Redux is a predictable state container for JavaSc RTK includes utilities that help simplify many common use cases, including [store setup](https://redux-toolkit.js.org/api/configureStore), [creating reducers and writing immutable update logic](https://redux-toolkit.js.org/api/createreducer), and even [creating entire "slices" of state at once](https://redux-toolkit.js.org/api/createslice).ript apps.

The whole global state of your app is stored in an object tree inside a single store. The only way to change the state tree is to create an action, an object describing what happened, and dispatch it to the store. To specify how state gets updated in response to an action, you write pure reducer functions that calculate a new state based on the old state and the action.

This is a reducer - a function that takes a current state value and an  
 \* action object describing "what happened", and returns a new state value.  
 \* A reducer's function signature is: (state, action) => newState  
 \*  
 \* The Redux state should contain only plain JS objects, arrays, and primitives.  
 \* The root state value is usually an object. It's important that you should  
 \* not mutate the state object, but return a new object if the state changes.  
 \*  
 \* You can use any conditional logic you want in a reducer. In this example,  
 \* we use a switch statement, but it's not required.  
 \*/  
function counterReducer(state = { value: 0 }, action) {  
 switch (action.type) {  
 case 'counter/incremented':  
 return { value: state.value + 1 }  
 case 'counter/decremented':  
 return { value: state.value - 1 }  
 default:  
 return state  
 }  
}

Redux is a predictable state container for JavaScript apps. RTK includes utilities that help simplify many common use cases, including [store setup](https://redux-toolkit.js.org/api/configureStore), [creating reducers and writing immutable update logic](https://redux-toolkit.js.org/api/createreducer), and even [creating entire "slices" of state at once](https://redux-toolkit.js.org/api/createslice). The whole global state of your app is stored in an object tree inside a single store. The only way to change the state tree is to create an action, an object describing what happened, and dispatch it to the store. To specify how state gets updated in response to an action, you write pure reducer functions that calculate a new state based on the old state and the action}

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

**What Is "Redux"?**[**​**](https://redux.js.org/introduction/why-rtk-is-redux-today#what-is-redux)

The first thing to ask is, "what is Redux?"

Redux is really:

* A single store containing "global" state
* Dispatching plain object actions to the store when something happens in the app
* Pure reducer functions looking at those actions and returning immutably updated state

While it's not required, [your Redux code also normally includes](https://redux.js.org/tutorials/fundamentals/part-7-standard-patterns):

* Action creators that generate those action objects
* Middleware to enable side effects
* Thunk functions that contain sync or async logic with side effects
* Normalized state to enable looking up items by ID
* Memoized selector functions with the Reselect library for optimizing derived data
* The Redux DevTools Extension to view your action history and state changes
* TypeScript types for actions, state, and other functions

Additionally, Redux is normally used with the React-Redux library to let your React components talk to a Redux store.

**What Does the Redux Core Do?**[**​**](https://redux.js.org/introduction/why-rtk-is-redux-today#what-does-the-redux-core-do)

The Redux core is a very small and deliberately unopinionated library. It provides a few small API primitives:

* createStore to actually create a Redux store
* combineReducers to combine multiple slice reducers into a single larger reducer
* applyMiddleware to combine multiple middleware into a store enhancer
* compose to combine multiple store enhancers into a single store enhancer

**We specifically created Redux Toolkit to eliminate the "boilerplate" from hand-written Redux logic, prevent common mistakes, and provide APIs that simplify standard Redux tasks**.

To change something in the state, you need to dispatch an action. An action is a plain JavaScript object (notice how we don’t introduce any magic?) that describes what happened. Here are a few example actions:

Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magical about it—it’s just a function that takes state and action as arguments, and returns the next state of the app.