



# Redux Overview and Concepts

Mohamed ROMDANE

September 2021

[formation@ineotec.com](mailto:formation@ineotec.com)

# Prerequisites

- Familiarity with HTML & CSS.
- Familiarity with ES6 syntax and features
- Knowledge of React terminology: JSX, State, Function Components, Props, and Hooks
- Knowledge of asynchronous JavaScript and making AJAX requests
- **If you're not already comfortable with those topics, we encourage you to take some time to become comfortable with them first, and then come back to learn about Redux.** We'll be here when you're ready!
- You should make sure that you have the React and Redux DevTools extensions installed in your browser:
- React DevTools Extension:
  - React DevTools Extension for Chrome
  - React DevTools Extension for Firefox
- Redux DevTools Extension:
  - Redux DevTools Extension for Chrome
  - Redux DevTools Extension for Firefox

# What is Redux?#

- It helps to understand what this "Redux" thing is in the first place. What does it do? What problems does it help me solve? Why would I want to use it?
- **Redux is a pattern and library for managing and updating application state, using events called "actions".**
- It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

# Why Should I Use Redux?#

- Redux helps you manage "global" state - state that is needed across many parts of your application.
- **The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur.**
- Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

# When Should I Use Redux?#

- Redux helps you deal with shared state management, but like any tool, it has tradeoffs. Redux is more useful when:
- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex
- The app has a medium or large-sized codebase, and might be worked on by many people
- **Not all apps need Redux. Take some time to think about the kind of app you're building, and decide what tools would be best to help solve the problems you're working on.**

# Redux Libraries and Tools#

- Redux is a small standalone JS library. However, it is commonly used with several other packages:
- **React-Redux#**
- Redux can integrate with any UI framework, and is most frequently used with React. **React-Redux** is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.
- **Redux Toolkit#**
- **Redux Toolkit** is our recommended approach for writing Redux logic. It contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.
- **Redux DevTools Extension#**
- The **Redux DevTools Extension** shows a history of the changes to the state in your Redux store over time. This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".

# State Management#

- Let's start by looking at a small React counter component. It tracks a number in component state, and increments the number when a button is clicked:
- It is a self-contained app with the following parts:
- The **state**, the source of truth that drives our app;
- The **view**, a declarative description of the UI based on the current state
- The **actions**, the events that occur in the app based on user input, and trigger updates in the state

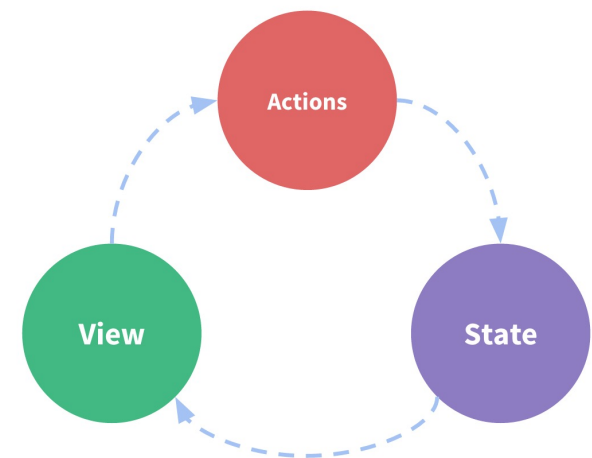
# State Management Example

```
function Counter() {  
  // State: a counter value  
  const [counter, setCounter] = useState(0)  
  
  // Action: code that causes an update to the state when something happens  
  const increment = () => {  
    setCounter(prevCounter => prevCounter + 1)  
  }  
  
  // View: the UI definition  
  return (  
    <div>  
      Value: {counter} <button onClick={increment}>Increment</button>  
    </div>  
  )  
}
```



# One WayData Flow

- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state



# Main Idea

- The simplicity can break down when we have **multiple components that need to share and use the same state**, especially if those components are located in different parts of the application. Sometimes this can be solved by "lifting state up" to parent components, but that doesn't always help.
- One way to solve this is to extract the shared state from the components, and put it into a centralized location outside the component tree. With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!
- By defining and separating the concepts involved in state management and enforcing rules that maintain independence between views and states, we give our code more structure and maintainability.
- This is the basic idea behind Redux: a single centralized place to contain the global state in your application, and specific patterns to follow when updating that state to make the code predictable.

# Immutability#

- "Mutable" means "changeable". If something is "immutable", it can never be changed.
- JavaScript objects and arrays are all mutable by default. If I create an object, I can change the contents of its fields. If I create an array, I can change the contents as well. Fig1
- This is called *mutating* the object or array. It's the same object or array reference in memory, but now the contents inside the object have changed.
- **In order to update values immutably, your code must make *copies* of existing objects/arrays, and then modify the copies.**
- We can do this by hand using JavaScript's array / object spread operators, as well as array methods that return new copies of the array instead of mutating the original array. Fig2
- **Redux expects that all state updates are done immutably.**

# Immutability Example

```
const obj = { a: 1, b: 2 }  
// still the same object outside, but the contents have changed  
obj.b = 3  
  
const arr = ['a', 'b']  
// In the same way, we can change the contents of this array  
arr.push('c')  
arr[1] = 'd'
```

```
const obj = {  
  a: {  
    // To safely update obj.a.c, we have to copy each piece  
    c: 3  
  },  
  b: 2  
}  
  
const obj2 = {  
  // copy obj  
  ...obj,  
  // overwrite a  
  a: {  
    // copy obj.a  
    ...obj.a,  
    // overwrite c  
    c: 42  
  }  
}  
  
const arr = ['a', 'b']  
// Create a new copy of arr, with "c" appended to the end  
const arr2 = arr.concat('c')  
  
// or, we can make a copy of the original array:  
const arr3 = arr.slice()  
// and mutate the copy:  
arr3.push('c')
```

# Actions#

- An **action** is a plain JavaScript object that has a type field. **You can think of an action as an event that describes something that happened in the application.**
- The type field should be a string that gives this action a descriptive name, like "todos/todoAdded". We usually write that type string like "domain/eventName", where the first part is the feature or category that this action belongs to, and the second part is the specific thing that happened.
- An action object can have other fields with additional information about what happened. By convention, we put that information in a field called payload.
- A typical action object might look like this:
- `const addTodoAction = { type: 'todos/todoAdded', payload: 'Buy milk'}`

# Action Creators#

- An **action creator** is a function that creates and returns an action object.
- We typically use these so we don't have to write the action object by hand every time:
- `const addTodo = text => { return { type: 'todos/todoAdded', payload: text }}`

# Reducers#

- A **reducer** is a function that receives the current state and an action object, decides how to update the state if necessary, and returns the new state: `(state, action) => newState`.
- **You can think of a reducer as an event listener which handles events based on the received action (event) type.**
- Reducers must *always* follow some specific rules:
- They should only calculate the new state value based on the state and action arguments
- They are not allowed to modify the existing state. Instead, they must make *immutable updates*, by copying the existing state and making changes to the copied values.
- They must not do any asynchronous logic, calculate random values, or cause other "side effects"
- We'll talk more about the rules of reducers later, including why they're important and how to follow them correctly.
- The logic inside reducer functions typically follows the same series of steps:
- Check to see if the reducer cares about this action
  - If so, make a copy of the state, update the copy with new values, and return it
- Otherwise, return the existing state unchanged

# Reducer Example

- Reducers can use any kind of logic inside to decide what the new state should be: if/else, switch, loops, and so on.

```
const initialState = { value: 0 }

function counterReducer(state = initialState, action) {
  // Check to see if the reducer cares about this action
  if (action.type === 'counter/increment') {
    // If so, make a copy of `state`
    return {
      ...state,
      // and update the copy with the new value
      value: state.value + 1
    }
  }
  // otherwise return the existing state unchanged
  return state
}
```



# Why Are They Called 'Reducers?'

- The `Array.reduce()` method lets you take an array of values, process each item in the array one at a time, and return a single final result.
- You can think of it as "reducing the array down to one value".
- `Array.reduce()` takes a callback function as an argument, which will be called one time for each item in the array. It takes two arguments:
- `previousResult`, the value that your callback returned last time
- `currentItem`, the current item in the array
- The first time that the callback runs, there isn't a `previousResult` available, so we need to also pass in an initial value that will be used as the first `previousResult`.
- If we wanted to add together an array of numbers to find out what the total is, we could write a reduce callback that looks like this:
- Fig 1
- Notice that this `addNumbers` "reduce callback" function doesn't need to keep track of anything itself. It takes the `previousResult` and `currentItem` arguments, does something with them, and returns a new result value.
- **A Redux reducer function is exactly the same idea as this "reduce callback" function!** It takes a "previous result" (the state), and the "current item" (the action object), decides a new state value based on those arguments, and returns that new state.
- If we were to create an array of Redux actions, call `reduce()`, and pass in a reducer function, we'd get a final result the same way:
- Fig 2
- We can say that **Redux reducers reduce a set of actions (over time) into a single state.**
- The difference is that with `Array.reduce()` it happens all at once, and with Redux, it happens over the lifetime of your running app.

# Reducer Example

```
const actions = [  
  { type: 'counter/increment' },  
  { type: 'counter/increment' },  
  { type: 'counter/increment' }  
]  
  
const initialState = { value: 0 }  
  
const finalResult = actions.reduce(counterReducer, initialState)  
console.log(finalResult)  
// {value: 3}
```

```
const numbers = [2, 5, 8]  
  
const addNumbers = (previousResult, currentItem) => {  
  console.log({ previousResult, currentItem })  
  return previousResult + currentItem  
}  
  
const initialValue = 0  
  
const total = numbers.reduce(addNumbers, initialValue)  
// {previousResult: 0, currentItem: 2}  
// {previousResult: 2, currentItem: 5}  
// {previousResult: 7, currentItem: 8}  
  
console.log(total)  
// 15
```

# Store#

- The current Redux application state lives in an object called the **store**.
- The store is created by passing in a reducer, and has a method called `getState` that returns the current state value:

```
import { configureStore } from '@reduxjs/toolkit'

const store = configureStore({ reducer: counterReducer })

console.log(store.getState())
// {value: 0}
```

# Dispatch#

- The Redux store has a method called dispatch.
- **The only way to update the state is to call store.dispatch() and pass in an action object.**
- The store will run its reducer function and save the new state value inside, and we can call getState() to retrieve the updated value:
- Fig 1
- **You can think of dispatching actions as "triggering an event"** in the application.
- Something happened, and we want the store to know about it.
- Reducers act like event listeners, and when they hear an action they are interested in, they update the state in response.
- We typically call action creators to dispatch the right action:
- Fig 2

# Dispatch Figures

```
store.dispatch({ type: 'counter/increment' })  
  
console.log(store.getState())  
// {value: 1}
```

```
const increment = () => {  
  return {  
    type: 'counter/increment'  
  }  
}  
  
store.dispatch(increment())  
  
console.log(store.getState())  
// {value: 2}
```

# Selectors#

- **Selectors** are functions that know how to extract specific pieces of information from a store state value.
- As an application grows bigger, this can help avoid repeating logic as different parts of the app need to read the same data:

```
const selectCounterValue = state => state.value

const currentValue = selectCounterValue(store.getState())
console.log(currentValue)
// 2
```

# Redux Application Data Flow#

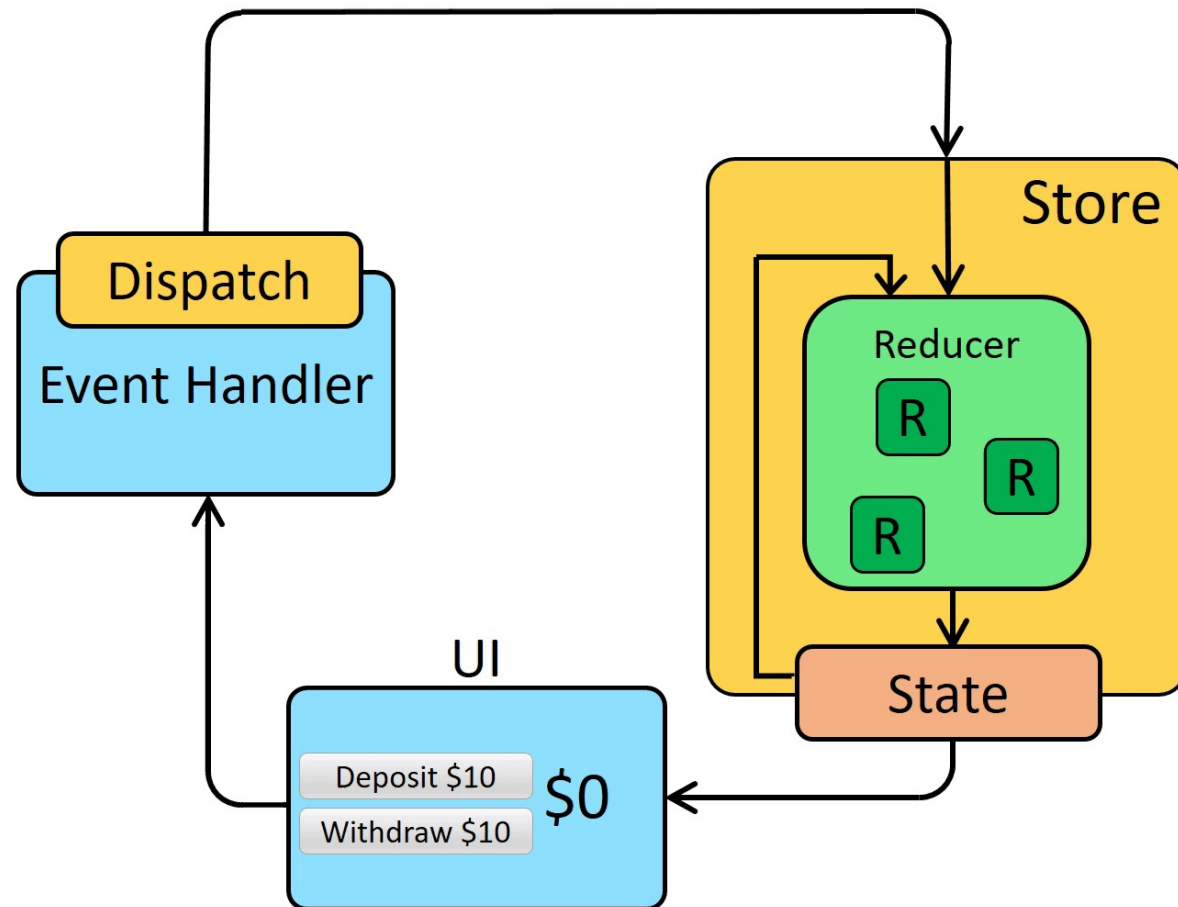
- Earlier, we talked about "one-way data flow", which describes this sequence of steps to update the app:
- State describes the condition of the app at a specific point in time
- The UI is rendered based on that state
- When something happens (such as a user clicking a button), the state is updated based on what occurred
- The UI re-renders based on the new state

# Detailed Steps

- Initial setup:
  - A Redux store is created using a root reducer function
  - The store calls the root reducer once, and saves the return value as its initial state
  - When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.
- Updates:
  - Something happens in the app, such as a user clicking a button
  - The app code dispatches an action to the Redux store, like `dispatch({type: 'counter/increment'})`
  - The store runs the reducer function again with the previous state and the current action, and saves the return value as the new state
  - The store notifies all parts of the UI that are subscribed that the store has been updated
  - Each UI component that needs data from the store checks to see if the parts of the state they need have changed.
  - Each component that sees its data has changed forces a re-render with the new data, so it can update what's shown on the screen



# Data Flow



# Summary

- **Redux is a library for managing global application state**
  - Redux is typically used with the React-Redux library for integrating Redux and React together
  - Redux Toolkit is the recommended way to write Redux logic
- **Redux uses a "one-way data flow" app structure**
  - State describes the condition of the app at a point in time, and UI renders based on that state
  - When something happens in the app:
    - The UI dispatches an action
    - The store runs the reducers, and the state is updated based on what occurred
    - The store notifies the UI that the state has changed
  - The UI re-renders based on the new state
- **Redux uses several types of code**
  - *Actions* are plain objects with a type field, and describe "what happened" in the app
  - *Reducers* are functions that calculate a new state value based on previous state + an action
  - A Redux *store* runs the root reducer whenever an action is *dispatched*

# Urlographie

- <https://decembersoft.com/posts/starting-a-react-redux-project-with-typescript/>