

**Store**

A store holds the whole [state tree](https://redux.js.org/understanding/thinking-in-redux/glossary#state) of your application. The only way to change the state inside it is to dispatch an [action](https://redux.js.org/understanding/thinking-in-redux/glossary#action) on it.

A store is not a class. It's just an object with a few methods on it. To create it, pass your root [reducing function](https://redux.js.org/understanding/thinking-in-redux/glossary#reducer) to [createStore](https://redux.js.org/api/createstore).

**REDUCER:**In redux, the reducers are the [**pure functions**](https://www.geeksforgeeks.org/pure-functions-in-javascript/) that contain the logic and calculation that needed to be performed on the state. These functions accept the initial state of the state being used and the action type. It updates the state and responds with the new state. This updated state is sent back to the view components of the react to make the necessary changes. Basically, In short, we can say that Reducer’s work is to return the updated state and to also describe how the state changes.

Actions are the only source of information for the store as per Redux official documentation. It carries a payload of information from your application to store.

As discussed earlier, actions are plain JavaScript object that must have a type attribute to indicate the type of action performed. It tells us what had happened. Types should be defined as string constants in your application as given below

## Introduction

Dispatching actions in Redux is the fundamental method of updating a Redux store's state. Actions are used to store relevant information for the state, and they reach the store through the dispatch() method available on the store object. You can use either store.dispatch() directly or this.props.dispatch() for dispatching these actions. This guide walks you through the comparison between the two methods to help you understand which method is better and why.

createStore, combineReducers, applyMiddleware, bindActionCreator, compose

# createStore(reducer, [preloadedState], [enhancer])

Creates a Redux [store](https://redux.js.org/api/store) that holds the complete state tree of your app. There should only be a single store in your app.

#### Arguments[​](https://redux.js.org/api/createstore#arguments)

1. reducer (Function): A [reducing function](https://redux.js.org/understanding/thinking-in-redux/glossary#reducer) that returns the next [state tree](https://redux.js.org/understanding/thinking-in-redux/glossary#state), given the current state tree and an [action](https://redux.js.org/understanding/thinking-in-redux/glossary#action) to handle.
2. [preloadedState] (any): The initial state. You may optionally specify it to hydrate the state from the server in universal apps, or to restore a previously serialized user session. If you produced reducer with [combineReducers](https://redux.js.org/api/combinereducers), this must be a plain object with the same shape as the keys passed to it. Otherwise, you are free to pass anything that your reducer can understand.
3. [enhancer] (Function): The store enhancer. You may optionally specify it to enhance the store with third-party capabilities such as middleware, time travel, persistence, etc. The only store enhancer that ships with Redux is [applyMiddleware()](https://redux.js.org/api/applymiddleware).

# combineReducers(reducers)

As your app grows more complex, you'll want to split your [reducing function](https://redux.js.org/understanding/thinking-in-redux/glossary#reducer) into separate functions, each managing independent parts of the [state](https://redux.js.org/understanding/thinking-in-redux/glossary#state).

The combineReducers helper function turns an object whose values are different reducing functions into a single reducing function you can pass to [createStore](https://redux.js.org/api/createstore).

The resulting reducer calls every child reducer, and gathers their results into a single state object. **The state produced by combineReducers() namespaces the states of each reducer under their keys as passed to combineReducers()**

# \ applyMiddleware(...middleware)

Middleware is the suggested way to extend Redux with custom functionality. Middleware lets you wrap the store's [dispatch](https://redux.js.org/api/store#dispatchaction) method for fun and profit. The key feature of middleware is that it is composable. Multiple middleware can be combined together, where each middleware requires no knowledge of what comes before or after it in the chain.

# bindActionCreators(actionCreators, dispatch)

Turns an object whose values are [action creators](https://redux.js.org/understanding/thinking-in-redux/glossary#action-creator), into an object with the same keys, but with every action creator wrapped into a [dispatch](https://redux.js.org/api/store#dispatchaction) call so they may be invoked directly.

Normally you should just call [dispatch](https://redux.js.org/api/store#dispatchaction) directly on your [Store](https://redux.js.org/api/store) instance. If you use Redux with React, [react-redux](https://github.com/gaearon/react-redux) will provide you with the [dispatch](https://redux.js.org/api/store#dispatchaction) function so you can call it directly, too.

The only use case for bindActionCreators is when you want to pass some action creators down to a component that isn't aware of Redux, and you don't want to pass [dispatch](https://redux.js.org/api/store#dispatchaction) or the Redux store to it.

# compose(...functions)

Composes functions from right to left.

This is a functional programming utility, and is included in Redux as a convenience. You might want to use it to apply several [store enhancers](https://redux.js.org/understanding/thinking-in-redux/glossary#store-enhancer) in a row.

#### Arguments[​](https://redux.js.org/api/compose#arguments)

1. (arguments): The functions to compose. Each function is expected to accept a single parameter. Its return value will be provided as an argument to the function standing to the left, and so on. The exception is the right-most argument which can accept multiple parameters, as it will provide the signature for the resulting composed function.

#### Returns[​](https://redux.js.org/api/compose#returns)

(Function): The final function obtained by composing the given functions from right to left.

#### Example[​](https://redux.js.org/api/compose#example)

This example demonstrates how to use compose to enhance a [store](https://redux.js.org/api/store) with [applyMiddleware](https://redux.js.org/api/applymiddleware) and a few developer tools from the [redux-devtools](https://github.com/reduxjs/redux-devtools) package.

import { createStore, applyMiddleware, compose } from 'redux'  
import thunk from 'redux-thunk'  
import DevTools from './containers/DevTools'  
import reducer from '../reducers'  
  
const store = createStore(  
 reducer,  
 compose(applyMiddleware(thunk), DevTools.instrument())  
)