need to import React to use Hooks.

1. **How do you get redux scaffolding using create-react-app?**

Redux team has provided official redux+js or redux+typescript templates for create-react-app project. The generated project setup includes,

* 1. Redux Toolkit and React-Redux dependencies
  2. Create and configure Redux store
  3. React-Redux <Provider> passing the store to React components
  4. Small "counter" example to demo how to add redux logic and React-Redux hooks API to interact with the store from components

The below commands need to be executed along with template option as below,

* 1. **Javascript template:**

npx create-react-app my-app --template redux

* 1. **Typescript template:**

npx create-react-app my-app --template redux-typescript

1. **What are React Server components?**

React Server Component is a way to write React component that gets rendered in the server-side with the purpose of improving React app performance. These components allow us to load components from the backend.

**Note:** React Server Components is still under development and not recommended for production yet.

1. **What is prop drilling?**

Prop Drilling is the process by which you pass data from one component of the React Component tree to another by going through other components that do not need the data but only help in passing it around.