1. **Your React Components:** These are the UI elements of your application that need to interact with the state managed by Redux.

2. **Actions:** Actions are plain JavaScript objects that describe events or changes in your application. They are dispatched by components to notify the store that something has happened.

3. **Reducers:** Reducers are functions that specify how the state should change in response to actions. They take the current state and an action, and return a new state.

4. **Store:** The store is a centralized container that holds the entire state of your application. It's responsible for managing state updates and notifying components when the state changes.

5. **Your App:** This represents the complete application, including the React components, actions, reducers, and the Redux store.

## THE STORE

**1.Central Data Container:** The store in Redux is like a centralized data container for your application. It holds the entire application's state, which includes all the data that your app needs to function.

**2.Single Source of Truth:** Redux promotes the idea of having a single source of truth for your application's state. This means that all the data your app uses is stored in a single object within the store, making it easier to manage and maintain.

**3.State Management:** The store manages the state of your application. Instead of scattering state across different components, Redux gathers it in one place. This makes it simpler to track changes and manage updates.

**4.Immutable State:** In Redux, the state stored in the store is immutable, which means you can't directly modify it. Instead, you create new state objects by dispatching actions. This helps to maintain a clear history of changes and enables efficient change detection.

**5.Actions and Reducers:** To update the state in the store, you dispatch actions. Actions are plain JavaScript objects that describe what happened in your app. Reducers are functions that specify how the state should change based on the actions. They take the current state and an action, and return a new state. The store uses reducers to update its state.

## THE REDUCER

**1.State Modification Logic:** A reducer in Redux is a pure JavaScript function responsible for defining how the application's state changes in response to actions. It takes the current state and an action as input and returns a new state.

**2.Predictable State Changes:** Reducers ensure that state changes are predictable and well-defined. Given the same input (current state and action), a reducer will always produce the same output (new state), which makes debugging and testing easier.

**3.Immutable State:** Reducers follow the principle of immutability, meaning they never modify the existing state. Instead, they create a new state object by copying the current state and applying changes. This prevents unintended side effects.

**4.Switching Based on Action Type:** Reducers use a switch statement or if-else chains to determine how the state should change based on the type of action. Each case corresponds to a different action type and defines how the state should be updated for that action.

**5.Combine Reducers:** As applications grow, they might have multiple pieces of state to manage. Redux allows combining multiple reducers into a single root reducer using the **combineReducers** function. Each reducer handles a specific slice of the application state, which keeps code modular and maintainable.

## THE ACTIONS

1. **Describing Events:** An action in Redux is a plain JavaScript object that describes an event or an intention to change the state of your application. It represents what happened in your app, such as a button click, user input, or data retrieval.

2. **Type Property:** Actions have a required **type** property that indicates the type of action being performed. This is usually defined as a string constant, like **'INCREMENT'** or **'FETCH_DATA'**.

3. **Payload:** Sometimes, actions also carry additional information, called the payload. The payload can be any data relevant to the action, such as new values, user input, or fetched data. This information helps reducers determine how the state should change.

4. **Dispatching Actions:** To initiate a state change, an action is dispatched using the **dispatch** function provided by Redux. Components or other parts of your app trigger actions to notify the store about events or user interactions.

5. **Connecting Actions to Reducers:** Reducers use the type of an action to decide how the state should be modified. When an action is dispatched, Redux routes the action to the appropriate reducer based on its type. The reducer processes the action and produces a new state.

1. **Setup Redux:**
   - Install the required packages: Use npm or yarn to install `redux` and `react-redux` packages in your project.

2. **Create Actions:**
   - Define action types: Create constants (strings) to represent different action types.
   - Create action creators: Write functions that return action objects with a `type` and optional `payload` property.

3. **Create Reducers:**
   - Define initial state: Define the initial state for your application.
   - Write reducer functions: Create pure functions that take the current state and an action as parameters, and return a new state based on the action type.

4. **Combine Reducers:**
   - If needed, use the `combineReducers` function from Redux to combine multiple reducer functions into a single root reducer.

5. **Create Store:**
   - Create the Redux store: Use the `createStore` function from Redux, passing in your root reducer and optionally the initial state.

6. **Integrate with React Components:**
   - Wrap your app with the `Provider` component from `react-redux` to make the Redux store accessible to all components.

7. **Connect Components:**
   - Use the `connect` function from `react-redux` to connect specific components to the Redux store.
   - Map state and dispatch to props: Define which parts of the state each component needs and which actions it will dispatch.

8. **Dispatch Actions:**
   - Dispatch actions: In your components, call the action creators and dispatch the actions to notify the store about events or changes.

9. **Access State:**
   - Access state: Use the mapped state properties from the connected components to access the state from the store.

10. **Update Components:**
   - Components re-render: When the state changes due to dispatched actions, connected components automatically re-render with the updated data.

**1.Reducer Boilerplate Reduction:** **createSlice** is a function provided by Redux Toolkit that significantly reduces the boilerplate code required for defining Redux state, actions, and reducers.

**2.All-in-One Definition:** When you use **createSlice**, you define the initial state, action types, action creators, and reducer logic all in one place. This makes your code cleaner and easier to maintain.

**3.Automatic Action Creators: createSlice** automatically generates action creators based on the action types you define in the **reducers** object. You don't need to manually write separate action creators for each action.

**4.Immer Integration:** Under the hood, **createSlice** uses the Immer library, which simplifies the process of updating immutable state. You can write reducer logic that looks like mutable updates, but it produces new immutable state behind the scenes.

**5.Exporting Reducer and Action Creators:** After defining a slice using **createSlice**, you can export the generated action creators and the reducer function directly. This makes it easy to integrate the slice into your Redux setup.

To manage the state of a button in one component and then accessing that state in another component.

**1.Setup Redux Toolkit:**
First, make sure you have Redux Toolkit installed in your project.
npm install @reduxjs/toolkit react-redux

**2.Create a Slice with createSlice:**

Create a new file named buttonSlice.js to define your button's state and actions.

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

const buttonSlice = createSlice({
  name: 'button',
  initialState: false,
  reducers: {
    toggleButton: (state) => !state,
  },
});

export const { toggleButton } = buttonSlice.actions;
export default buttonSlice.reducer;
```

## 3.Configure the Store:

Set up your Redux store by combining the button reducer with other reducers if needed.

```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import buttonReducer from './buttonSlice';

const store = configureStore({
  reducer: {
    button: buttonReducer,
    // Other reducers if any
  },
});

export default store;
```

## 4.Provide the Store:

Wrap your main application component with the Provider from react-redux to make the store available throughout your app.

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);
```

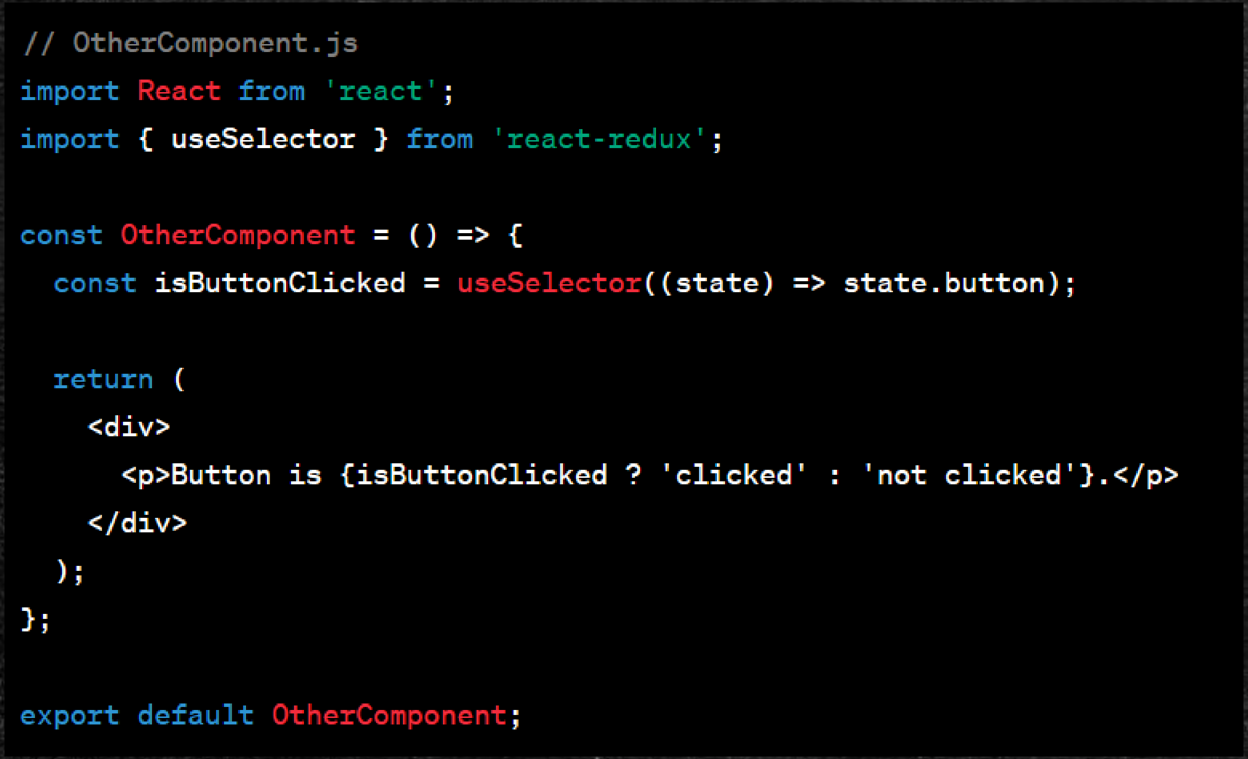
## 5. Use Redux State in ButtonComponent:

In your ButtonComponent, you'll use Redux to manage the button state instead of useState.

```
// ButtonComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { toggleButton } from './buttonSlice';

const ButtonComponent = () => {
  const isButtonClicked = useSelector((state) => state.button);
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    dispatch(toggleButton());
  };

  return (
    <div>
      <button onClick={handleButtonClick}>
        {isButtonClicked ? 'Clicked' : 'Not Clicked'}
      </button>
    </div>
  );
};

export default ButtonComponent;
```

## 6.Access Redux State in OtherComponent:

In your OtherComponent, you can access the button state from Redux



```
// OtherComponent.js
import React from 'react';
import { useSelector } from 'react-redux';

const OtherComponent = () => {
  const isButtonClicked = useSelector((state) => state.button);

  return (
    <div>
      <p>Button is {isButtonClicked ? 'clicked' : 'not clicked'}.</p>
    </div>
  );
};

export default OtherComponent;
```

## 7.Using the Components:

You can now use both ButtonComponent and OtherComponent in your main application.

```
// App.js
import React from 'react';
import ButtonComponent from './ButtonComponent';
import OtherComponent from './OtherComponent';

const App = () => {
  return (
    <div>
      <ButtonComponent />
      <OtherComponent />
    </div>
  );
};

export default App;
```

## How to use action.payload to pass data when dispatching an action

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

const buttonSlice = createSlice({
  name: 'button',
  initialState: { clicked: false, color: 'red' },
  reducers: {
    toggleButton: (state, action) => {
      state.clicked = !state.clicked;
      state.color = action.payload;
    },
  },
});

export const { toggleButton } = buttonSlice.actions;
export default buttonSlice.reducer;
```

Update ButtonComponent.js:

Modify ButtonComponent.js to toggle the button's color using action.payload.

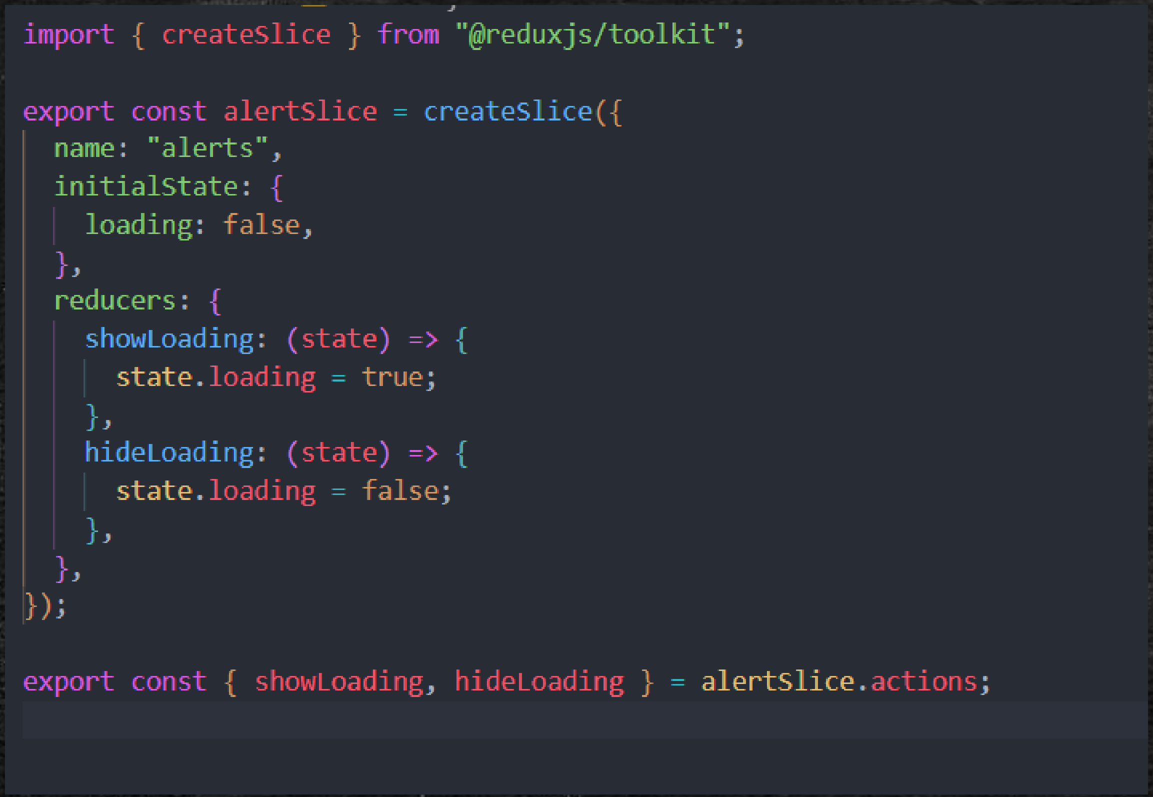
```
// ButtonComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { toggleButton } from './buttonSlice';

const ButtonComponent = () => {
  const { clicked, color } = useSelector((state) => state.button);
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    const newColor = color === 'red' ? 'blue' : 'red';
    dispatch(toggleButton(newColor));
  };

  return (
    <div>
      <button
        onClick={handleButtonClick}
        style={{ backgroundColor: color }}
      >
        {clicked ? 'Clicked' : 'Not Clicked'}
      </button>
    </div>
  );
};

export default ButtonComponent;
```

```javascript
// buttonSlice.js
import { createSlice } from '@reduxjs/toolkit';

const buttonSlice = createSlice({
  name: 'button',
  initialState: { clicked: 0 },
  reducers: {
    toggleButton: (state) => {
      state.clicked += 1;
    },
  },
});

export const { toggleButton } = buttonSlice.actions;
export default buttonSlice.reducer;
```

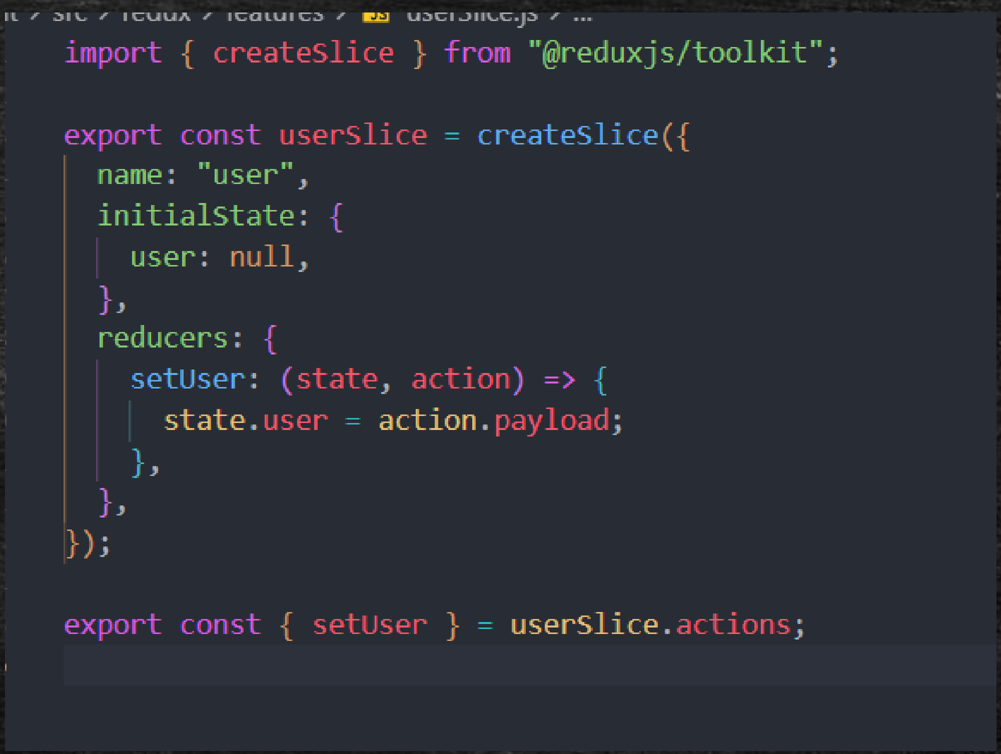
```javascript
// ButtonComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { toggleButton } from './buttonSlice';

const ButtonComponent = () => {
  const { clicked } = useSelector((state) => state.button);
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    dispatch(toggleButton());
  };

  return (
    <div>
      <button onClick={handleButtonClick}>
        Clicked: {clicked} times
      </button>
    </div>
  );
};

export default ButtonComponent;
```

```javascript
// buttonSlice.js
import { createSlice } from '@reduxjs/toolkit';

const buttonSlice = createSlice({
  name: 'button',
  initialState: { clicked: 0 },
  reducers: {
    incrementClicked: (state, action) => {
      state.clicked += action.payload;
    },
  },
});

export const { incrementClicked } = buttonSlice.actions;
export default buttonSlice.reducer;
```

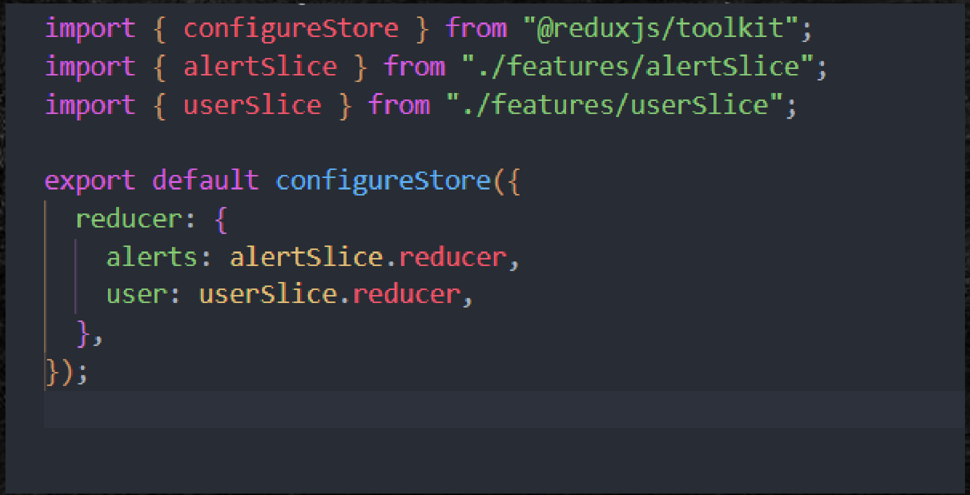
```javascript
// ButtonComponent.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { incrementClicked } from './buttonSlice';

const ButtonComponent = () => {
  const { clicked } = useSelector((state) => state.button);
  const dispatch = useDispatch();

  const handleButtonClick = () => {
    dispatch(incrementClicked(1)); // Increment count by 1
  };

  return (
    <div>
      <button onClick={handleButtonClick}>
        Clicked: {clicked} times
      </button>
    </div>
  );
};

export default ButtonComponent;
```



```javascript
import { createSlice } from "@reduxjs/toolkit";

export const alertSlice = createSlice({
  name: "alerts",
  initialState: {
    loading: false,
  },
  reducers: {
    showLoading: (state) => {
      state.loading = true;
    },
    hideLoading: (state) => {
      state.loading = false;
    },
  },
});

export const { showLoading, hideLoading } = alertSlice.actions;
```



```javascript
import { createSlice } from "@reduxjs/toolkit";

export const userSlice = createSlice({
  name: "user",
  initialState: {
    user: null,
  },
  reducers: {
    setUser: (state, action) => {
      state.user = action.payload;
    },
  },
});

export const { setUser } = userSlice.actions;
```



```javascript
import { configureStore } from "@reduxjs/toolkit";
import { alertSlice } from "./features/alertSlice";
import { userSlice } from "./features/userSlice";

export default configureStore({
  reducer: {
    alerts: alertSlice.reducer,
    user: userSlice.reducer,
  },
});
```