Q1. What is Redux?

The basic idea of Redux is that the entire application state is kept in a single store. The store is simply a javascript object. The only way to change the state is by firing actions from your application and then writing reducers for these actions that modify the state. The entire state transition is kept inside reducers and should not have any side-effects. Redux is based on the idea that there should be only a single source of truth for your application state, be it UI state like which tab is active or Data state like the user profile details.
All of these data is retained by redux in a closure that redux calls a store . It also provides us a recipe of creating the said store, namely createStore(x). The createStore function accepts another function, x as an argument. The passed in function is responsible for returning the state of the application at that point in time, which is then persisted in the store
This store can only be updated by dispatching an action. Our App dispatches an action, it is passed into reducer; the reducer returns a fresh instance of the state; the store notifies our App and it can begin it's re render as required.

Q2. What is Redux Thunk used for?

Redux thunk is middleware that allows us to write action creators that return a function instead of an action. The thunk can then be used to delay the dispatch of an action if a certain condition is met. This allows us to handle the asyncronous dispatching of actions. The inner function receives the store methods dispatch and getState as parameters. To enable Redux Thunk, we need to use applyMiddleware()

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './reducers/index';

// Note: this API requires redux@>=3.1.0
const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);
```

Q3. Explain the components of Redux.

Redux is composed of the following components: Action — Actions are payloads of information that send data from our application to our store. They are the only source of information for the store. We send them to the store using store.dispatch(). Primarly, they are just an object describes what happened in our app. Reducer — Reducers specify how the application's state changes in response to actions sent to the store. Remember that actions only describe what happened, but don't describe how the application's state changes. So this place determines how state will change to an action. Store — The Store is the object that brings Action and Reducer together. The store has the following responsibilities: Holds application state; Allows access to state via getState(); Allows state to be updated via dispatch(action); Registers listeners via subscribe(listener); Handles unregistering of listeners via the function returned by subscribe(listener).
It's important to note that we'll only have a single store in a Redux application. When we want to split your data handling logic, we'll use reducer composition instead of many stores.

Q4: What do you understand by "Single source of truth"?

Redux uses 'Store' for storing the application's entire state at one place. So all the component's state are stored in the Store and they receive updates from the Store itself. The single state tree makes it easier to keep track of changes over time and debug or inspect the application.

Q5: List down the components of Redux.

- Action – It's an object that describes what happened.
- Reducer – It is a place to determine how the state will change.
- Store – State/ Object tree of the entire application is saved in the Store.
- View – Simply displays the data provided by the Store

Q6: Explain the role of Reducer.

Reducers are pure functions which specify how the application's state changes in response to an ACTION. Reducers work by taking in the previous state and action, and then it returns a new state. It determines what sort of update needs to be done based on the type of action, and then returns new values. It returns the previous state as it is if no work needs to be done.

Q7: What is the significance of Store in Redux?

A store is a JavaScript object which can hold the application's state and provide a few helper methods to access the state, dispatch actions and register listeners. The entire state/ object tree of an application is saved in a single store. As a result of this, Redux is very simple and predictable. We can pass middleware to the store to handle processing of data as well as to keep a log of various actions that change the state of stores. All the actions return a new state via reducers.

Q8: What are the advantages of Redux?

Predictability of outcome: Since there is always one source of truth, i.e. the store, there is no confusion about how to sync the current state with actions and other parts of the application.
Maintainability: It is simple to maintain a strict structure and predictable outcome.
Server-side rendering: To the client side, You just need to pass the store created on the server. This is helpful for initial render and provides a high-quality user experience as it optimizes the application performance.
Developer tools: From actions to state changes, developers can track everything going on in the application in real time.
Community and ecosystem: Redux has a huge community behind it which makes it even more captivating to use. A large community of talented individuals contribute to the betterment of the library and develop various applications with it.
Ease of testing: Redux's code is mostly functions which are small, pure and isolated. This makes the code testable and independent.
Organization: Redux is precise about how code should be organized, this makes the code more consistent and easier when a team works with it.

Q9: Benefits of Redux?

Maintainability: maintenance of Redux becomes easier due to strict code structure and organisation.
Organization: code organisation is very strict hence the stability of the code is high which intern increases the work to be much easier.
Server rendering: This is useful, particularly to the preliminary render, which keeps up a better user experience or search engine optimization. The server-side created stores are forwarded to the client side.
Developer tools: It is Highly traceable so changes in position and changes in the application all such instances make the developers have a real-time experience.
Ease of testing: The first rule of writing testable code is to write small functions that do only one thing and that are independent. Redux's code is made of functions that used to be: small, pure and isolated.

Q10: What is the typical flow of data in a React + Redux app?

Call-back from UI component dispatches an action with a payload, these dispatched actions are intercepted and received by the reducers. this interception will generate a new application state. from here the actions will be propagated down through a hierarchy of components from Redux store. The below diagram depicts the entity structure of a redux+react setup.

Q11 Explain Reducers in Redux?

The initial state and action are received by the reducers. Based on the action type, it returns a new state for the store. The state maintained by reducers are immutable. The below-given reducer it holds the current state and action as an argument for it and then returns the next

Q12 Redux workflow features?

Reset: Allow to reset the state of the store
Revert: Roll back to the last committed state
Sweep: All disabled actions that you might have fired by mistake will be removed
Commit: makes the current state the initial state

Q13 Explain action's in Redux?

Actions in Redux are functions which return an action object. The action type and the action data are packed in the action object. which also allows a donor to be added to the system. Actions send data between the store and application. All information's retrieved by the store are produced by the actions.
nternal Actions are built on top of Javascript objects and associate a type property to it.

Q14: What is "store" in Redux?

Store is the object that holds the application state and provides a few helper methods to access the state, dispatch actions and register listeners. The entire state is represented by a single store. Any action returns a new state via reducers. That makes Redux very simple and predictable.

Q15: How would you disable the store redux, so it does not accept any changes to state?.

One way to do it is by setting an exit flag in the root state reducer, so it is set to true it just let the state pass unmodified.

Q16: How to access redux store outside a react component?

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

ReduxCurd
reduxQuestions
mernReduxCurd
jsonPlaceholder

Curd
curdActions.js

```
export const createItem =(params)=>({
type:'CREATE_ITEMS',
```

data:params
0

rootReducers.js

```
const initialState={
items:[
{id:1, title:JavaScript', date:new Date()},
{id:2, title:'React', date:new Date()},
{id:3, title:'Redux', date:new Date()},
{id:4, title:'MERN', date:new Date()},
] }

export default (state = initialState, action)=>{
switch(action.type){
case 'LIST_ITEM':
return {
..-State,
item:state.root
}
case ‘CREATE_ITEMS':{
let c_Items = Object.assign([], state.items.map(d
=> Object.assign({},d)});
c_Items.push(action.data)
return {...state,items:c_Items}
}
default:
return {...state }
} }
```

index.js

```
mport { combineReducers } from ‘redux’;
import root from './rootReducers'

export default combineReducers({
root:root
}
```

curd.jsx

```
import React, { Component } from ‘react’;
import { connect } from ‘react-redux';
import { createItem } from '../actions/cureActions';

class Curd extends Component{
state={
isUpdate:false
}
onSubmit = (e) => {
e.preventDefault()
let obj ={
id: this.id.value,
title:this.title.value
}
this.state.isUpdate ? this.props.update(obj) : this.props.create(obj)
this.setState({
isUpdate:false
1)
}
render(){
return(
'<'div>
'<'form onSubmit={this.onSubmit} >
'<'input type="text" placeholder="Name" ref={id=>this.id=id} autoFocus/>
'<'input type="number" placeholder="age" ref={title=>this.title=title} autoFocus/>
'<'button>{this.state.isUpdate ? ‘update’ : ‘Add'}'</'button>
'<'/form>
{this.props.items.map(ids=>(
'<'div>
'<'p>name: {ids.id}'<'/p>
```

'<'p>age: {ids.title}'<'/p>

'<'/div>
)
}
}

const mapStateToProps = (state) => ({
items:state.root.items
)

const mapDispatchToProps = (dispatch) => ({
create : (params) => dispatch(createItem(params)),
)

export default connect(mapStateToProps,mapDispatchToProps)(Curd);

MernReduxCurd
reduxQuestions
ReduxCurd
jsonPlaceholder
JsonPlaceholder
reduxQuestions
ReduxCurd
mernReduxCurd

MernReduxCurd

Fatch api
postActions.js

import { FETCH_POSTS, NEW_POST, UPDATE } from './types.js'; export const fetchPosts = () => dispatch => { console.log('fetching'); fetch('https://jsonplaceholder.typicode.com/posts', { method: 'GET', headers: { 'content-type': 'application/json' }, }) .then(res => res.json()) .then(posts => dispatch({ type: FETCH_POSTS, payload: posts })); }; export const createPost = postData => dispatch => { console.log('action called'); fetch('https://jsonplaceholder.typicode.com/posts', { method: 'POST', headers: { 'content-type': 'application/json' }, body: JSON.stringify(postData) }) .then(res => res.json()) .then(post => dispatch({ type: NEW_POST, payload: post })); }; export const updatePost = updateData => dispatch => { console.log('update data'); fetch('https://jsonplaceholder.typicode.com/posts/1', { method: 'PUT', headers: { 'content-type': 'application/json' }, body: JSON.stringify({ id: 1, title: 'foo', body: 'bar', userId: 1 }), }) .then(res => res.json()) .then(json => console.log(json)) }

types.js

export const FETCH_POSTS = 'FETCH_POSTS'; export const NEW_POST = 'NEW_POST'; export const UPDATE = 'UPDATE';

postReducers.js

import { FETCH_POSTS, NEW_POST, UPDATE } from '../actions/types.js'; const initialState = { items: [], item: {} }; export default function(state = initialState, action) { switch (action.type) { case FETCH_POSTS: return { ...state, items: action.payload }; case NEW_POST: return { ...state, item: action.payload }; case UPDATE: return { ...state, item: action.payload }; default: return state; } }

index.js

import { combineReducers } from 'redux'; import postReducer from './postReducer.js'; export default combineReducers({ posts: postReducer, });

store.js

import { createStore, applyMiddleware, compose } from 'redux'; import thunk from 'redux-thunk'; import rootReducer from './reducers'; const initialState = {}; const middleware = [thunk]; const store = createStore(rootReducer, initialState, compose(applyMiddleware(...middleware), window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__())); export default store;

postForm.jsx

import React, { Component } from 'react'; import { connect } from 'react-redux'; import { createPost } from '../actions/postActions.js'; class PostForm extends Component { constructor(props) { super(props); this.state = { title: '', body: '' }; this.onChange = this.onChange.bind(this); this.onSubmit = this.onSubmit.bind(this); } onChange(e) { this.setState({ [e.target.name]: e.target.value }); } onSubmit(e) { e.preventDefault(); const post = { title: this.state.title, body: this.state.body }; this.props.createPost(post); } render() { return (

Add Posts

Title:
{this.state.title}

Body:

```
          </div>
          <br />
```

posts.jsx

import React, { Component } from 'react'; import { connect } from 'react-redux'; import { fetchPosts } from '../actions/postActions.js'; class Posts extends Component { componentWillMount() { this.props.fetchPosts(); } componentWillReceiveProps(nextProps) { if (nextProps.newPost) { this.props.posts.unshift(nextProps.newPost); } } render() { const postItems = this.props.posts.map((post) => (

{post.title}

{post.body}

)); return (

Posts

{postItems}
); } } const mapStateToProps = state => ({ posts: state.posts.items, newPost: state.posts.item }); export default connect(mapStateToProps, { fetchPosts })(Posts);

indexComponents.jsx

import React, { Component } from 'react'; import { Provider } from 'react-redux'; import Posts from './components/Posts.js'; import PostForm from './components/PostForm.js'; import store from './store.js'; class App extends Component { componentWillMount() { fetch(''); } render() { return (

); } } export default App;