Redux in React

React redux is an advanced state management library for React. In a big and complex application, large number of components are used. React recommends to move the state to the top level component and pass the state to the nested component using properties. It helps to some extent but it becomes complex when the components increases

React redux chips in and helps to maintain state at the application level. React redux allows any component to access the state at any time. Also, it allows any component to change the state of the application at any time.

Redux is a predictable state container for JavaScript apps.

**Why Use Redux?**

We will also see how some of its core building blocks work, such as store, actions, and reducers and how they all come together and make Redux the global state management library that it is.

Let's take an e-commerce website for example. An e-commerce website will have several components like the cart component, user profile component, previously viewed section component, and so on.

We'll take the cart component which displays the number of items in a user's cart. The state of the cart component will consist of all the items the user has added to the cart and the total number of those items. At all times the application is up and running, this component has to show the updated number of items in the user's cart.

Whenever a user adds an item to the cart, the application has to internally handle that action by adding that item to the cart object. It has to maintain its state internally and also show the user the total number of items in the cart in the UI.

Similarly, removing an item from the cart should decrease the number of items in the cart internally. It should remove the item from the cart object and also display the updated total number of items in the cart in the UI.

We may very well maintain the internal state of the components inside them, but as and when an application grows bigger, it may have to share some state between components. This is not just only to show them in the view, but also to manage or update them or perform some logic based on their value.

This task of handling multiple states from multiple components efficiently can become challenging when the application grows in size.

This is where Redux comes into the picture. Being a state management library, Redux will basically store and manage all the application's states.

It also provides us with some important APIs using which we can make changes to the existing state as well as fetch the current state of the application.

**What Makes Redux Predictable?**

State is **Read-only** in Redux. What makes Redux predictable is that to make a change in the state of the application, we need to dispatch an action which describes what changes we want to make in the state.

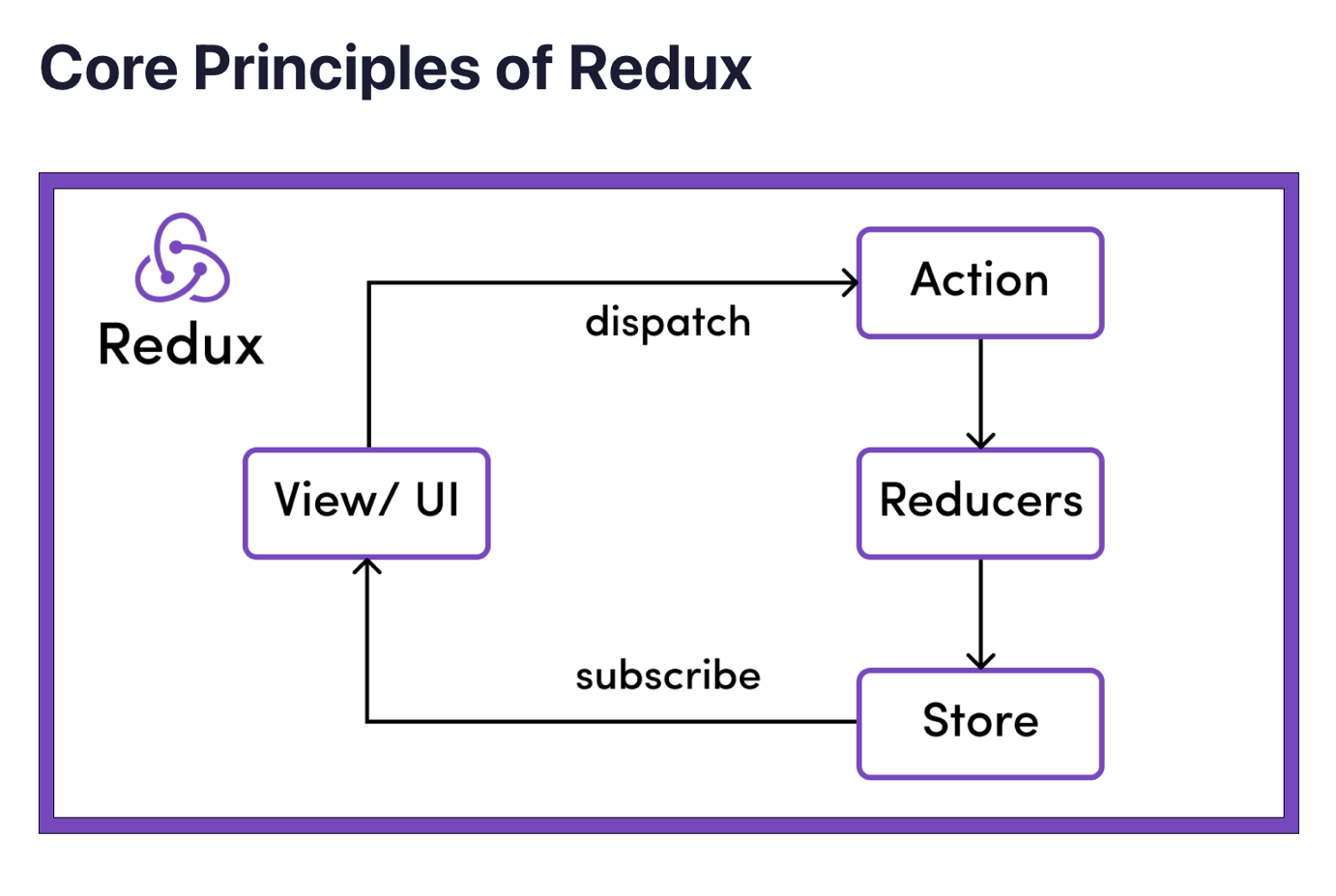
These actions are then consumed by something known as reducers, whose sole job is to accept two things (the action and the current state of the application) and return a new updated instance of the state.

We'll talk more about actions and reducers in the following sections.

Note that reducers do not change any part of the state. Rather a reducer produces a new instance of the state with all the necessary updates.

So continuing with our above example of an e-commerce website, if the initial state of the cart is that it has 0 items, then an action of **adding one item** to the cart will increase the number of items in the cart by 1. And firing the action of **adding one item** to the cart again will increase the number of items in the cart to 2.

Given an initial state, with a specific list of **actions** in a specific order, it'll always provide us with the exact same final state of the entity. This is how Redux makes state management predictable.



### What is the Redux Store?

*The global state of an application is stored in an object tree within a single store*

The Redux store is the main, central bucket which stores all the states of an application. It should be considered and maintained as a **single source of truth** for the state of the application.

If the store is provided to the **App.js** (by wrapping the App component within the <Provider> </Provider> tag) as shown in the code snippet below, then all its children (children components of App.js) can also access the state of the application from the store. This makes it act as a global state.

A diagram of a child

Description automatically generated

The state of the whole application is stored in the form of a **JS object tree** in a **single store** as shown below.

// this is how the store object structure looks like

{

noOfItemInCart: 2,

cart: [

{

bookName: "Harry Potter and the Chamber of Secrets",

noOfItem: 1,

},

{

bookName: "Harry Potter and the Prisoner of Azkaban",

noOfItem: 1

}

]

}

### What Are Actions in Redux?

*The only way to change the state is to emit an action, which is an object describing what happened*

As mentioned above, state in Redux is read-only. This helps you restrict any part of the view or any network calls to write/update the state directly.

Instead, if anyone wants to change the state of the application, then they'll need to express their intention of doing so by **emitting or dispatching an action**.

Let's take the example of the above store example where we have 2 books in the store: "Harry Potter and the Chamber of Secrets" and "Harry Potter and the Prisoner of Azkaban". There's just one copy of each.

Now if the user wants to add another item to the cart, then they will have to click on the **"Add to Cart"** button next to the item.

On the click of the **"Add to Cart"** button, an action will be dispatched. This action is nothing but a JS object describing what changes need to be done in the store. Something like this:

// Rest of the code

const dispatch = useDispatch()

const addItemToCart = () => {

return {

type: "ADD\_ITEM\_TO\_CART"

payload: {

bookName: "Harry Potter and the Goblet of Fire",

noOfItem: 1,

}

}

}

<button onClick = {() => dispatch(addItemToCart())}>Add to cart</button>

Note how in the above example, we dispatch an action on click of the button. Or rather, to be more specific, we dispatch something known as an **action creator** – that is, the function addItemToCart(). This in turn returns an action which is a plain JS object describing the purpose of the action denoted by the type key along with any other data required for the state change. In this case, it's the name of the book to be added to the cart denoted by the payload key.

**Every action must have at least** a type associated with it. Any other detail that needs to be passed is optional and will depend on the type of action we dispatch.

For example, the above code snippet dispatches the following action:

// Action that got created by the action creator addItemToCart()

{

type: "ADD\_ITEM\_TO\_CART" // Note: Every action must have a type key

payload: {

bookName: "Harry Potter and the Goblet of Fire",

noOfItem: 1,

}

}

### What Are Reducers in Redux?

*To specify how the state tree is transformed by actions, we write pure reducers*

A diagram of a process

Description automatically generated

Reducers, as the name suggests, take in two things: **previous state** and **an action**. Then they reduce it (read it return) to one entity: the **new updated instance of state**.

So reducers are basically pure JS functions which take in the previous state and an action and return the newly updated state.

There can either be one reducer if it is a simple app or multiple reducers taking care of different parts or slices of the global state in a bigger application.

For example, there can be a reducer handling the state of the cart in a shopping application, then there can be a reducer handling the user details part of the application, and so on.

Whenever an action is dispatched, **all the reducers are activated**. Each reducer filters out the action using a switch statement switching on the **action type**. Whenever the switch statement matches with the action passed, the corresponding reducers take the necessary action to make the update and return a fresh new instance of the global state.

Continuing with our above example, we can have a reducer as follows:

const initialCartState = {

noOfItemInCart: 0,

cart: []

}

// NOTE:

// It is important to pass an initial state as default to

// the state parameter to handle the case of calling

// the reducers for the first time when the

// state might be undefined

const cartReducer = (state = initialCartState, action) => {

switch (action.type) {

case "ADD\_ITEM\_TO\_CART":

return {

...state,

noOfItemInCart: state.noOfItemInCart + 1,

cart : [

...state.cart,

action.payload

]

}

case "DELETE\_ITEM\_FROM\_CART":

return {

// Remaining logic

}

default:

return state

} // Important to handle the default behaviour

} // either by returning the whole state as it is

// or by performing any required logic

In the above code snippet, we created a reducer called cartReducer which is a pure JS function. This function accepts two parameters: state and action.

Note that the state parameter is a default parameter which accepts an initial state. This is to handle the scenario when **the reducer is called for the first time** when the state value is undefined.

Also note that every reducer should handle the default case where, if none of the switch cases match with the passed action, then the reducer should return state as it is or perform any required logic on it before passing the state.

Whenever we dispatch an action with a certain type, we need to make sure to have appropriate reducers to handle that action.

In the above example, on clicking the button, we had dispatched an **action** with an **action creator** called addItemToCart(). This action creator has dispatched an action with the type ADD\_ITEM\_TO\_CART.

Next, we have created a **reducer** called cartReducer which takes the state (with the default initial state) and the action as parameters. It switches on the **action type**, and then whichever case matches with the dispatched action type, it makes the necessary update and returns the fresh new version of the updated state.

Note here that **state in redux is immutable**. So, the reducers make a copy of the entire current state first, make the necessary changes, and then return a fresh new instance of the state – with all the necessary changes/ updates.

So in the above example, we first make a copy of the entire state using the spread operator ...state. Then we increment the noOfItemInCart by 1, update the cart array by adding the new object passed in the action.payload shown below, and then finally return the updated object.

{

bookName: "Harry Potter and the Goblet of Fire",

noOfItem: 1,

}

After the reducers have updated the state, if we go and console.log the state, then we would see the following result:

// Updated store

{

noOfItemInCart: 3, // Incremented by 1

cart: [

{

bookName: "Harry Potter and the Chamber of Secrets",

noOfItem: 1,

},

{

bookName: "Harry Potter and the Prisoner of Azkaban",

noOfItem: 1

},

{ // Newly added object

bookName: "Harry Potter and the Goblet of Fire",

noOfItem: 1,

}

]

}