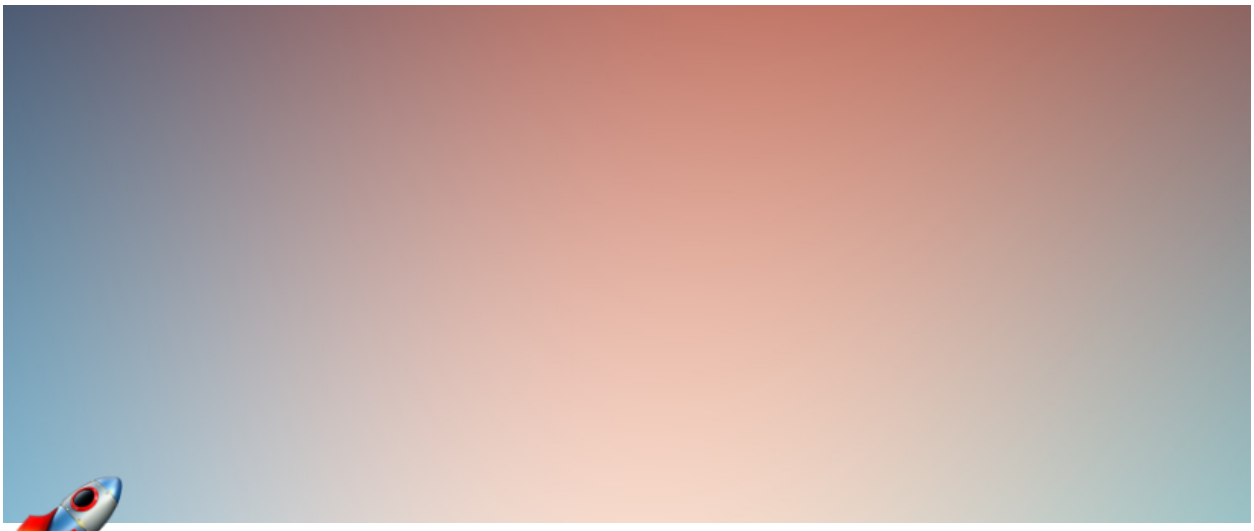

Episode-12 | Let’s Build Our Store

💡 Please make sure to follow along with the whole "**Namaste React**" series, starting from Episode-1 and continuing through each subsequent episode. The notes are designed to provide detailed explanations of each concept along with examples to ensure thorough understanding. Each episode builds upon the knowledge gained from the previous ones, so starting from the beginning will give you a comprehensive understanding of React dvelopment.

💡 I've got a quick tip for you. To get the most out of these notes, it's a good idea to watch **Episode-12** first. Understanding what "**Akshay**" shares in the video will make these notes way easier to understand.

Q) `useContext` vs `Redux`

> `useContext` and `Redux` are both tools used for state management in React applications, but they serve different purposes and have different use cases. Let's explore the key differences between useContext and Redux:

💡 **useContext:**

`Scope`: `useContext` is part of the React core and is used for managing state within the component tree. It provides a way to access the value of a context directly within a component and its descendants. It's typically used for smaller-scale state management needs within a component or a small section of the application.

`Complexity`: useContext is simpler and more lightweight compared to Redux. It's a part of the React library and doesn't introduce additional concepts or boilerplate code.

`Component Coupling`: State managed with useContext is local to the component or a subtree of components where the context is provided. This can lead to more isolated and less globally shared state.

`Integration`: It's seamlessly integrated into React and works well with the component lifecycle. You can create and consume contexts within functional components using the useContext hook.

💡 **Redux:**

`Scope`: Redux is a state management library that provides a global state container for the entire application. It allows you to manage the application state in a predictable and centralized manner.

`Complexity`: Redux introduces a set of concepts, such as actions, reducers, and a store. This can make it more complex compared to using useContext for local state management. However, it becomes valuable in larger and more complex applications.

`Component Coupling`: State managed with Redux is global, which means any component can connect to and access the state. This can be advantageous for sharing state across different parts of the application.

`Integration`: Redux needs to be integrated separately into a React application. You need to create actions, reducers, and a store. Components interact with the global state using the connect function or hooks like useSelector and useDispatch.

> 💡 **Use Cases:**

`Use useContext When`: We have smaller-scale state management needs within a component or a local subtree. We want a lightweight solution without introducing additional complexity. Our state doesn't need to be shared extensively across different parts of the application.

`Use Redux When`: We have a complex application with a large state that needs to be shared across many components. We want a predictable state management pattern with a unidirectional data flow. We need middleware for advanced features like asynchronous actions.

Choose `useContext for simpler and local state management within components or small sections of our application`. Choose `Redux for more complex applications where we need a global state that can be easily shared across different components`.

Q) Advantages of using `Redux Toolkit over Redux`?

`Redux Toolkit` is a set of utility functions and abstractions that simplifies and streamlines the process of working with Redux. It is designed to address some of the common pain points and boilerplate associated with using plain Redux. Here are some advantages of using Redux Toolkit over plain Redux:

`Less Boilerplate Code`: Redux Toolkit helps us write less code. It provides shortcuts that save us from typing a lot of repetitive and verbose code, making our Redux logic cleaner and more concise.

`Easier Async Operations`: If our app deals with things like fetching data from a server, Redux Toolkit makes it simpler. It has a tool called createAsyncThunk that handles async actions in a way that's easy to understand and use.

`Simpler Store Setup`: Setting up your Redux store is easier with Redux Toolkit. It has a function called configureStore that simplifies the process, and it comes with sensible defaults, so you don't have to configure everything from scratch.

`Built-in DevTools Support`: If you use Redux DevTools for debugging, Redux Toolkit has built-in support. Enabling it is as easy as adding one line of code when setting up your store.

`Encourages Best Practices`: Redux Toolkit is recommended by the official Redux documentation. It encourages you to follow best practices in Redux development, making sure your code is more maintainable and aligns with industry standards.

`Handles Immutability for You`: Working with immutable data (making sure you don't accidentally change your data) is usually a bit tricky. Redux Toolkit uses a library called Immer to handle this behind the scenes, so you can write more straightforward and readable code.

`Backward Compatibility`: If you already have a Redux app, you can slowly transition to Redux Toolkit without rewriting everything. It's designed to be compatible with your existing Redux code.

`Faster Development`: With Redux Toolkit, you can get things done more quickly. You spend less time setting up and configuring Redux and more time focusing on building features for your app.

In simple terms, Redux Toolkit is like a set of tools that makes working with Redux easier. It simplifies common tasks, reduces the amount of code you need to write, and encourages good coding practices. If you're starting a new project or thinking about improving an existing one, Redux Toolkit can save you time and make your life as a developer more enjoyable.

Q) Explain `Dispatcher` ?

> In Redux, a `dispatcher` is not a standalone concept; instead, it's a term often used to refer to a function called dispatch. The dispatch function is a key part of the Redux store, and it plays a crucial role in the Redux data flow.

💡 Here's a breakdown of the dispatch function and its role in Redux:

1 `Dispatch Function`: The dispatch function is provided by the Redux store. We use it to send actions to the store. An action is a plain JavaScript object that describes what should change in the application's state.

2 `Usage`: When we want to update the state in our Redux store, we create an action and dispatch it using the dispatch function.

```
const myAction = { type: 'INCREMENT' };
store.dispatch(myAction);
```

- Here, the `INCREMENT` action is an example. The dispatch function is responsible for sending this action to the Redux store.

3 `Middleware`: The dispatch function is also a crucial point in the Redux middleware chain. Middleware can intercept actions before they reach the reducer or modify actions on the way out. Middleware functions receive the dispatch function, allowing them to either pass the action along or stop it.

4 `Redux Store`: The dispatch function is a core method provided by the Redux store. When an action is dispatched, the store passes the action through its reducer, which is a function that specifies how the state should change in response to the action.

5 `Asynchronous Actions`: Redux supports asynchronous actions using middleware like redux-thunk or redux-saga. The dispatch function allows you to handle asynchronous operations by dispatching actions inside functions (thunks) and handling those actions asynchronously.

📌 Here's an example of how you might use dispatch in a React component:

```
import { useDispatch } from 'react-redux';

const MyComponent = () => {
```

```
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    // Dispatching an action to increment the count
    dispatch({ type: 'INCREMENT' });
  };

  return (
    <button onClick={handleButtonClick}>
      Increment Count
    </button>
  );
};
```

In this example, the `useDispatch` hook from react-redux gives us access to the dispatch function, which we then use to send an action to the Redux store when the button is clicked. This action will be processed by the reducer, updating the state accordingly.

Q) Explain `Reducer` ?

> **In Redux Toolkit, the `createSlice function is commonly used to create reducers`. It simplifies the process of defining actions and the corresponding reducer logic, reducing boilerplate code. Let's break down the key concepts related to creating reducers with createSlice in Redux Toolkit :**

`Creating a Slice`: Instead of creating a standalone reducer function, you use createSlice to define a "slice" of your Redux store. A slice includes actions, a reducer, and the initial state.

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
```

```
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});

// Extracting actions and reducer from the slice
export const { increment, decrement } = counterSlice.actions;
export default counterSlice.reducer;
```

2 `Automatically Generated Action Creators`: `createSlice` automatically generates action creators for each reducer function. In the example above, increment and decrement are automatically created and exported for use in your components.

3 `Immutability with immer`: Redux Toolkit uses the immer library internally to handle immutability. This means you can write reducer logic that appears to directly modify the state, but immer ensures it produces a new state without mutating the original.

```
reducers: {
  increment: (state) => {
    state.value += 1;  // Immer takes care of creating a new
state
  },
},
```

4 `Reducer Function`: The createSlice function returns an object that includes a reducer property. This reducer is a function that you can use in your store's configuration.

```
const rootReducer = combineReducers({
  counter: counterSlice.reducer,
  // ... other reducers
});
```

5 `Initial State`: The initialState property in createSlice defines the initial state of your slice. This is the starting point for your state before any actions are dispatched.

6 `Reducer Logic`: The logic inside each reducer function specifies how the state should change in response to the associated action. In the example, the increment and decrement reducers modify the value property of the state.

7 `Simplifying Reducer Composition`: With createSlice, we can easily compose reducers using combineReducers or by directly adding the slice's reducer to the root reducer. This simplifies the overall reducer composition in our application.

Using createSlice in Redux Toolkit streamlines the process of defining reducers, actions, and initial states, making your Redux code more concise and readable. It encourages best practices, such as immutability and simplicity, while reducing the boilerplate traditionally associated with Redux.

Q) Explain `Slice` ?

> In Redux Toolkit, a `slice` is a collection of Redux-related code, including reducer logic and actions, that corresponds to a specific piece of the application state. Slices are created using the createSlice utility function provided by Redux Toolkit. The primary purpose of slices is to encapsulate the logic related to a specific part of the state, making the code more modular and easier to manage.

🛠️ Here's a breakdown of key concepts related to slices in Redux Toolkit:

Creating a Slice: The createSlice function takes an options object with the following properties:

1 `name (string)` : A string that identifies the slice. This is used as the prefix for the generated action types.

```
import { createSlice } from '@reduxjs/toolkit';

const mySlice = createSlice({
  name: 'mySlice',
  initialState: { /* ... */ },
  reducers: {
    // ...reducers
  },
});
```

2 `initialState (any)`: The initial state value for the slice. This is the starting point for your state before any actions are dispatched. 3 `reducers (object)`: An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  initialState: { /* ... */ },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
});
```

Output: The createSlice function returns an object with the following properties:

`name (string)`: The name of the slice. `reducer (function)`: The reducer function generated based on the provided reducers. This is the function you use in your store configuration. `actions (object)`: An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```

Using a Slice: Once we've created a slice, you can use its reducer and actions in your Redux store configuration and in your React components.

1 `Configuring the Store`: We can include the generated reducer in your store configuration.

```
import { configureStore } from '@reduxjs/toolkit';
import mySliceReducer from './path/to/mySlice';

const store = configureStore({
  reducer: {
    mySlice: mySliceReducer,
    // ...other reducers
  },
});
```

2 `Dispatching Actions`: In our React components, we can use the generated action creators to dispatch actions.

```
import { useDispatch } from 'react-redux';
import { increment } from './path/to/mySlice';

const MyComponent = () => {
  const dispatch = useDispatch();

  const handleIncrement = () => {
    dispatch(increment());
  };
```

```
  // ... rest of the component logic
};
```

Using `slices in Redux Toolkit promotes a modular and organized approach to state management`. Each slice encapsulates the logic related to a specific part of the state, making it easier to understand, maintain, and scale your Redux code.

Q) Explain `Selector` ?

> In Redux Toolkit, a `selector` is a function that extracts specific pieces of data from the Redux store. It allows you to compute derived data from the store state and efficiently access specific parts of the state tree. Selectors play a crucial role in managing the state in a clean and efficient way.

Redux Toolkit provides the createSlice and createAsyncThunk utilities along with the createSelector function from the reselect library to help manage selectors easily.

📣 Here's an explanation of how selectors work in Redux Toolkit:

1 `Defining Selectors with createSlice`: When we create a slice using createSlice, we can include selectors in the extraReducers field. These selectors can compute and return specific pieces of data from the state.

```
import { createSlice } from '@reduxjs/toolkit';

const mySlice = createSlice({
  name: 'mySlice',
  initialState: { data: [] },
  reducers: {
    // ...reducers
```

```
  },
  extraReducers: (builder) => {
    builder
      .addCase(otherSliceAction, (state, action) => {
        // logic for handling other slice's action
      })
      .addDefaultCase((state, action) => {
        // default logic for handling actions not handled by
this slice
      });
  },
  selectors: (state) => ({
    // selector functions here
    selectData: () => state.data,
    selectFilteredData: (filter) => state.data.filter(item =>
item.includes(filter)),
  }),
});

export const { selectData, selectFilteredData } = mySlice.sel
ectors;
```

2 `Using Reselect with createSlice`: If we need more advanced memoization and composition of selectors, you can use the createSlice function along with the reselect library.

```
import { createSlice, createSelector } from '@reduxjs/toolki
t';

const mySlice = createSlice({
  // ... other options
  selectors: {
    selectData: (state) => state.data,
    selectFilteredData: createSelector(
      (state) => state.data,
      (_, filter) => filter,
```

```
      (data, filter) => data.filter(item => item.includes(fil
ter))
    ),
  },
});

export const { selectData, selectFilteredData } = mySlice.sel
ectors;
```

3 `Using Selectors in Components`: Once we've defined selectors, we can use them in your React components using the useSelector hook from the react-redux library. This hook allows you to efficiently extract and subscribe to parts of the Redux store.

```
import { useSelector } from 'react-redux';
import { selectData, selectFilteredData } from './mySlice';

const MyComponent = () => {
  const data = useSelector(selectData);
  const filteredData = useSelector(state => selectFilteredDat
a(state, 'someFilter'));

  // ... rest of the component logic
};
```

Selectors help keep your state management logic clean and efficient by allowing us to centralize the computation of derived data from the Redux store. They contribute to better organization, improved performance, and easier maintenance of our Redux code.

Q) Explain `createSlice` and the configuration it takes?

> `createSlice` is a utility function provided by Redux Toolkit that simplifies the `process of creating Redux slices`. A `Redux slice is a piece of the Redux store that includes a set of actions, a reducer, and an initial state`. The createSlice function helps reduce boilerplate code associated with defining actions and the reducer for a specific slice of your Redux store.

Here's an explanation of the configuration options that createSlice takes:

📣 **Syntax:**

```
createSlice(options)
```

📣 **Configuration Options:**

1 `name (string)`: A string that identifies the slice. This is used as the prefix for the generated action types.

```
const mySlice = createSlice({
  name: 'mySlice',
  // ... other options
});
```

2 `initialState (any)`: The initial state value for the slice. This is the starting point for your state before any actions are dispatched.

```
const mySlice = createSlice({
  initialState: { value: 0 },
  // ... other options
});
```

3 `reducers (object)`: An object where each key-value pair represents a reducer function. The keys are the names of the actions, and the values are the corresponding reducer logic.

```
const mySlice = createSlice({
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
      state.value -= 1;
    },
  },
  // ... other options
});
```

4 `extraReducers (builder callback)`: A callback function that allows you to define additional reducers outside of the reducers field. It is called with a builder object that provides methods for adding reducers based on other action types.

```
const mySlice = createSlice({
  extraReducers: (builder) => {
    builder
      .addCase(otherSliceAction, (state, action) => {
        // logic for handling other slice's action
      })
      .addDefaultCase((state, action) => {
        // default logic for handling actions not handled by
this slice
      });
  },
  // ... other options
});
```

5 `slice (string)`: An optional string that specifies a slice of the state to be used with the createAsyncThunk utility. This is useful when working with asynchronous

actions.

```
const mySlice = createSlice({
  slice: 'myAsyncSlice',
  // ... other options
});
```

6 `extraReducers (object)`: An alternative way to define extra reducers using an object directly. Each key represents an action type, and the value is the corresponding reducer function.

```
const mySlice = createSlice({
  extraReducers: {
    [otherSliceAction.type]: (state, action) => {
      // logic for handling other slice's action
    },
    // ... additional action types
  },
  // ... other options
});
```

 **Output :**

The createSlice function returns an object with the following properties:

`name (string)`: The name of the slice. `reducer (function)`: The reducer function generated based on the provided reducers and extraReducers. This is the function you use in your store configuration. `actions (object)`: An object containing the action creators for each defined reducer. These action creators can be directly used to dispatch actions.

```
const { increment, decrement } = mySlice.actions;
```

These are the main configuration options for createSlice in Redux Toolkit. It provides a convenient way to define actions, reducers, and initial states for slices of our Redux store, reducing the amount of boilerplate code and promoting best practices.