how to manage state in your React application with Redux.

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. There are more concepts to learn, and more code to write. It also adds some indirection to your code, and asks you to follow certain restrictions. It's a trade-off between short term and long term productivity.

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

- **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:

```
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>  )}
```

Copy

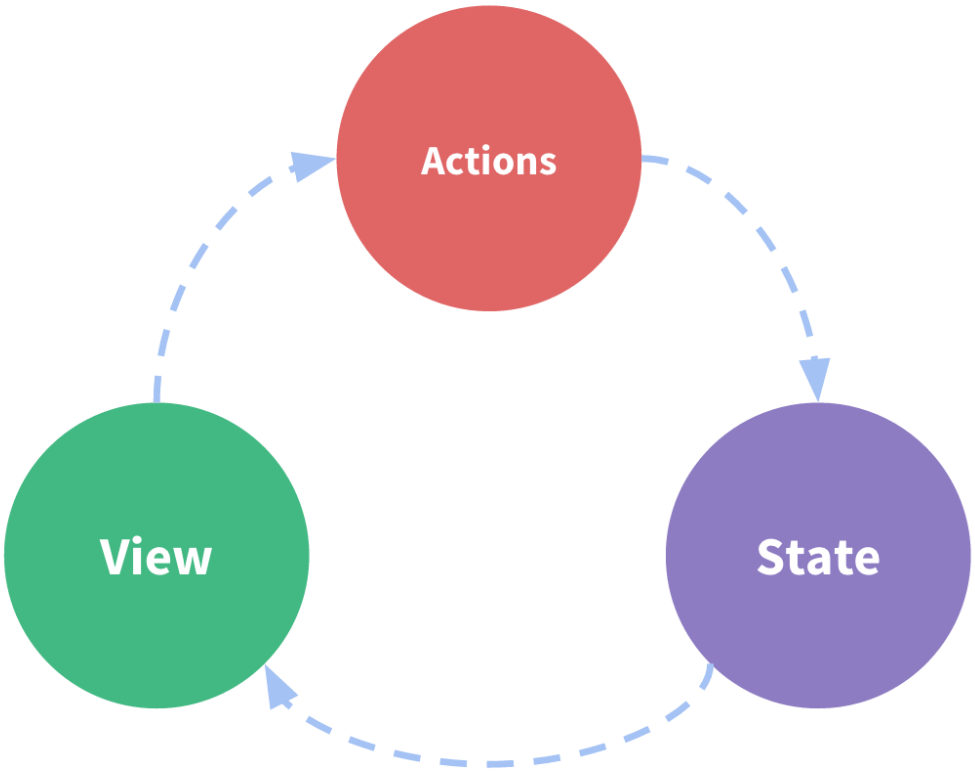
It is a self-contained app with the following parts:

- **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

This is a small example of "one-way data 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



However, 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:

```
const obj = { a: 1, b: 2 }// still the same object outside, but the contents have changedobj.b = 3

const arr = ['a', 'b']// In the same way, we can change the contents of this arrayarr.push('c')arr[1] = 'd'
```

Copy

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:

```
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 endconst 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')
```

Copy

Redux expects that all state updates are done immutably. We'll look at where and how this is important a bit later, as well as some easier ways to write immutable update logic.