Redux

"Redux is a predictable state container for JavaScript apps“

Redux is for JavaScript applications

Redux is not tied to React

Can be used with React, Angular, Vue or even vanilla JavaScript

Redux is a library for JavaScript applications

Redux is a state container

Redux stores the state of your application

Consider a React app - state of a component

State of an app is the state represented by all the individual components of that app

<u>LoginFormComponent</u>

```
state = {
  username: ‘ ’,
  password: ‘ ’,
  submitting: false
}
```

<u>UserListComponent</u>

```
state = {
  users: [ ]
}
```

<u>Application</u>

```
state = {
  isUserLoggedIn: true,
  username: ‘Vishwas’,
  profileUrl: ‘ ’,
  onlineUsers: [ ],
  isModalOpened: false
}
```

Redux is predictable

Predictable in what way?

Redux is a state container

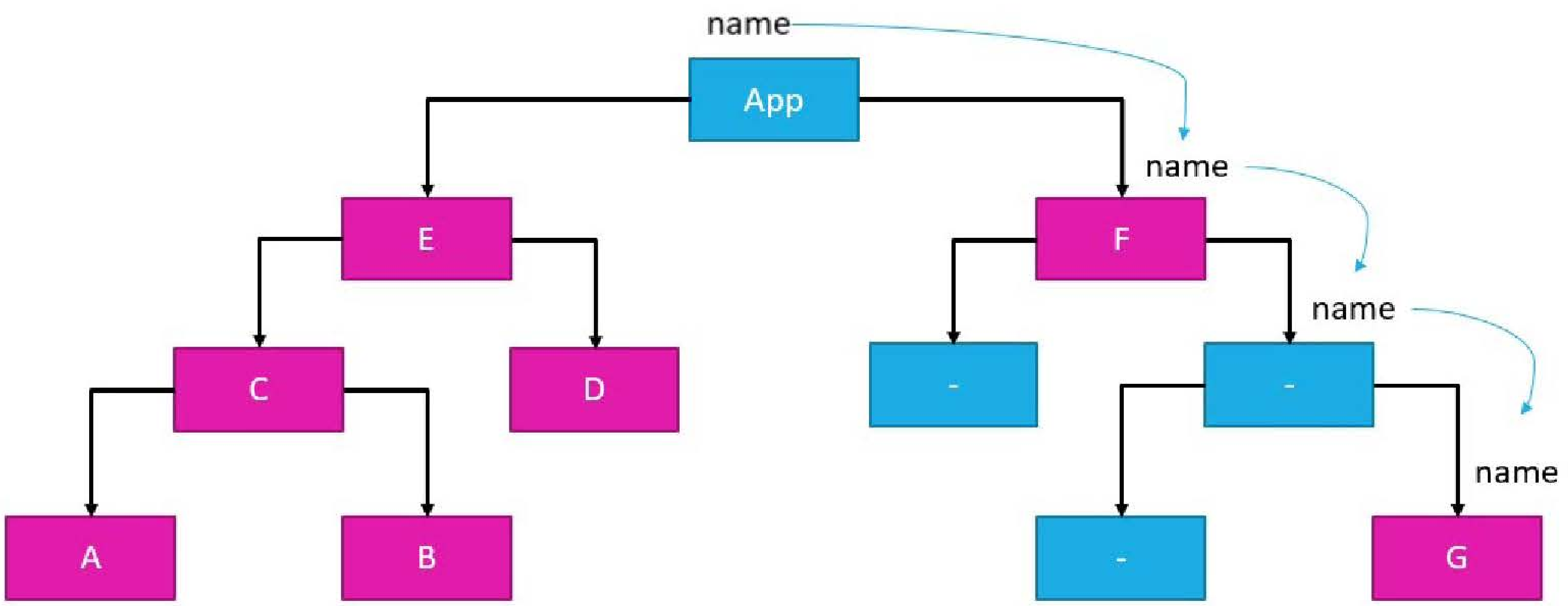
The state of the application can change

Ex: todo list app – item (pending) → item (completed)

In redux, all state transitions are explicit and it is possible to keep track of them

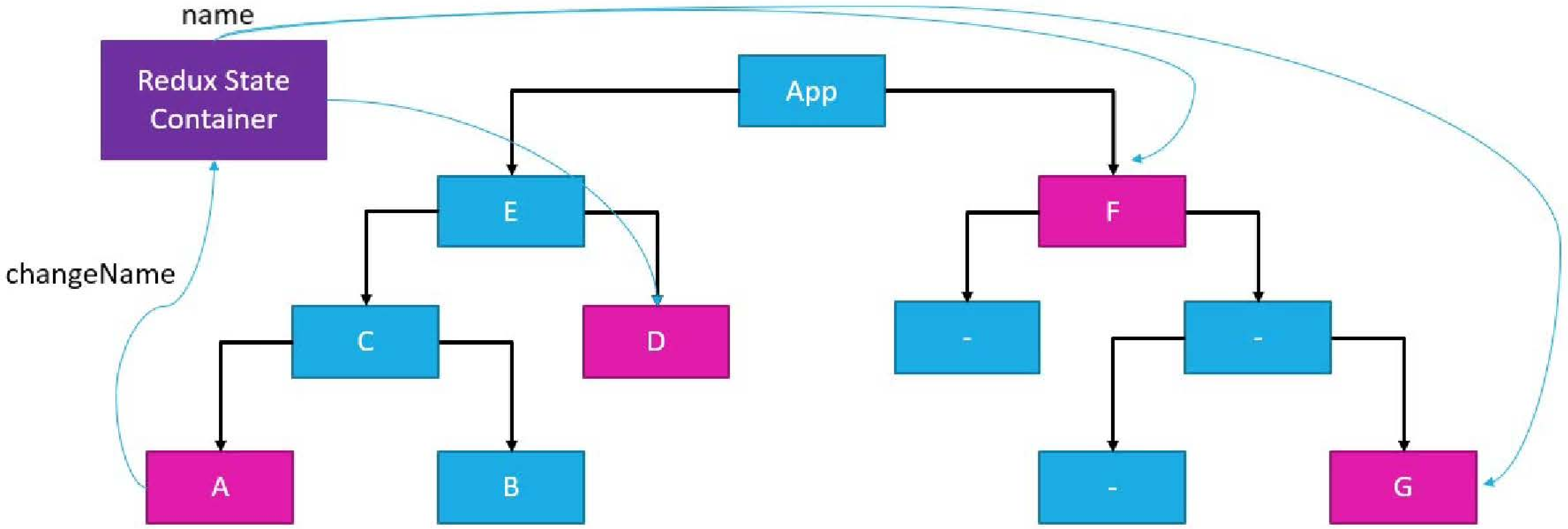
The changes to your application's state become predictable

State in a React App

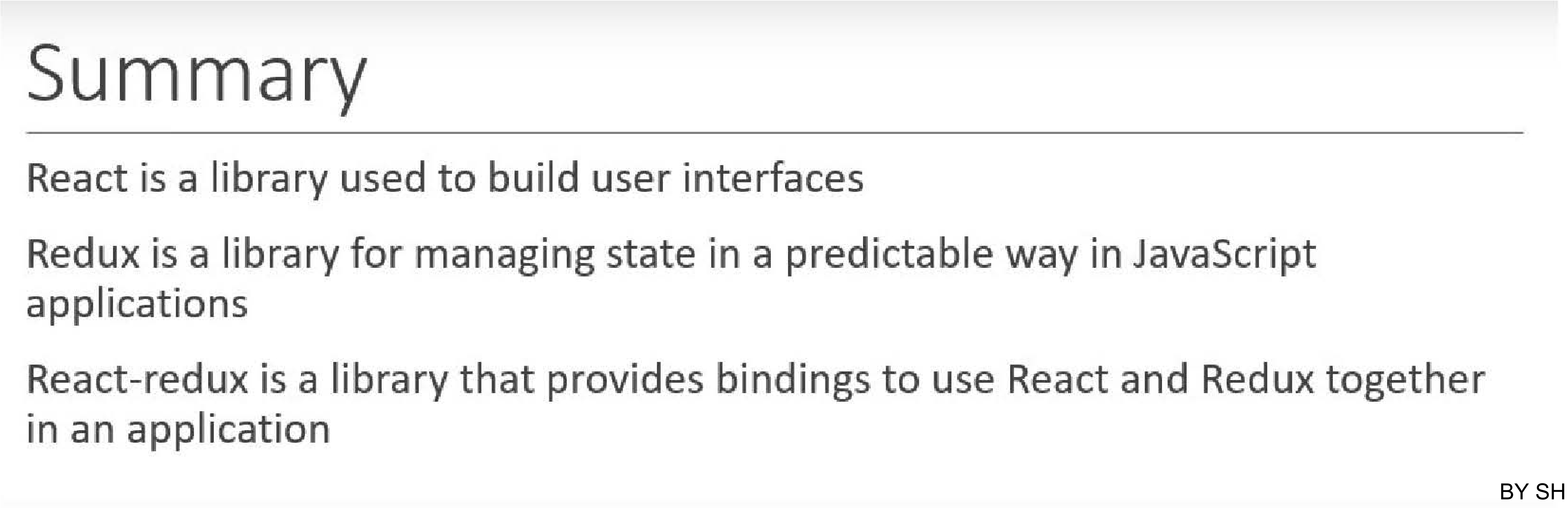


Why do we need redux even if react has its own state management
ans - Prop drilling

React + Redux



In redux, state is contained outside of the app component. If component A wants to change the state, it communicates with the state container, it updates the state and sends its value to only those components that actually use it



Summary

React is a library used to build user interfaces

Redux is a library for managing state in a predictable way in JavaScript applications

React-redux is a library that provides bindings to use React and Redux together in an application

BY SHIVKARAMAN

Three Core Concepts

Cake Shop

Entities

Shop – Stores cakes on a shelf
Shopkeeper – At the front of the store
Customer – At the store entrance

Activities

Customer – Buy a cake
Shopkeeper – Remove a cake from the shelf
– Receipt to keep track

Cake Shop Scenario	Redux	Purpose
Shop	Store	Holds the state of your application
Intention to BUY_CAKE	Action	Describes what happened
Shopkeeper	Reducer	Ties the store and actions together

A **store** that holds the state of your application.

An **action** that describes the changes in the state of the application.

A **reducer** which actually carries out the state transition depending on the action.

Three Principles

First Principle

"***The state of your whole application is stored in an object tree within a single store***"

Maintain our application state in a single object which would be managed by the Redux store

Cake Shop –

Let's assume we are tracking the number of cakes on the shelf Then we will represent the state of cake shop as the no of cakes present in the shop. Now redux wil manage the state

```
{
  numberOfCakes: 10
}
```

Second Principle

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

To update the state of your app, you need to let Redux know about that with an action

Not allowed to directly update the state object

Cake Shop: You are not allowed to directly take the cake from the shelf. We need to let the shopkeeper know about our action

Let the shopkeeper know about our action – BUY_CAKE

```
{
  type: BUY_CAKE
}
```

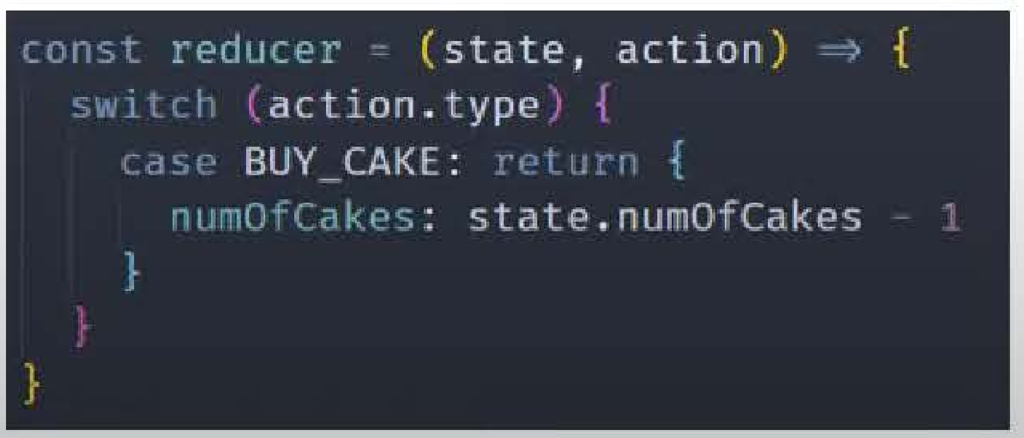
Third Principle

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

Reducer - (previousState, action) => newState It takes previous state, performs action and returns new state

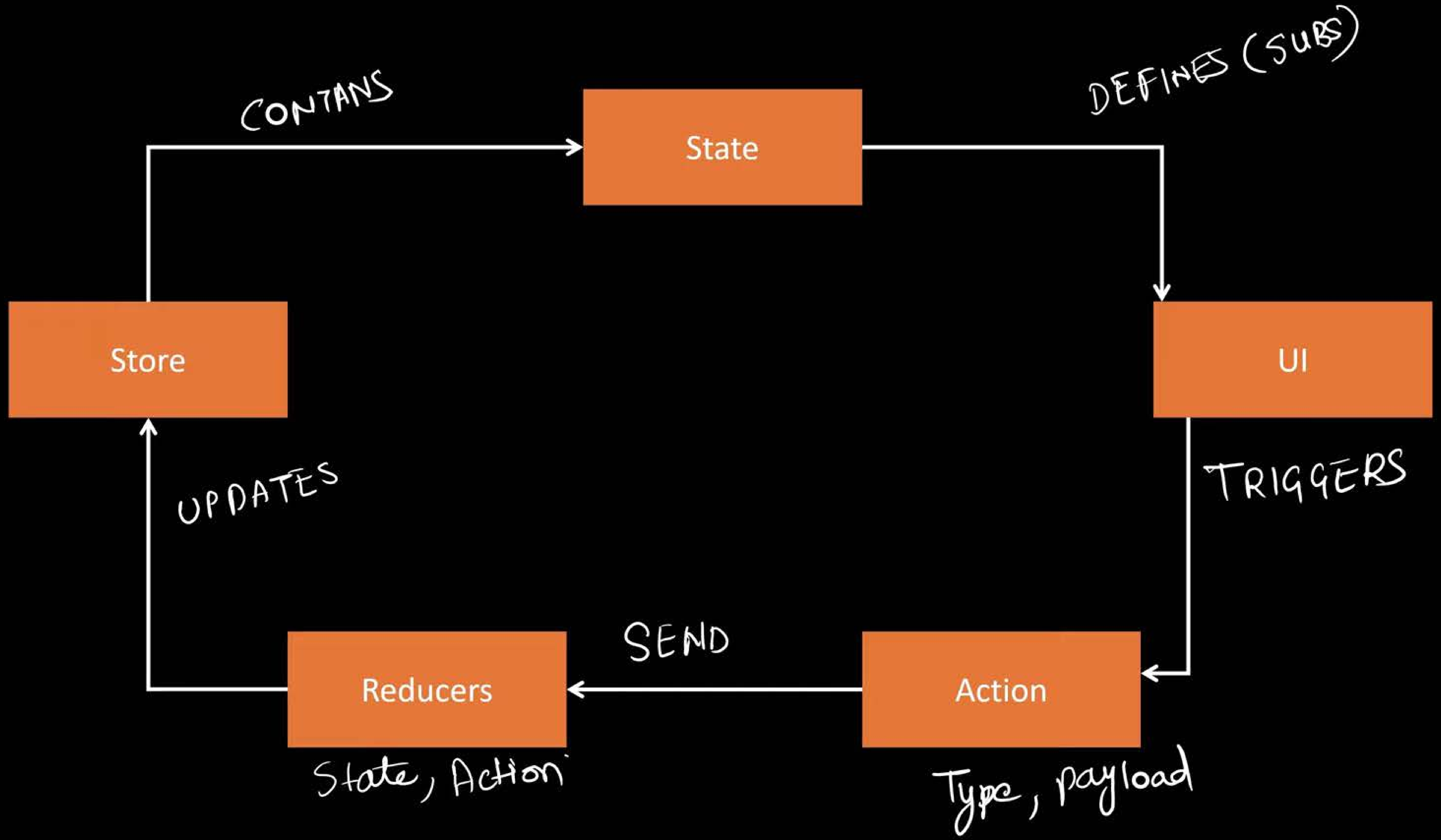
Cake Shop

Reducer is the shopkeeper



```
const reducer = (state, action) => {
  switch (action.type) {
    case BUY_CAKE: return {
      numOfCakes: state.numOfCakes - 1
    }
  }
}
```

Redux Architecture



State

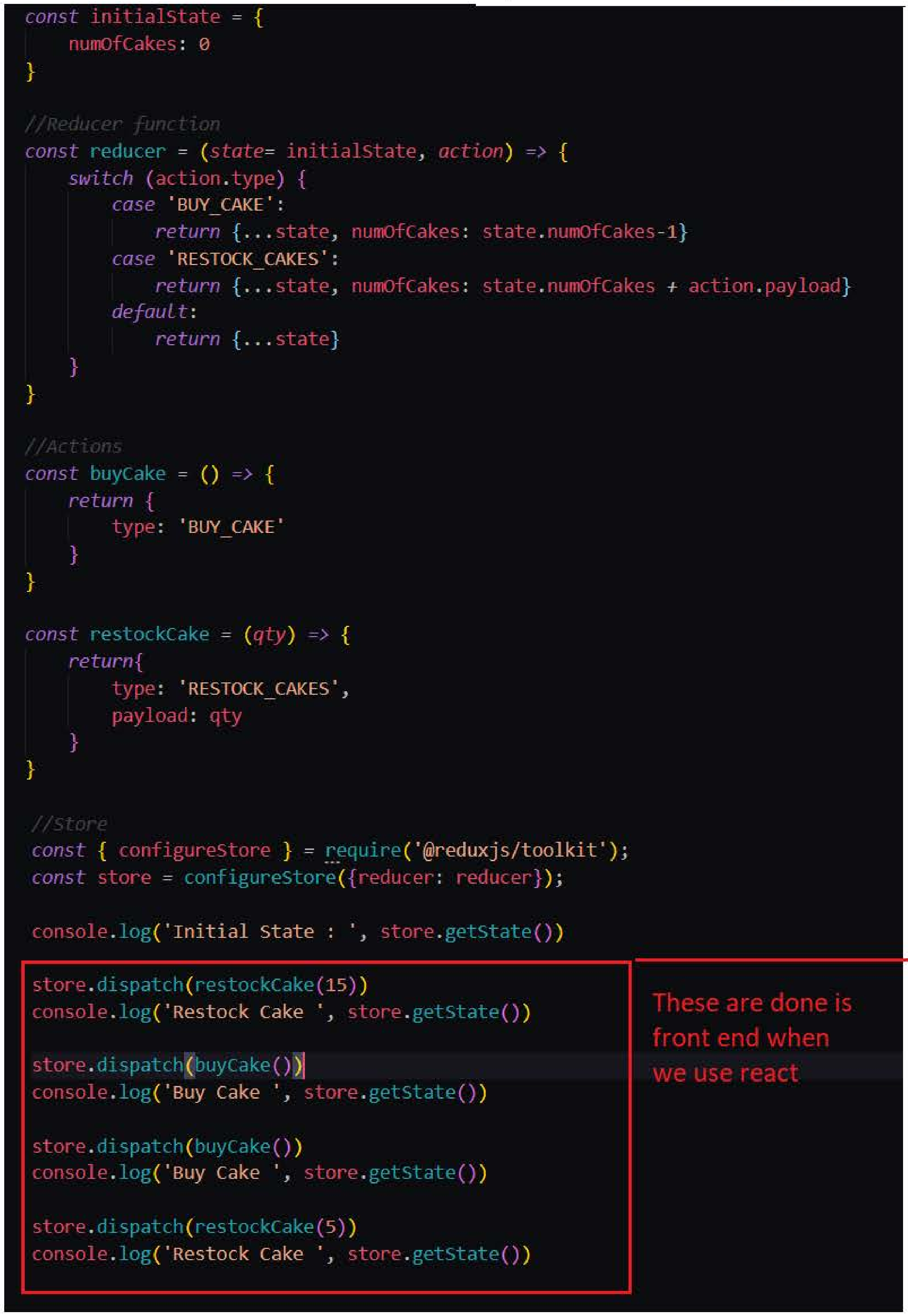
UI

Action

Reducers

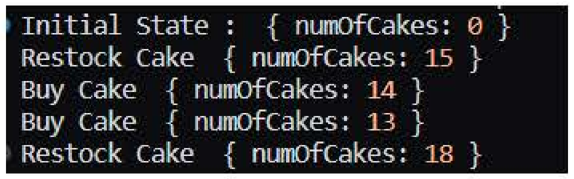
Store

A normal JS application which uses redux

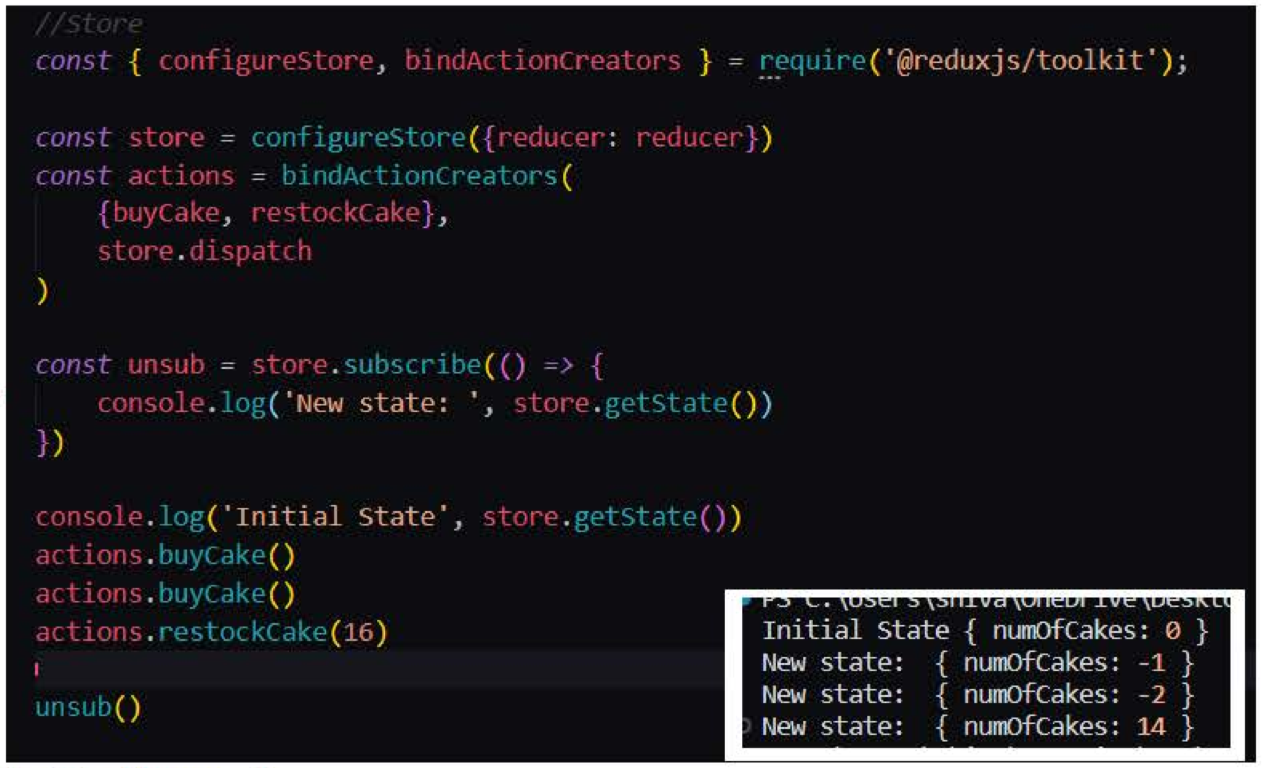


```
const initialState = {
    numOfCakes: 0
}

//Reducer function
const reducer = (state= initialState, action) => {
    switch (action.type) {
        case 'BUY_CAKE':
            return {...state, numOfCakes: state.numOfCakes-1}
        case 'RESTOCK_CAKES':
            return {...state, numOfCakes: state.numOfCakes + action.payload}
        default:
            return {...state}
    }
}

//Actions
const buyCake = () => {
    return {
        type: 'BUY_CAKE'
    }
}

const restockCake = (qty) => {
    return{
        type: 'RESTOCK_CAKES',
        payload: qty
    }
}

 //Store
 const { configureStore } = require('@reduxjs/toolkit');
 const store = configureStore({reducer: reducer});

 console.log('Initial State : ', store.getState())

 store.dispatch(restockCake(15))
 console.log('Restock Cake ', store.getState())

 store.dispatch(buyCake())
 console.log('Buy Cake ', store.getState())

 store.dispatch(buyCake())
 console.log('Buy Cake ', store.getState())

 store.dispatch(restockCake(5))
 console.log('Restock Cake ', store.getState())
```

These are done is front end when we use react



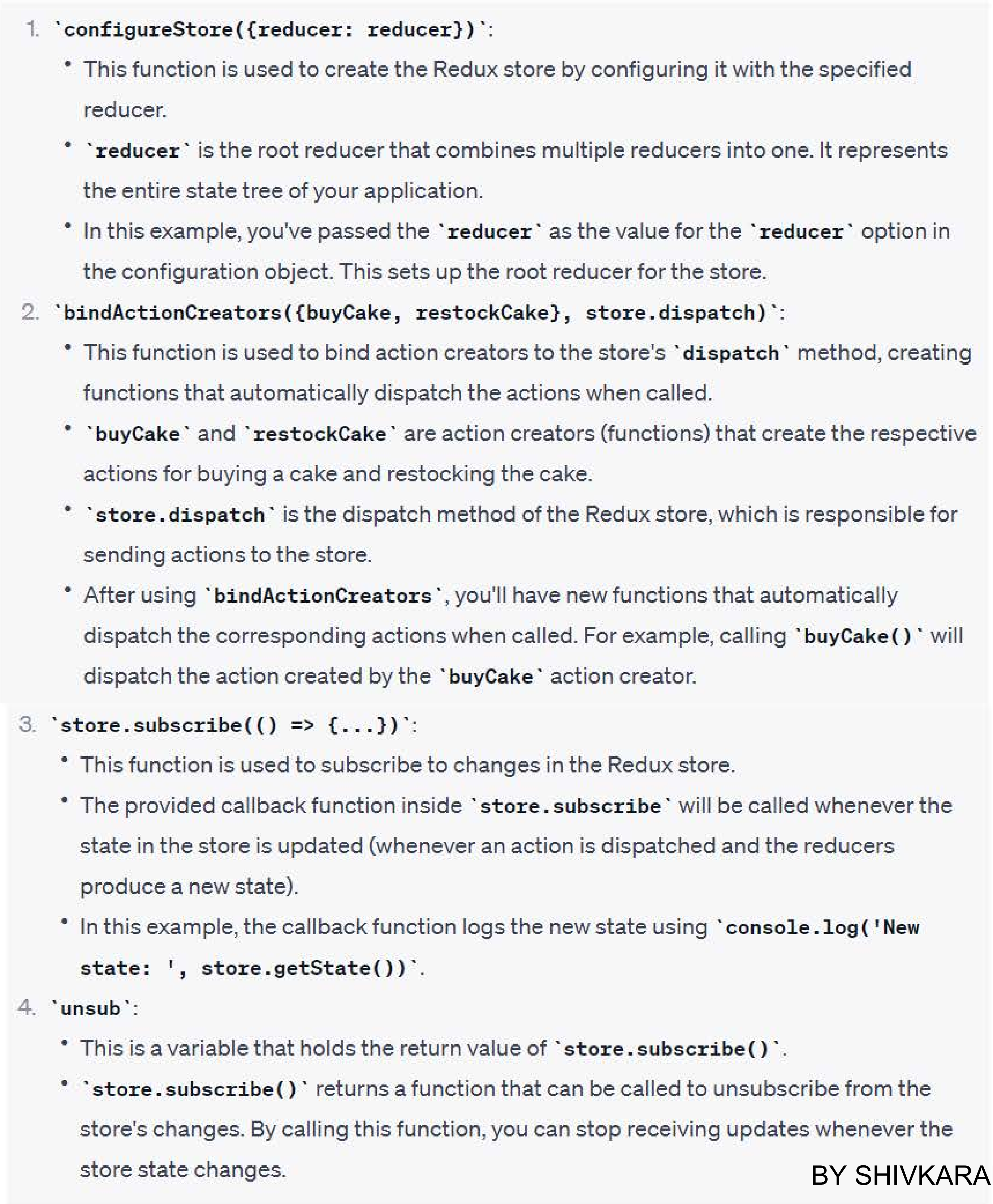
```
Initial State :  { numOfCakes: 0 }
Restock Cake  { numOfCakes: 15 }
Buy Cake  { numOfCakes: 14 }
Buy Cake  { numOfCakes: 13 }
Restock Cake  { numOfCakes: 18 }
```



```
//Store
const { configureStore, bindActionCreators } = require('@reduxjs/toolkit');

const store = configureStore({reducer: reducer})
const actions = bindActionCreators(
    {buyCake, restockCake},
    store.dispatch
)

const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})

console.log('Initial State', store.getState())
actions.buyCake()
actions.buyCake()
actions.restockCake(16)

unsub()
```

```
PS C:\Users\Shiva\OneDrive\Deskto
Initial State { numOfCakes: 0 }
New state:  { numOfCakes: -1 }
New state:  { numOfCakes: -2 }
New state:  { numOfCakes: 14 }
```

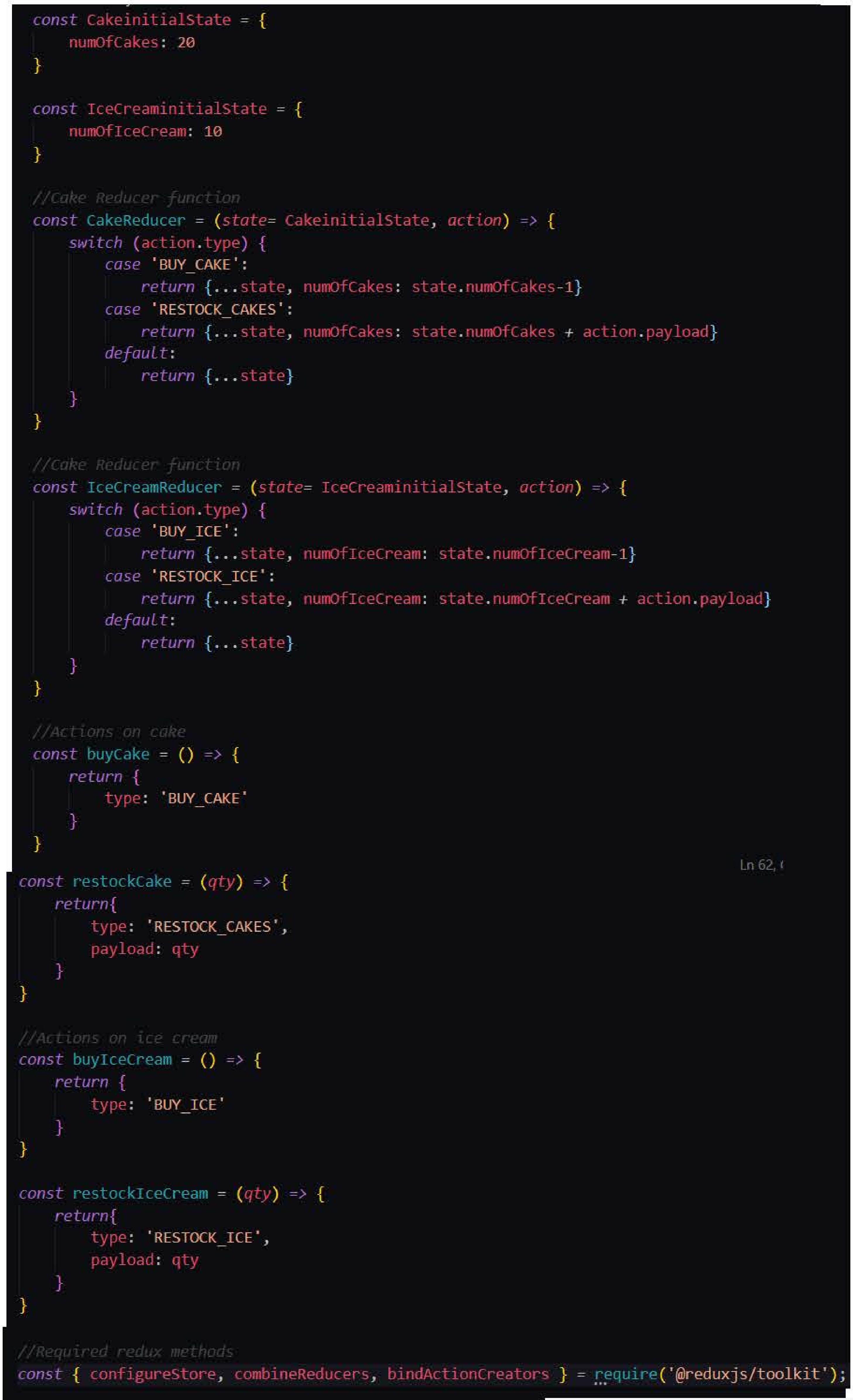
Better Approach



1. `configureStore({reducer: reducer})`:
   - This function is used to create the Redux store by configuring it with the specified reducer.
   - `reducer` is the root reducer that combines multiple reducers into one. It represents the entire state tree of your application.
   - In this example, you've passed the `reducer` as the value for the `reducer` option in the configuration object. This sets up the root reducer for the store.
2. `bindActionCreators({buyCake, restockCake}, store.dispatch)`:
   - This function is used to bind action creators to the store's `dispatch` method, creating functions that automatically dispatch the actions when called.
   - `buyCake` and `restockCake` are action creators (functions) that create the respective actions for buying a cake and restocking the cake.
   - `store.dispatch` is the dispatch method of the Redux store, which is responsible for sending actions to the store.
   - After using `bindActionCreators`, you'll have new functions that automatically dispatch the corresponding actions when called. For example, calling `buyCake()` will dispatch the action created by the `buyCake` action creator.
3. `store.subscribe(() => {...})`:
   - This function is used to subscribe to changes in the Redux store.
   - The provided callback function inside `store.subscribe` will be called whenever the state in the store is updated (whenever an action is dispatched and the reducers produce a new state).
   - In this example, the callback function logs the new state using `console.log('New state: ', store.getState())`.
4. `unsub`:
   - This is a variable that holds the return value of `store.subscribe()`.
   - `store.subscribe()` returns a function that can be called to unsubscribe from the store's changes. By calling this function, you can stop receiving updates whenever the store state changes.

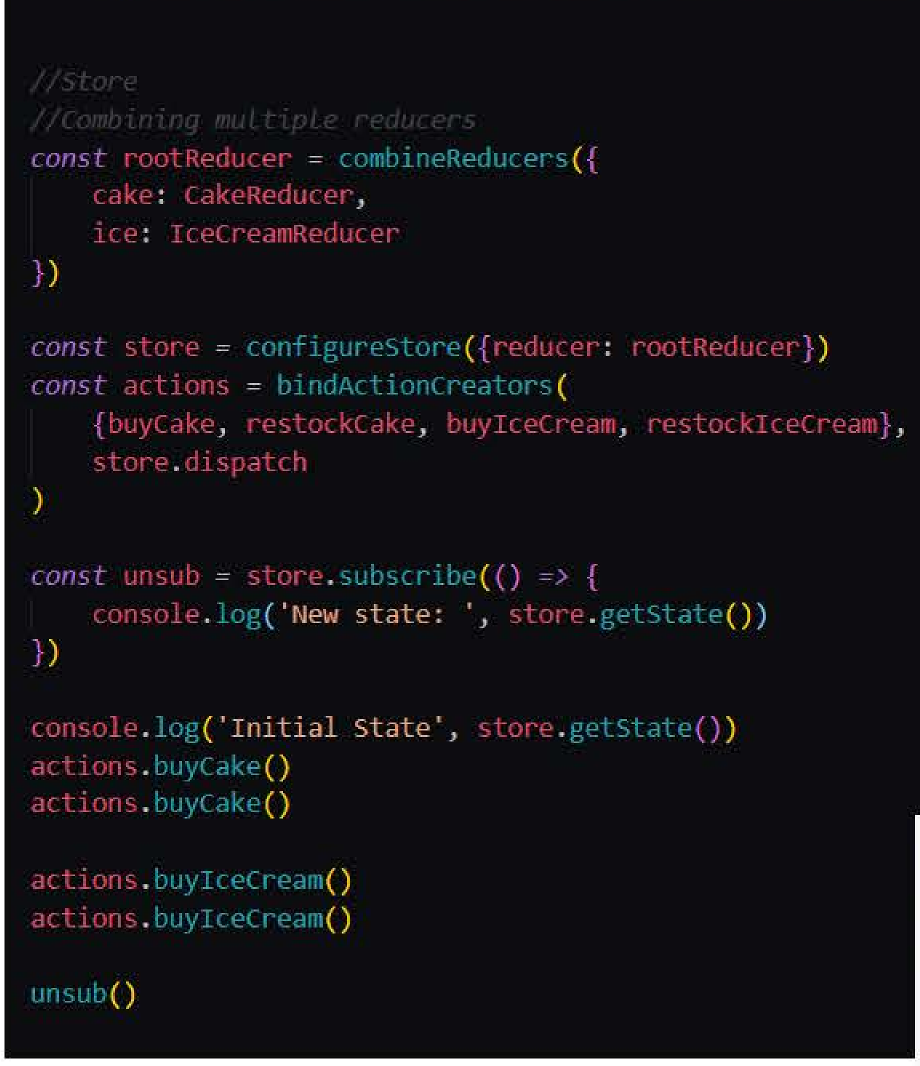
BY SHIVKARAMAN

Multiple Reducers

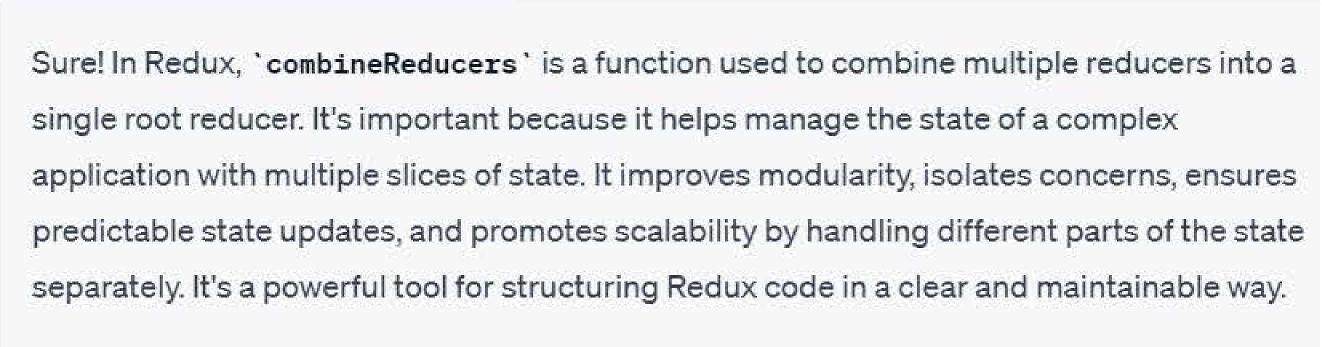


```
const CakeinitialState = {
    numOfCakes: 20
}

const IceCreaminitialState = {
    numOfIceCream: 10
}

//Cake Reducer function
const CakeReducer = (state= CakeinitialState, action) => {
    switch (action.type) {
        case 'BUY_CAKE':
            return {...state, numOfCakes: state.numOfCakes-1}
        case 'RESTOCK_CAKES':
            return {...state, numOfCakes: state.numOfCakes + action.payload}
        default:
            return {...state}
    }
}

//Cake Reducer function
const IceCreamReducer = (state= IceCreaminitialState, action) => {
    switch (action.type) {
        case 'BUY_ICE':
            return {...state, numOfIceCream: state.numOfIceCream-1}
        case 'RESTOCK_ICE':
            return {...state, numOfIceCream: state.numOfIceCream + action.payload}
        default:
            return {...state}
    }
}

//Actions on cake
const buyCake = () => {
    return {
        type: 'BUY_CAKE'
    }
}
```

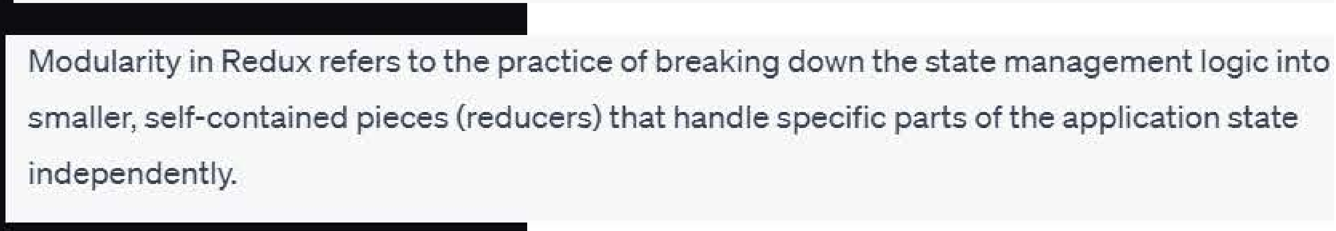
```
const restockCake = (qty) => {
    return{
        type: 'RESTOCK_CAKES',
        payload: qty
    }
}

//Actions on ice cream
const buyIceCream = () => {
    return {
        type: 'BUY_ICE'
    }
}

const restockIceCream = (qty) => {
    return{
        type: 'RESTOCK_ICE',
        payload: qty
    }
}

//Required redux methods
const { configureStore, combineReducers, bindActionCreators } = require('@reduxjs/toolkit');
```



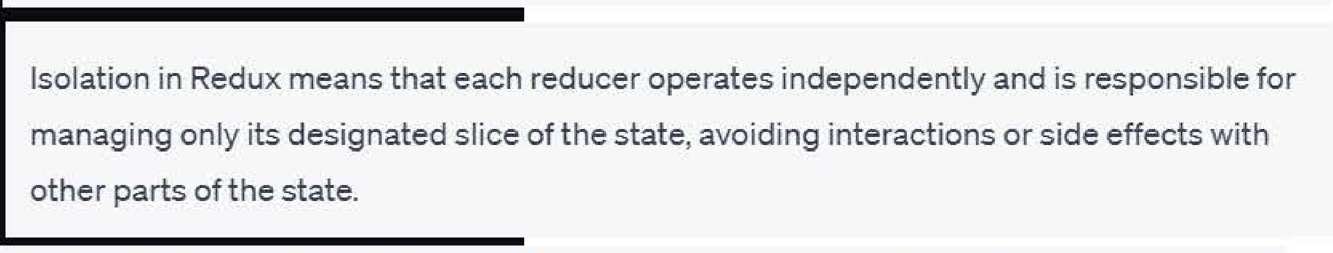
```
//Store
//Combining multiple reducers
const rootReducer = combineReducers({
    cake: CakeReducer,
    ice: IceCreamReducer
})

const store = configureStore({reducer: rootReducer})
const actions = bindActionCreators(
    {buyCake, restockCake, buyIceCream, restockIceCream},
    store.dispatch
)

const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})

console.log('Initial State', store.getState())
actions.buyCake()
actions.buyCake()

actions.buyIceCream()
actions.buyIceCream()

unsub()
```



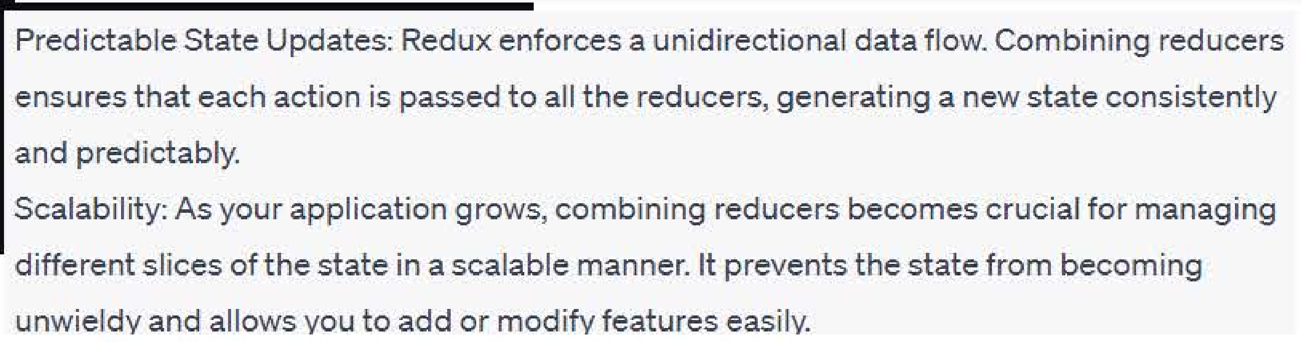
Sure! In Redux, `combineReducers` is a function used to combine multiple reducers into a single root reducer. It's important because it helps manage the state of a complex application with multiple slices of state. It improves modularity, isolates concerns, ensures predictable state updates, and promotes scalability by handling different parts of the state separately. It's a powerful tool for structuring Redux code in a clear and maintainable way.



Modularity in Redux refers to the practice of breaking down the state management logic into smaller, self-contained pieces (reducers) that handle specific parts of the application state independently.



Isolation in Redux means that each reducer operates independently and is responsible for managing only its designated slice of the state, avoiding interactions or side effects with other parts of the state.



Predictable State Updates: Redux enforces a unidirectional data flow. Combining reducers ensures that each action is passed to all the reducers, generating a new state consistently and predictably.

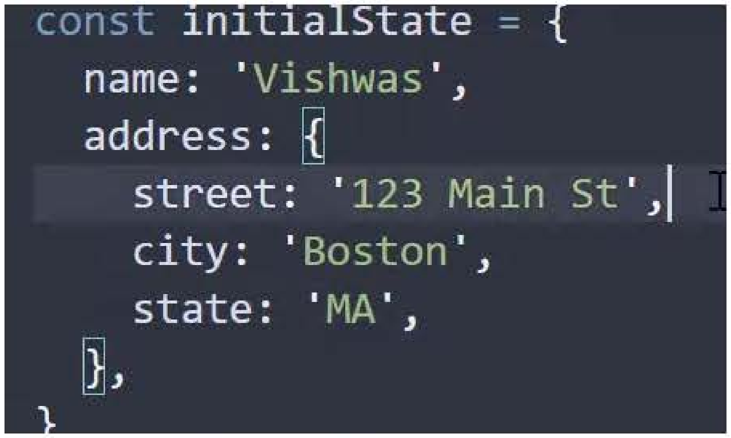
Scalability: As your application grows, combining reducers becomes crucial for managing different slices of the state in a scalable manner. It prevents the state from becoming unwieldy and allows you to add or modify features easily.

BY SHIVKARAMAN

Updating State in reducer Functions

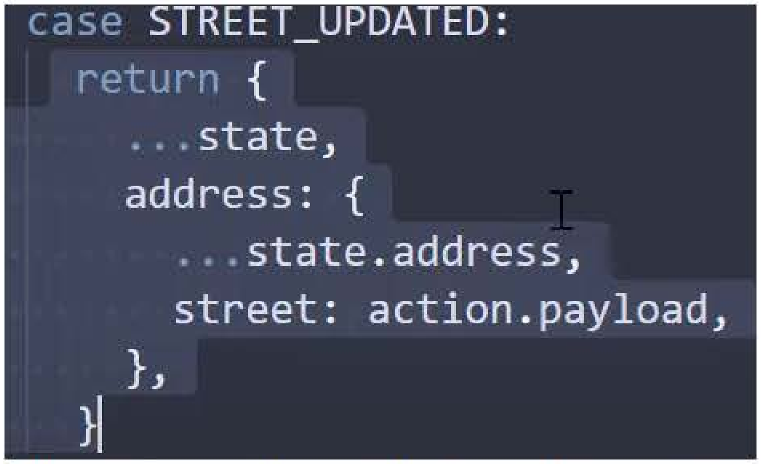
States should be treated as immutable in reducer functions. Reducer functions should never directly modify the state object that is passed to them. Instead, they should return a new state object that represents the updated state after processing an action.

State updates were simple till now, but as the complexity of the application grows, state update complexity grows as well.
Eg:



```
const initialState = {
  name: 'Vishwas',
  address: {
    street: '123 Main St',
    city: 'Boston',
    state: 'MA',
  },
}
```

Say this is initial state

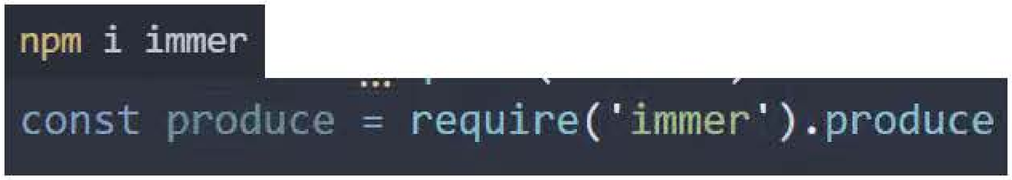


```
case STREET_UPDATED:
  return {
    ...state,
    address: {
      ...state.address,
      street: action.payload,
    },
  }
```

We can see here that updating street is not that simple.

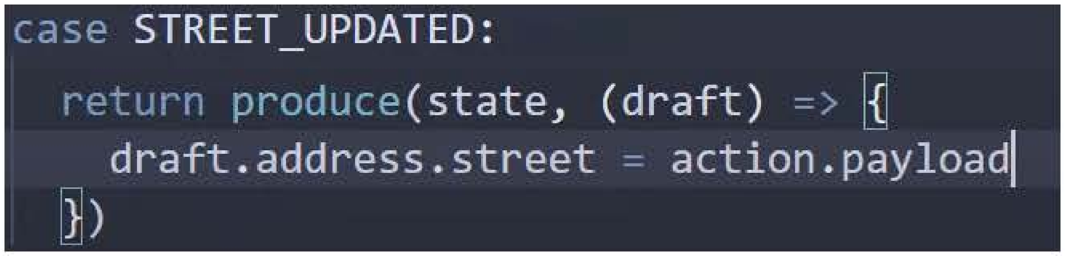
So updating state becomes more difficult as nesting increases

To make state updating simpler, we make use of a library called as **immer**



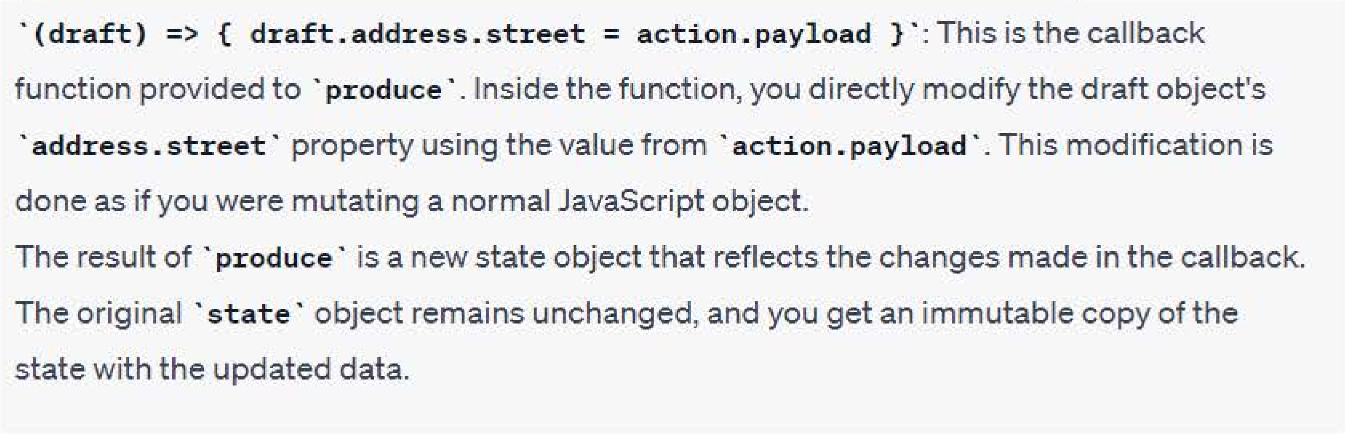
```
npm i immer
```

```
const produce = require('immer').produce
```



```
case STREET_UPDATED:
  return produce(state, (draft) => {
    draft.address.street = action.payload
  })
```

It takes the current state as the first argument and a callback function as the second argument. The callback function receives a "draft" version of the state, which can be modified directly as if it were mutable.



`(draft) => { draft.address.street = action.payload }`: This is the callback function provided to `produce`. Inside the function, you directly modify the draft object's `address.street` property using the value from `action.payload`. This modification is done as if you were mutating a normal JavaScript object.
The result of `produce` is a new state object that reflects the changes made in the callback. The original `state` object remains unchanged, and you get an immutable copy of the state with the updated data.

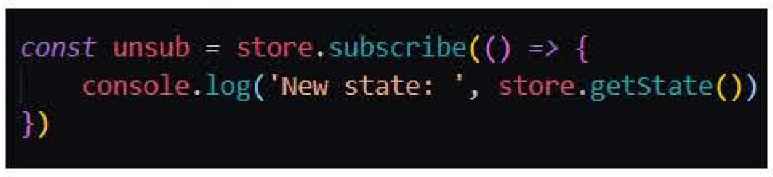
Middleware

Is the suggested way to extend Redux with custom functionality

Provides a third-party extension point between dispatching an action, and the moment it reaches the reducer

Use middleware for logging, crash reporting, performing asynchronous tasks etc

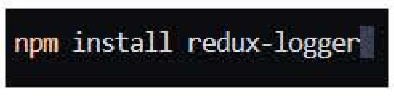
For example



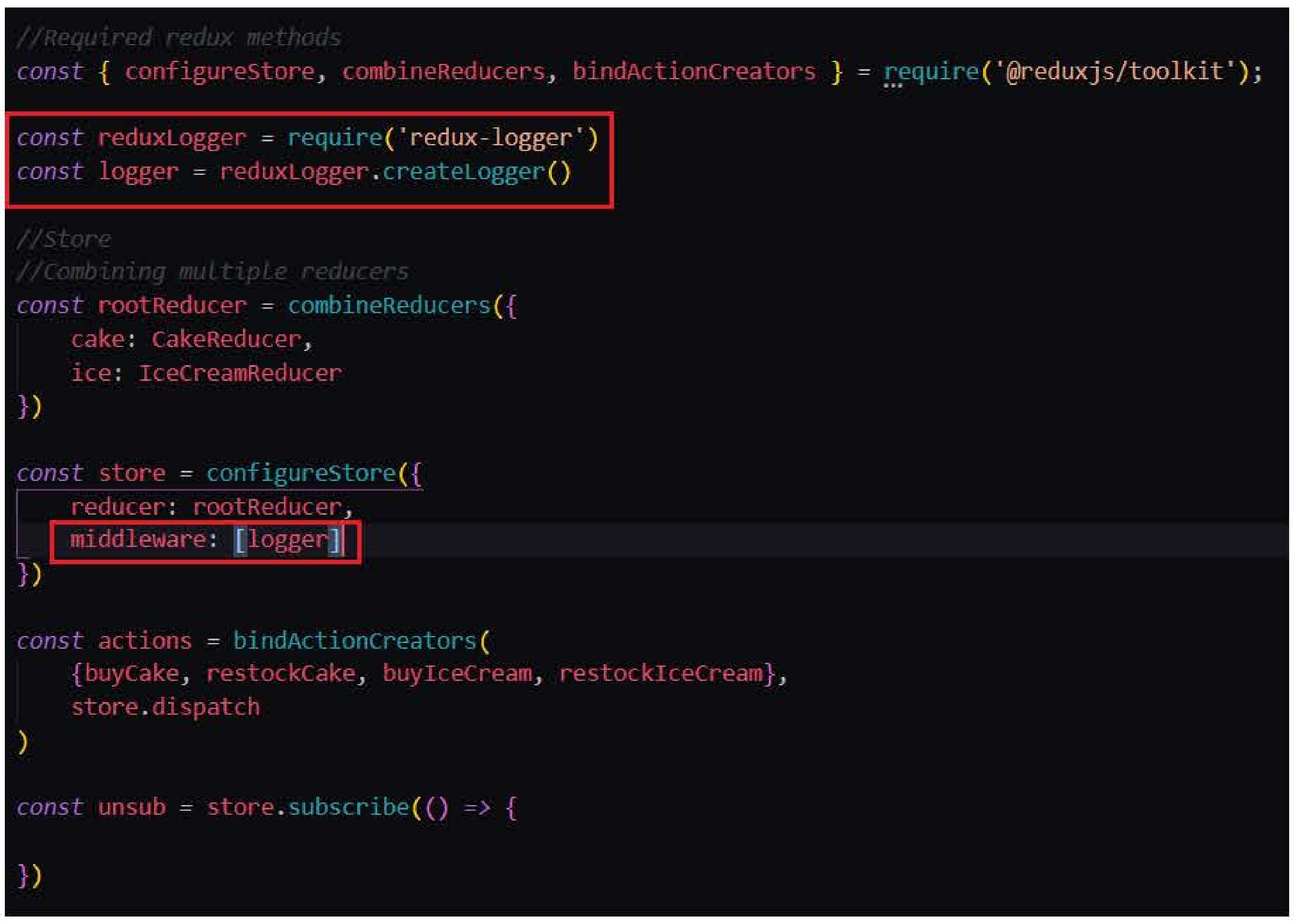
```
const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})
```

Here, we used to log the state of the data whenever there is an update in the data

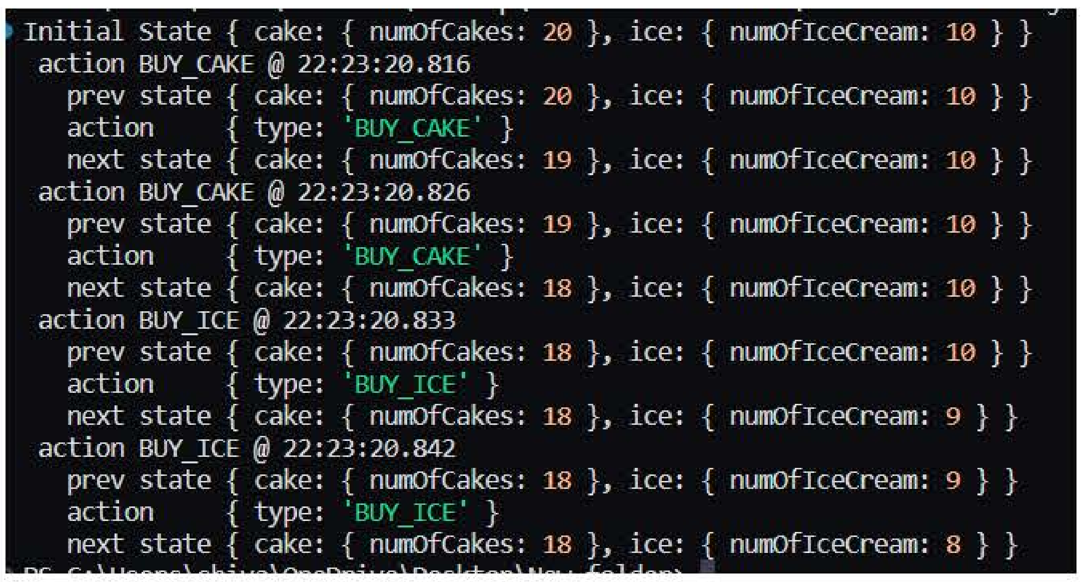
Instead of manually logging, we can use a library called as **redux-logger** and use it as a middleware



```
npm install redux-logger
```



```
//Required redux methods
const { configureStore, combineReducers, bindActionCreators } = require('@reduxjs/toolkit');

const reduxLogger = require('redux-logger')
const logger = reduxLogger.createLogger()

//Store
//Combining multiple reducers
const rootReducer = combineReducers({
    cake: CakeReducer,
    ice: IceCreamReducer
})

const store = configureStore({
    reducer: rootReducer,
    middleware: [logger]
})

const actions = bindActionCreators(
    {buyCake, restockCake, buyIceCream, restockIceCream},
    store.dispatch
)

const unsub = store.subscribe(() => {

})
```



```
Initial State { cake: { numOfCakes: 20 }, ice: { numOfIceCream: 10 } }
 action BUY_CAKE @ 22:23:20.816
   prev state { cake: { numOfCakes: 20 }, ice: { numOfIceCream: 10 } }
   action     { type: 'BUY_CAKE' }
   next state { cake: { numOfCakes: 19 }, ice: { numOfIceCream: 10 } }
 action BUY_CAKE @ 22:23:20.826
   prev state { cake: { numOfCakes: 19 }, ice: { numOfIceCream: 10 } }
   action     { type: 'BUY_CAKE' }
   next state { cake: { numOfCakes: 18 }, ice: { numOfIceCream: 10 } }
 action BUY_ICE @ 22:23:20.833
   prev state { cake: { numOfCakes: 18 }, ice: { numOfIceCream: 10 } }
   action     { type: 'BUY_ICE' }
   next state { cake: { numOfCakes: 18 }, ice: { numOfIceCream: 9 } }
 action BUY_ICE @ 22:23:20.842
   prev state { cake: { numOfCakes: 18 }, ice: { numOfIceCream: 9 } }
   action     { type: 'BUY_ICE' }
   next state { cake: { numOfCakes: 18 }, ice: { numOfIceCream: 8 } }
```

Actions

Synchronous Actions

As soon as an action was dispatched, the state was immediately updated.

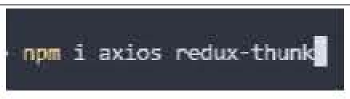
If you dispatch the CAKE_ORDERED action, the numOfCakes was right away decremented by 1.

Same with ICECREAM_ORDERED action as well.

Async Actions

Asynchronous API calls to fetch data from an end point and use that data in your application.

Async action creators



```
npm i axios redux-thunk
```

axios

Requests to an API end point : Axios is a standalone JavaScript library for making HTTP requests. Axios is used to fetch data asynchronously and redux is used to manage the data that is fetched

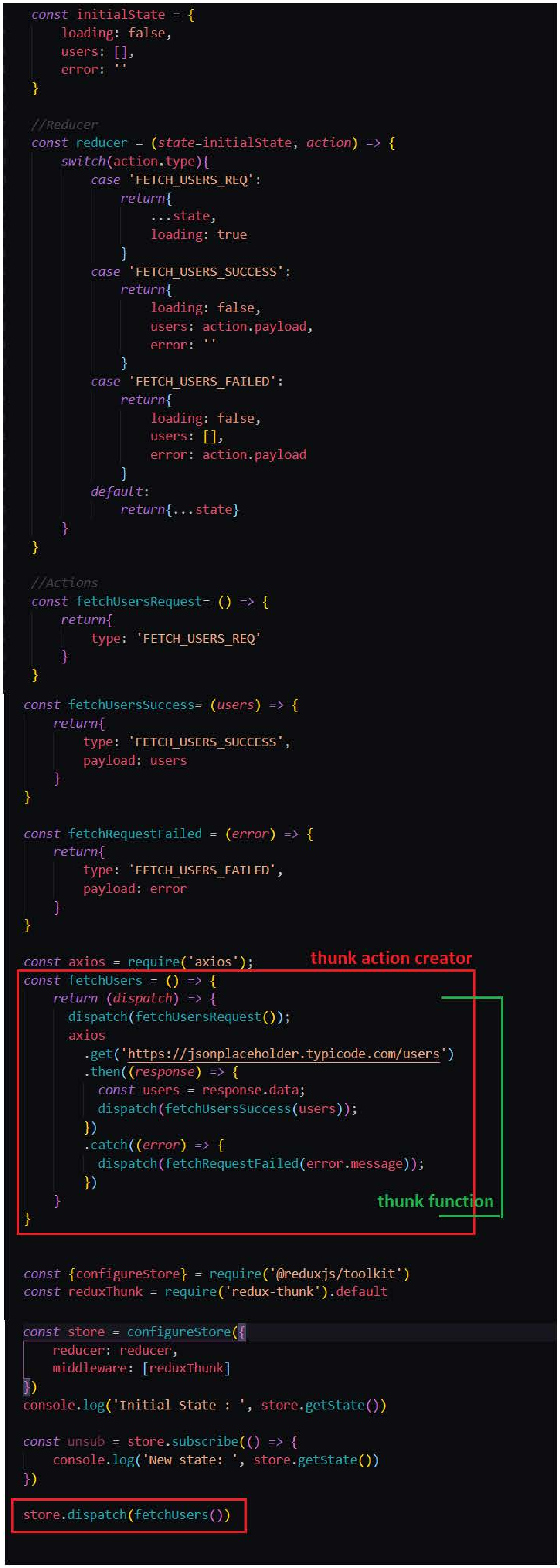
redux-thunk

Define async action creators : redux-thunk is a standard way to define asynchronous action creators in the application

Middleware

BY SHIVKARAMAN

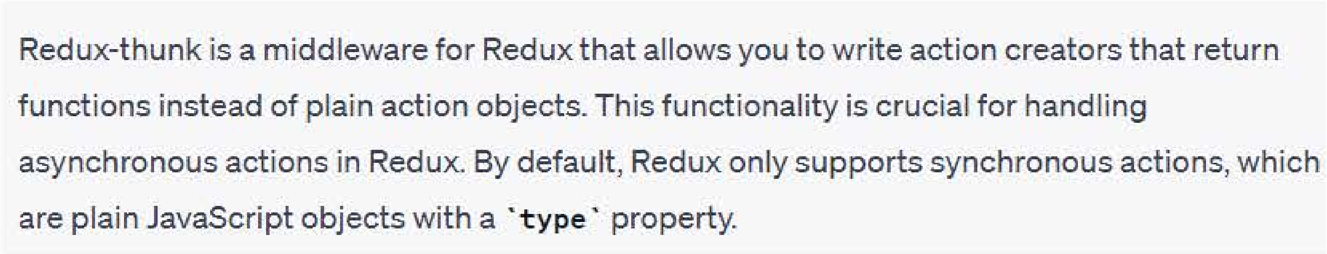
Asynchronous Action



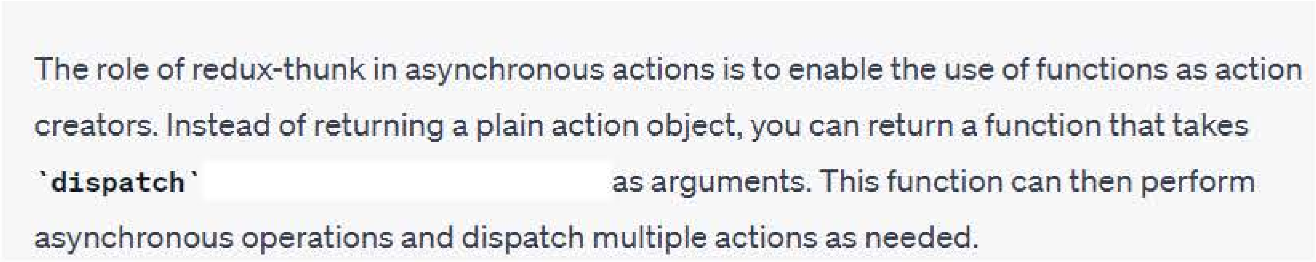
```
const initialState = {
    loading: false,
    users: [],
    error: ''
}

//Reducer
const reducer = (state=initialState, action) => {
    switch(action.type){
        case 'FETCH_USERS_REQ':
            return{
                ...state,
                loading: true
            }
        case 'FETCH_USERS_SUCCESS':
            return{
                loading: false,
                users: action.payload,
                error: ''
            }
        case 'FETCH_USERS_FAILED':
            return{
                loading: false,
                users: [],
                error: action.payload
            }
        default:
            return{...state}
    }
}

//Actions
const fetchUsersRequest= () => {
    return{
        type: 'FETCH_USERS_REQ'
    }
}
const fetchUsersSuccess= (users) => {
    return{
        type: 'FETCH_USERS_SUCCESS',
        payload: users
    }
}

const fetchRequestFailed = (error) => {
    return{
        type: 'FETCH_USERS_FAILED',
        payload: error
    }
}

const axios = require('axios');
const fetchUsers = () => {
    return (dispatch) => {
      dispatch(fetchUsersRequest());
      axios
        .get('https://jsonplaceholder.typicode.com/users')
        .then((response) => {
          const users = response.data;
          dispatch(fetchUsersSuccess(users));
        })
        .catch((error) => {
          dispatch(fetchRequestFailed(error.message));
        })
    }
}


const {configureStore} = require('@reduxjs/toolkit')
const reduxThunk = require('redux-thunk').default

const store = configureStore({
    reducer: reducer,
    middleware: [reduxThunk]
})
console.log('Initial State : ', store.getState())

const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})

store.dispatch(fetchUsers())
```

thunk action creator

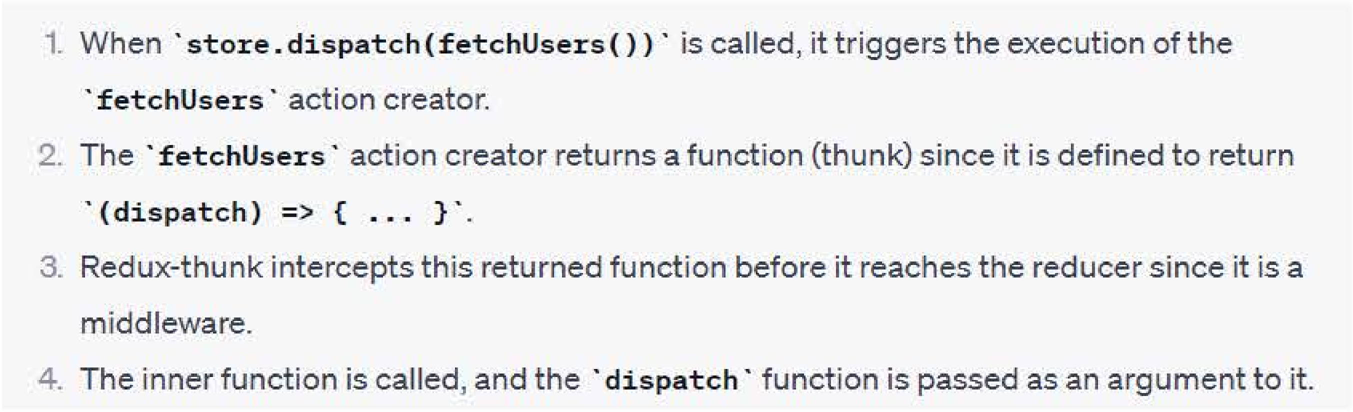
thunk function



Redux-thunk is a middleware for Redux that allows you to write action creators that return functions instead of plain action objects. This functionality is crucial for handling asynchronous actions in Redux. By default, Redux only supports synchronous actions, which are plain JavaScript objects with a `type` property.

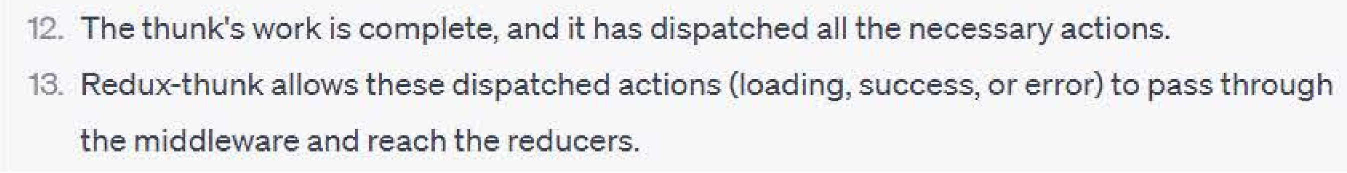


The role of redux-thunk in asynchronous actions is to enable the use of functions as action creators. Instead of returning a plain action object, you can return a function that takes `dispatch` as arguments. This function can then perform asynchronous operations and dispatch multiple actions as needed.



1. When `store.dispatch(fetchUsers())` is called, it triggers the execution of the `fetchUsers` action creator.
2. The `fetchUsers` action creator returns a function (thunk) since it is defined to return `(dispatch) => { ... }`.
3. Redux-thunk intercepts this returned function before it reaches the reducer since it is a middleware.
4. The inner function is called, and the `dispatch` function is passed as an argument to it.

5. Inside the thunk function, you can perform asynchronous tasks using axios, fetch, or other methods. During the asynchronous task, you can dispatch multiple actions, like a "loading" action to update the UI while waiting for the async operation.
Depending on the result of the async operation, you can dispatch "success" or "error" actions to handle the data or any encountered errors.

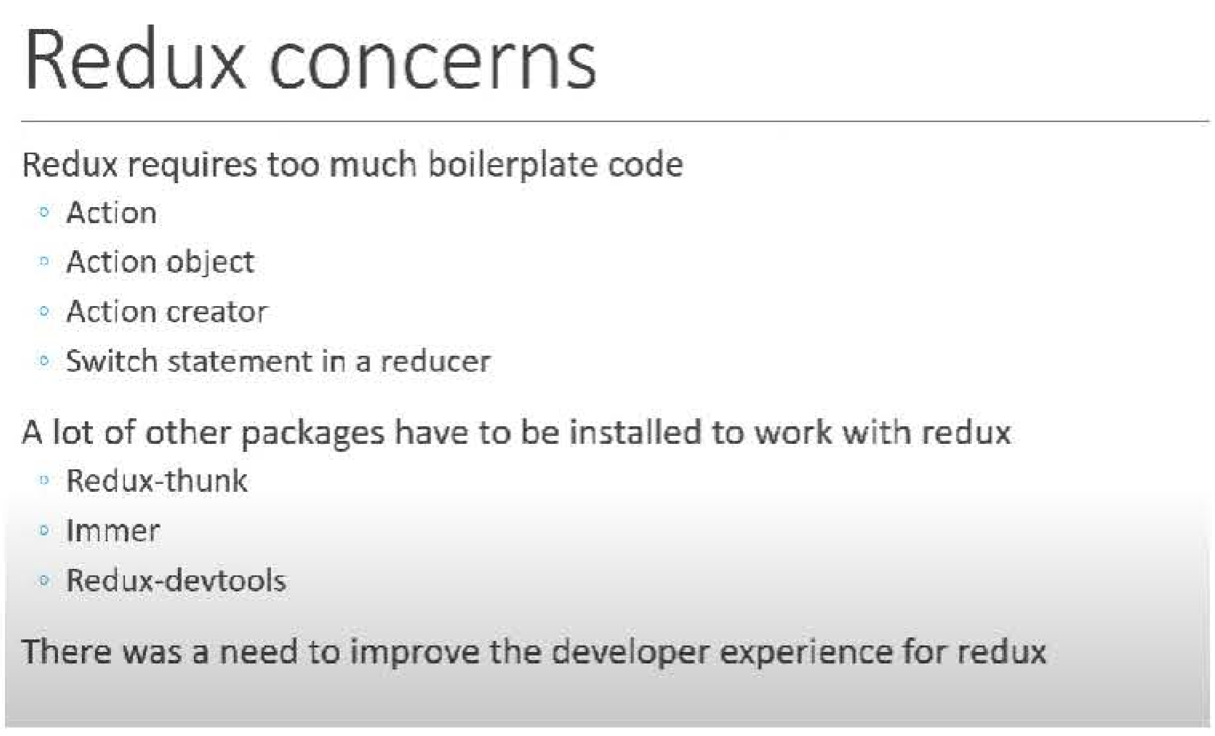


12. The thunk's work is complete, and it has dispatched all the necessary actions.
13. Redux-thunk allows these dispatched actions (loading, success, or error) to pass through the middleware and reach the reducers.

Here fetchUsers is an action creator.
Now what we know is that action creator retuens an object.
But redux-thunk allows an action creator to return a function.
The function need not be pure and is allowed to perform side effects like async api calls
It can perform dispatch actions because it recieves dispatch method as an arguement.

BY SHIVKARAMAN

Redux toolkit



Redux concerns

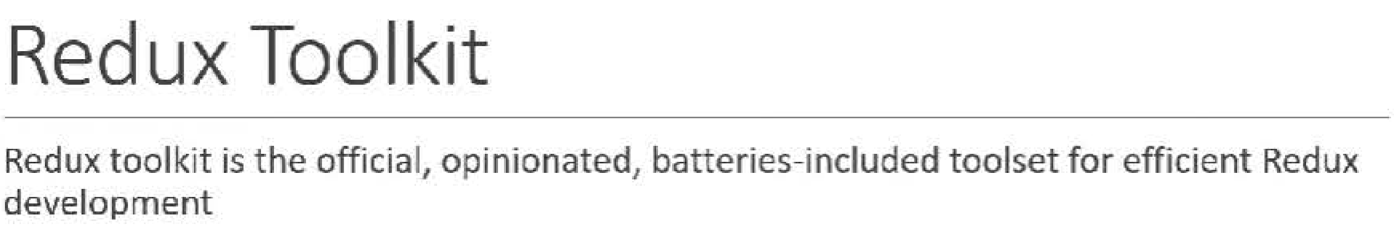
Redux requires too much boilerplate code

- Action
- Action object
- Action creator
- Switch statement in a reducer

A lot of other packages have to be installed to work with redux

- Redux-thunk
- Immer
- Redux-devtools

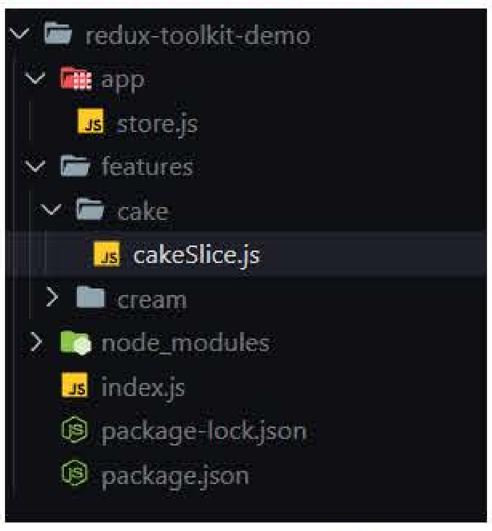
There was a need to improve the developer experience for redux



Redux Toolkit

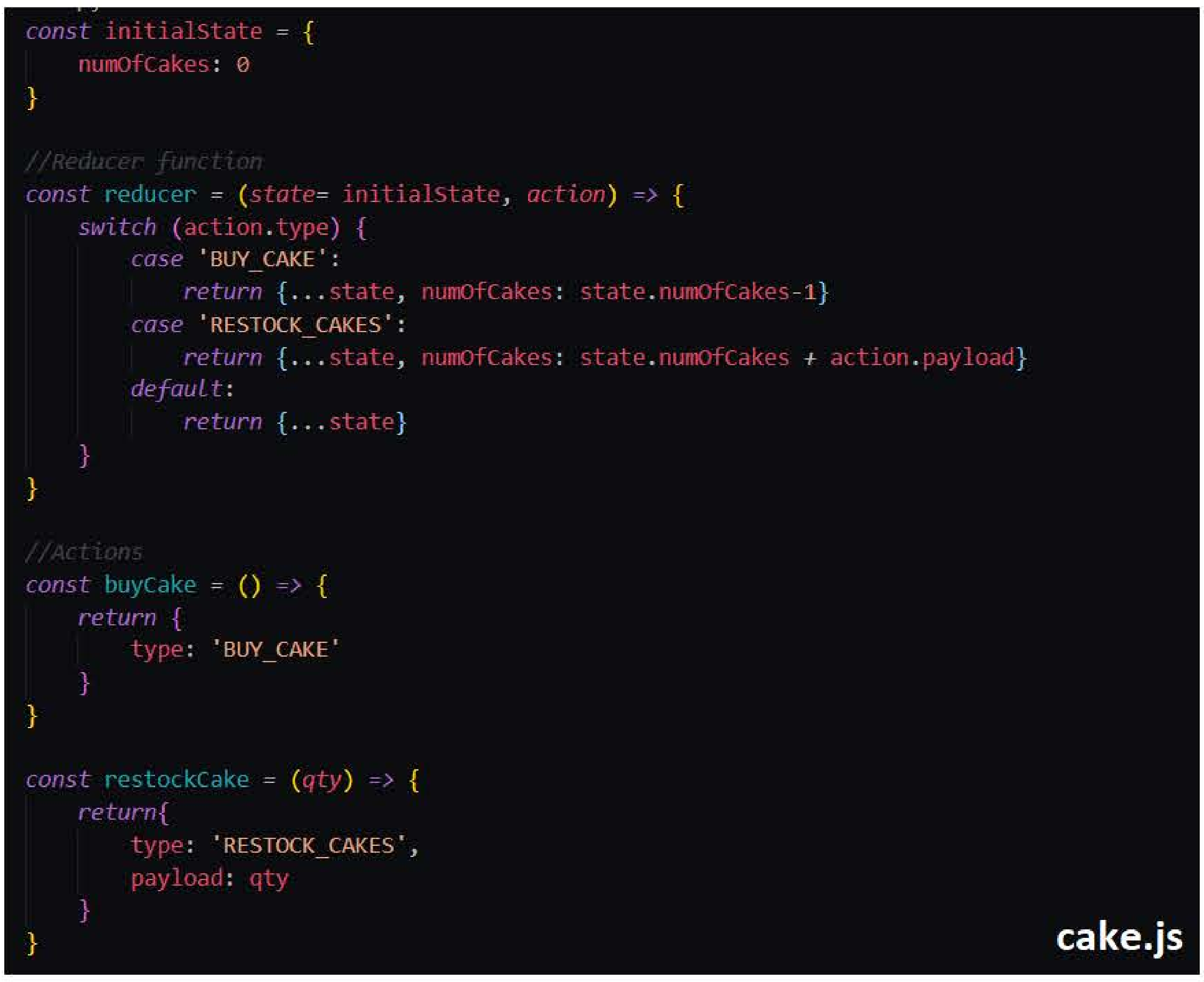
Redux toolkit is the official, opinionated, batteries-included toolset for efficient Redux development

The slice files shoule be present in the feature folder.



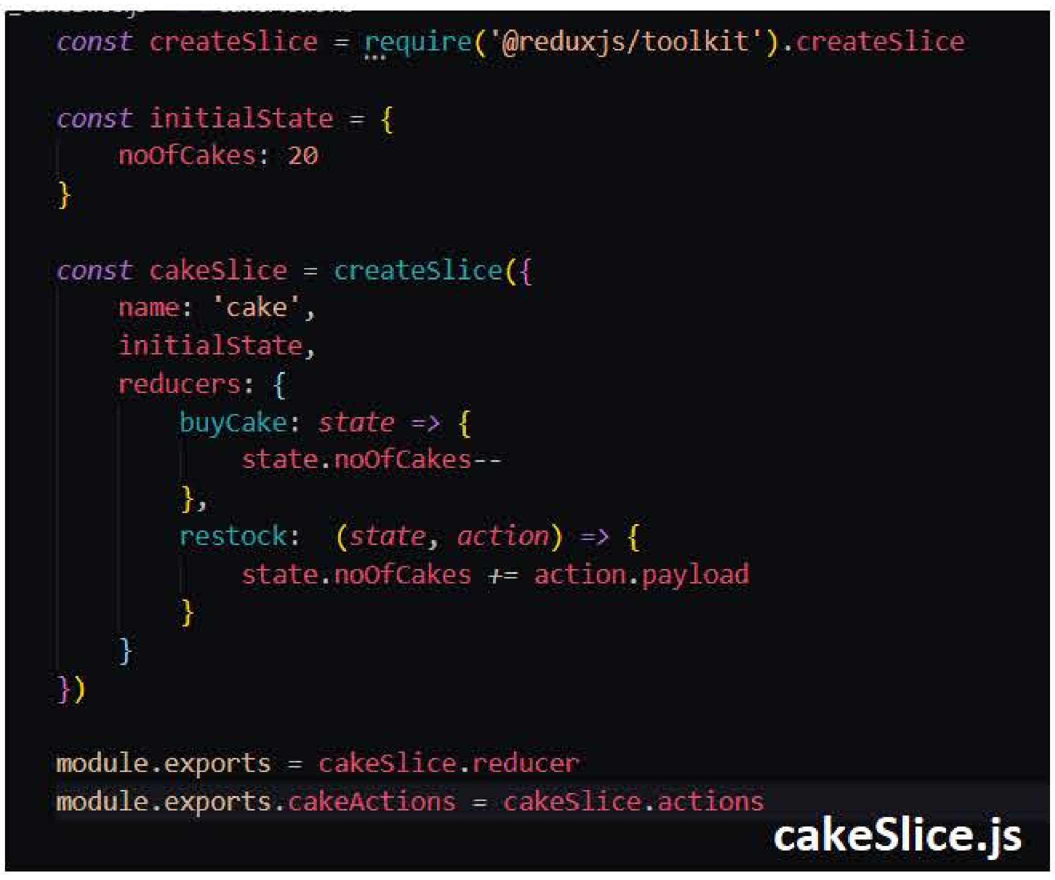
```
redux-toolkit-demo
  app
    store.js
  features
    cake
      cakeSlice.js
    cream
  node_modules
  index.js
  package-lock.json
  package.json
```

naming convention is ____Slice.js



```
const initialState = {
    numOfCakes: 0
}

//Reducer function
const reducer = (state= initialState, action) => {
    switch (action.type) {
        case 'BUY_CAKE':
            return {...state, numOfCakes: state.numOfCakes-1}
        case 'RESTOCK_CAKES':
            return {...state, numOfCakes: state.numOfCakes + action.payload}
        default:
            return {...state}
    }
}

//Actions
const buyCake = () => {
    return {
        type: 'BUY_CAKE'
    }
}

const restockCake = (qty) => {
    return{
        type: 'RESTOCK_CAKES',
        payload: qty
    }
}
```

cake.js



```
const createSlice = require('@reduxjs/toolkit').createSlice

const initialState = {
    noOfCakes: 20
}

const cakeSlice = createSlice({
    name: 'cake',
    initialState,
    reducers: {
        buyCake: state => {
            state.noOfCakes--
        },
        restock:  (state, action) => {
            state.noOfCakes += action.payload
        }
    }
})

module.exports = cakeSlice.reducer
module.exports.cakeActions = cakeSlice.actions
```

cakeSlice.js

module.exports: The cakeSlice.reducer is exported as the reducer function to be used when configuring the Redux store.

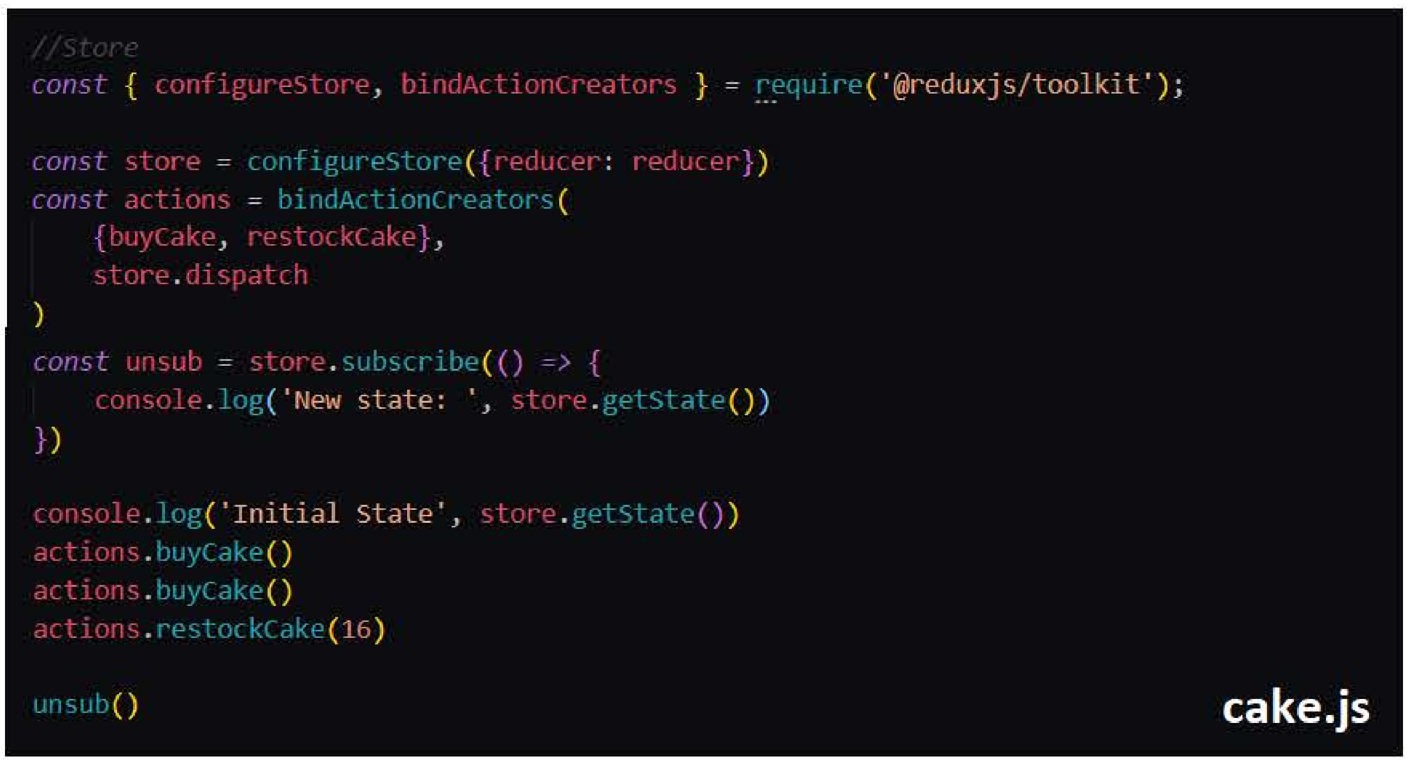
module.exports.cakeActions: The cakeSlice.actions object is exported to expose the action creators associated with this slice.

1. We may observe that we are directly upgrading the state here is noOfCakes--. createSlice internally uses the immer library which we saw earlier. Redux Toolkit automatically handles state updation by your behalf.

2.createSice automatically gererates action creaters as the same names of reducers

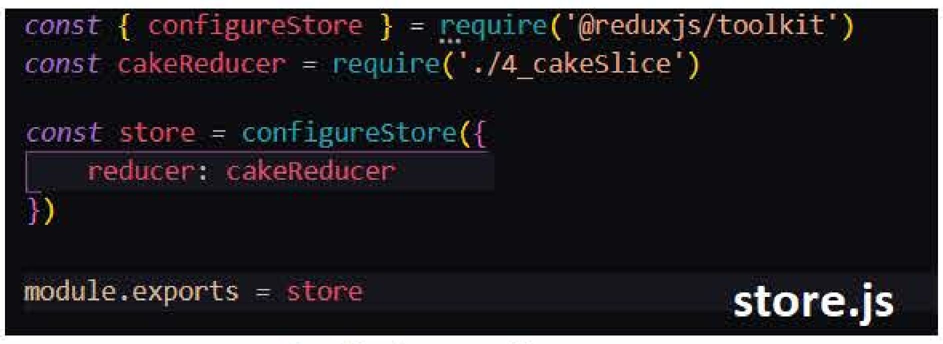
cakeSlice.actions -> This takes care of defining the action type constants, action creators

cakeSlice.reducer->This takes care of reducer function and handling immutable updates in the reducer. It uses the immer library behind the scenes



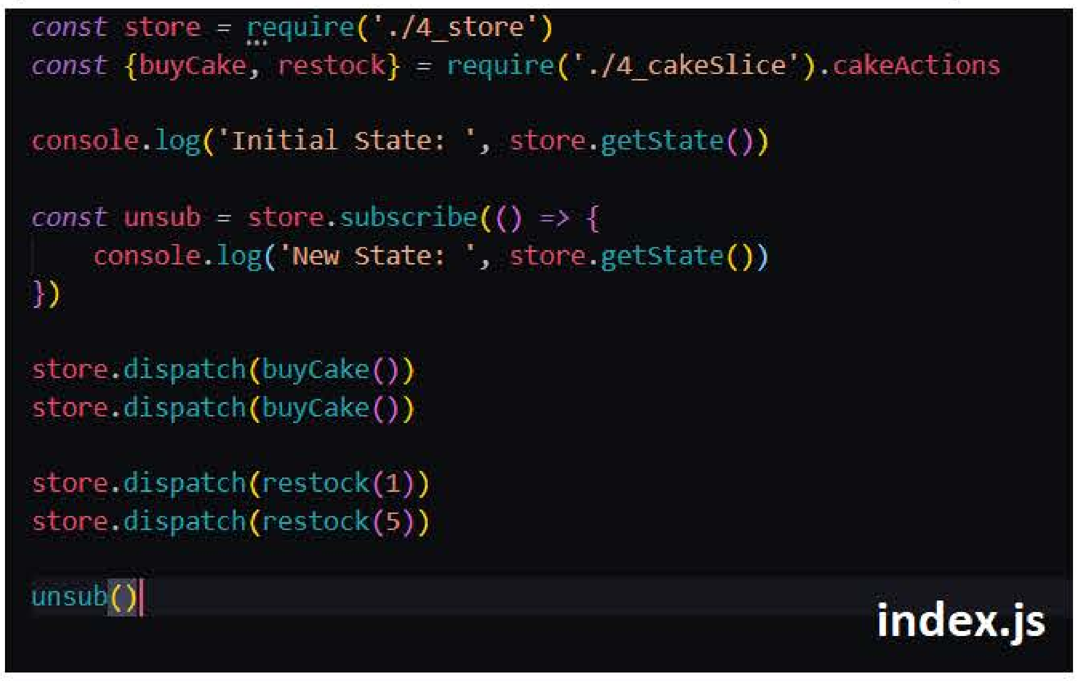
```
//Store
const { configureStore, bindActionCreators } = require('@reduxjs/toolkit');

const store = configureStore({reducer: reducer})
const actions = bindActionCreators(
    {buyCake, restockCake},
    store.dispatch
)
const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})

console.log('Initial State', store.getState())
actions.buyCake()
actions.buyCake()
actions.restockCake(16)

unsub()
```

cake.js



```
const { configureStore } = require('@reduxjs/toolkit')
const cakeReducer = require('./4_cakeSlice')

const store = configureStore({
    reducer: cakeReducer
})

module.exports = store
```

store.js

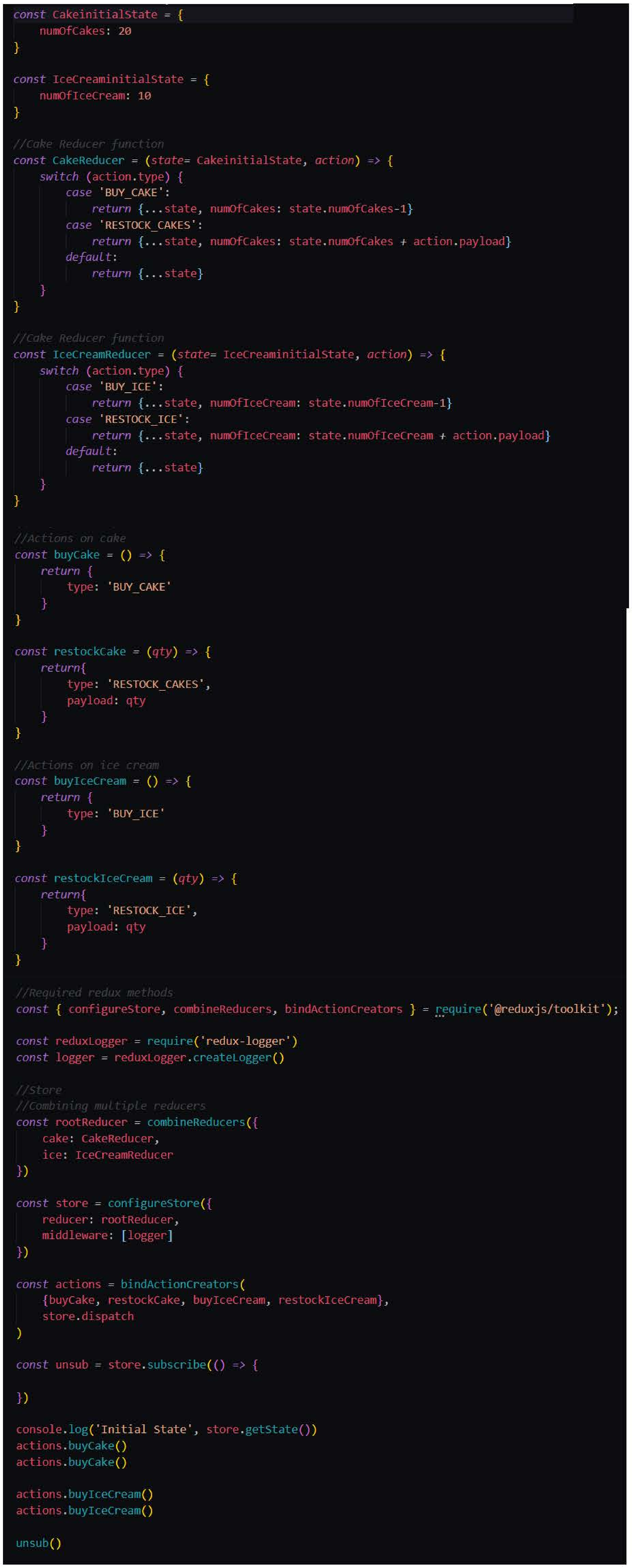


```
const store = require('./4_store')
const {buyCake, restock} = require('./4_cakeSlice').cakeActions

console.log('Initial State: ', store.getState())

const unsub = store.subscribe(() => {
    console.log('New State: ', store.getState())
})

store.dispatch(buyCake())
store.dispatch(buyCake())

store.dispatch(restock(1))
store.dispatch(restock(5))

unsub()
```

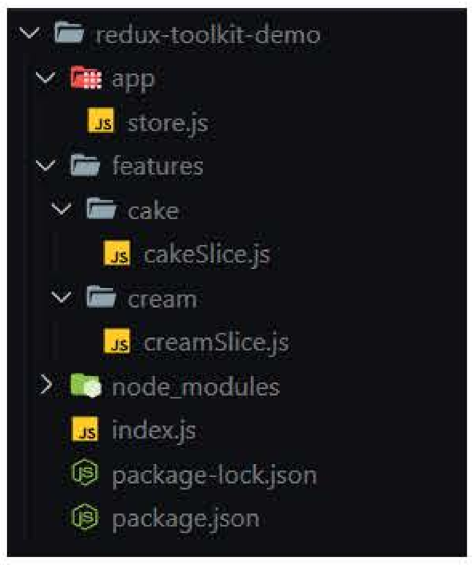
index.js

BY SHIVKARAMAN

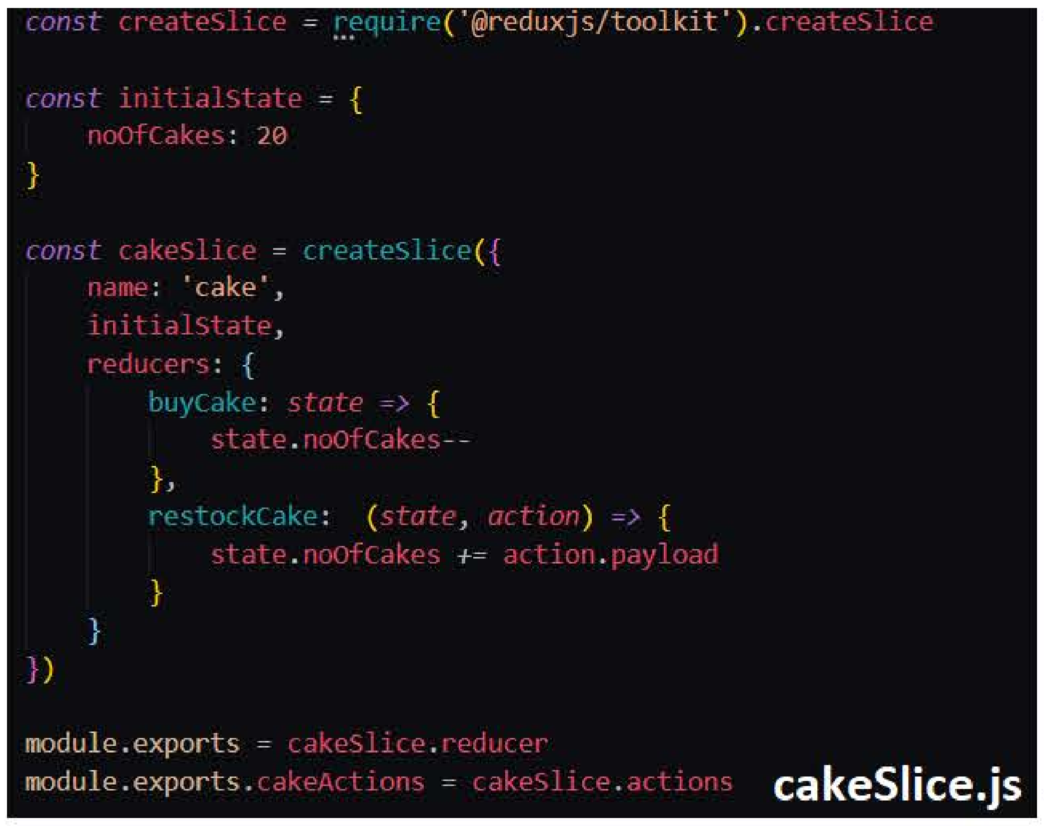
Multiple reducers



```
const CakeinitialState = {
    numOfCakes: 20
}

const IceCreaminitialState = {
    numOfIceCream: 10
}

//Cake Reducer function
const CakeReducer = (state= CakeinitialState, action) => {
    switch (action.type) {
        case 'BUY_CAKE':
            return {...state, numOfCakes: state.numOfCakes-1}
        case 'RESTOCK_CAKES':
            return {...state, numOfCakes: state.numOfCakes + action.payload}
        default:
            return {...state}
    }
}

//Cake Reducer function
const IceCreamReducer = (state= IceCreaminitialState, action) => {
    switch (action.type) {
        case 'BUY_ICE':
            return {...state, numOfIceCream: state.numOfIceCream-1}
        case 'RESTOCK_ICE':
            return {...state, numOfIceCream: state.numOfIceCream + action.payload}
        default:
            return {...state}
    }
}

//Actions on cake
const buyCake = () => {
    return {
        type: 'BUY_CAKE'
    }
}

const restockCake = (qty) => {
    return{
        type: 'RESTOCK_CAKES',
        payload: qty
    }
}

//Actions on ice cream
const buyIceCream = () => {
    return {
        type: 'BUY_ICE'
    }
}

const restockIceCream = (qty) => {
    return{
        type: 'RESTOCK_ICE',
        payload: qty
    }
}

//Required redux methods
const { configureStore, combineReducers, bindActionCreators } = require('@reduxjs/toolkit');

const reduxLogger = require('redux-logger')
const logger = reduxLogger.createLogger()

//Store
//Combining multiple reducers
const rootReducer = combineReducers({
    cake: CakeReducer,
    ice: IceCreamReducer
})

const store = configureStore({
    reducer: rootReducer,
    middleware: [logger]
})

const actions = bindActionCreators(
    {buyCake, restockCake, buyIceCream, restockIceCream},
    store.dispatch
)

const unsub = store.subscribe(() => {

})

console.log('Initial State', store.getState())
actions.buyCake()
actions.buyCake()

actions.buyIceCream()
actions.buyIceCream()

unsub()
```

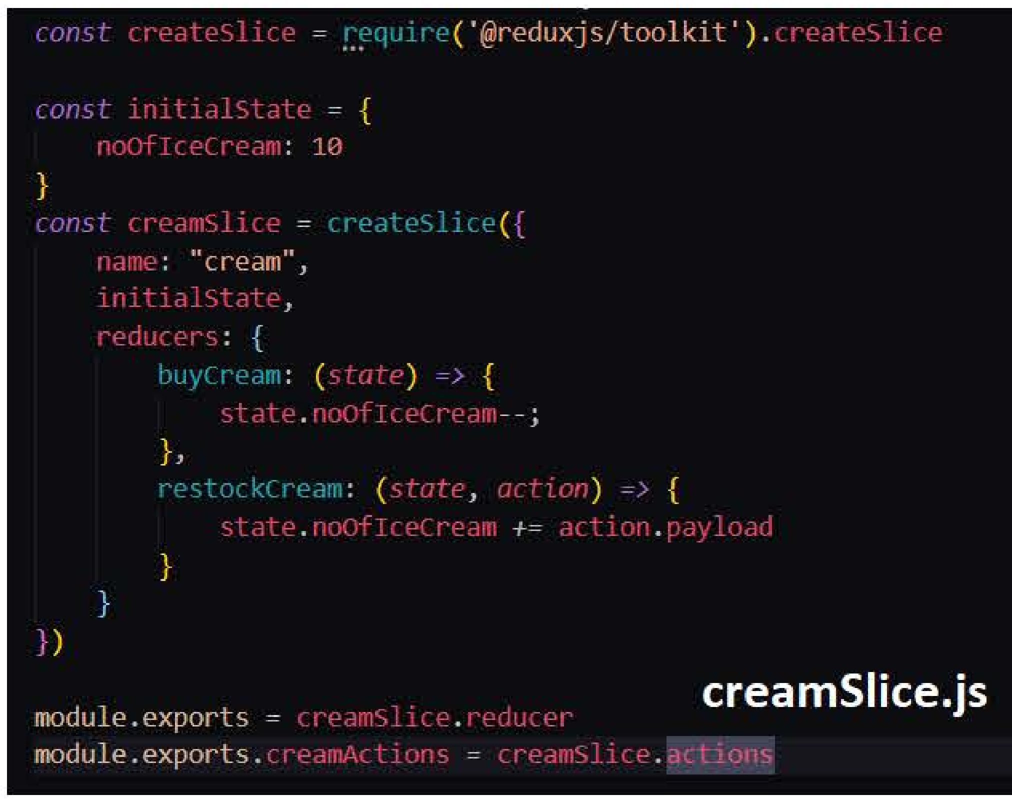


```
redux-toolkit-demo
  app
    store.js
  features
    cake
      cakeSlice.js
    cream
      creamSlice.js
  node_modules
  index.js
  package-lock.json
  package.json
```



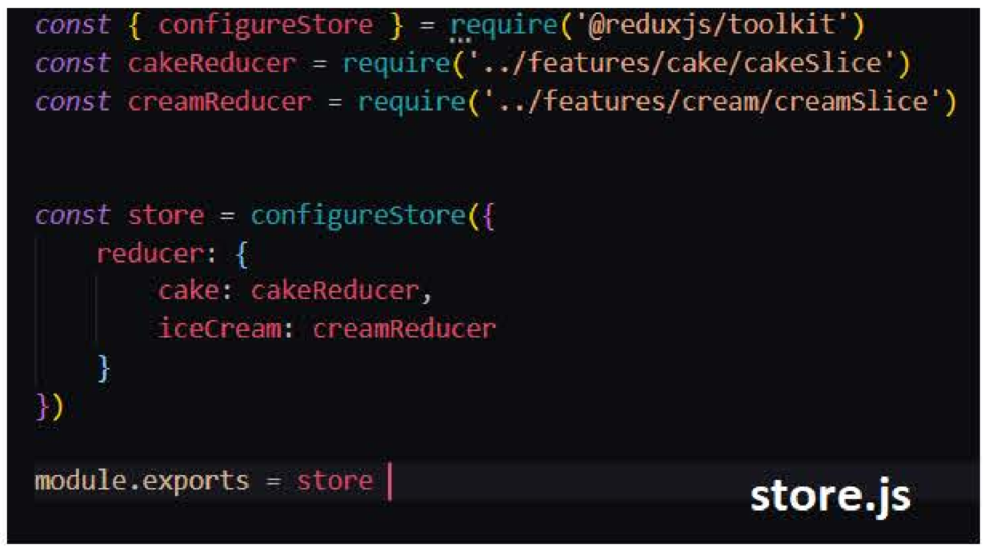
```
const createSlice = require('@reduxjs/toolkit').createSlice

const initialState = {
    noOfCakes: 20
}

const cakeSlice = createSlice({
    name: 'cake',
    initialState,
    reducers: {
        buyCake: state => {
            state.noOfCakes--
        },
        restockCake:  (state, action) => {
            state.noOfCakes += action.payload
        }
    }
})

module.exports = cakeSlice.reducer
module.exports.cakeActions = cakeSlice.actions
```

cakeSlice.js



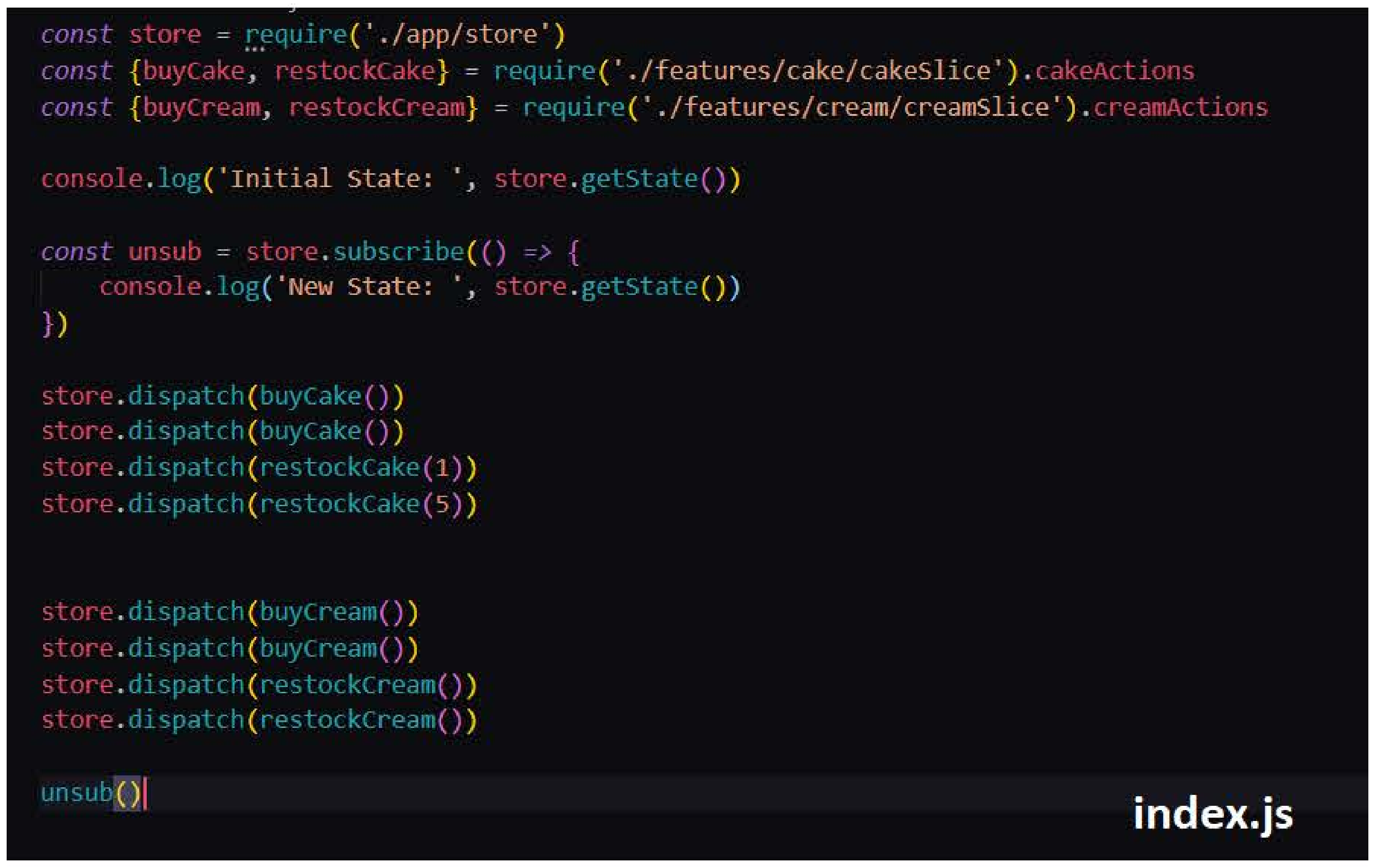
```
const createSlice = require('@reduxjs/toolkit').createSlice

const initialState = {
    noOfIceCream: 10
}
const creamSlice = createSlice({
    name: "cream",
    initialState,
    reducers: {
        buyCream: (state) => {
            state.noOfIceCream--;
        },
        restockCream: (state, action) => {
            state.noOfIceCream += action.payload
        }
    }
})

module.exports = creamSlice.reducer
module.exports.creamActions = creamSlice.actions
```

creamSlice.js



```
const { configureStore } = require('@reduxjs/toolkit')
const cakeReducer = require('../features/cake/cakeSlice')
const creamReducer = require('../features/cream/creamSlice')


const store = configureStore({
    reducer: {
        cake: cakeReducer,
        iceCream: creamReducer
    }
})

module.exports = store
```

store.js



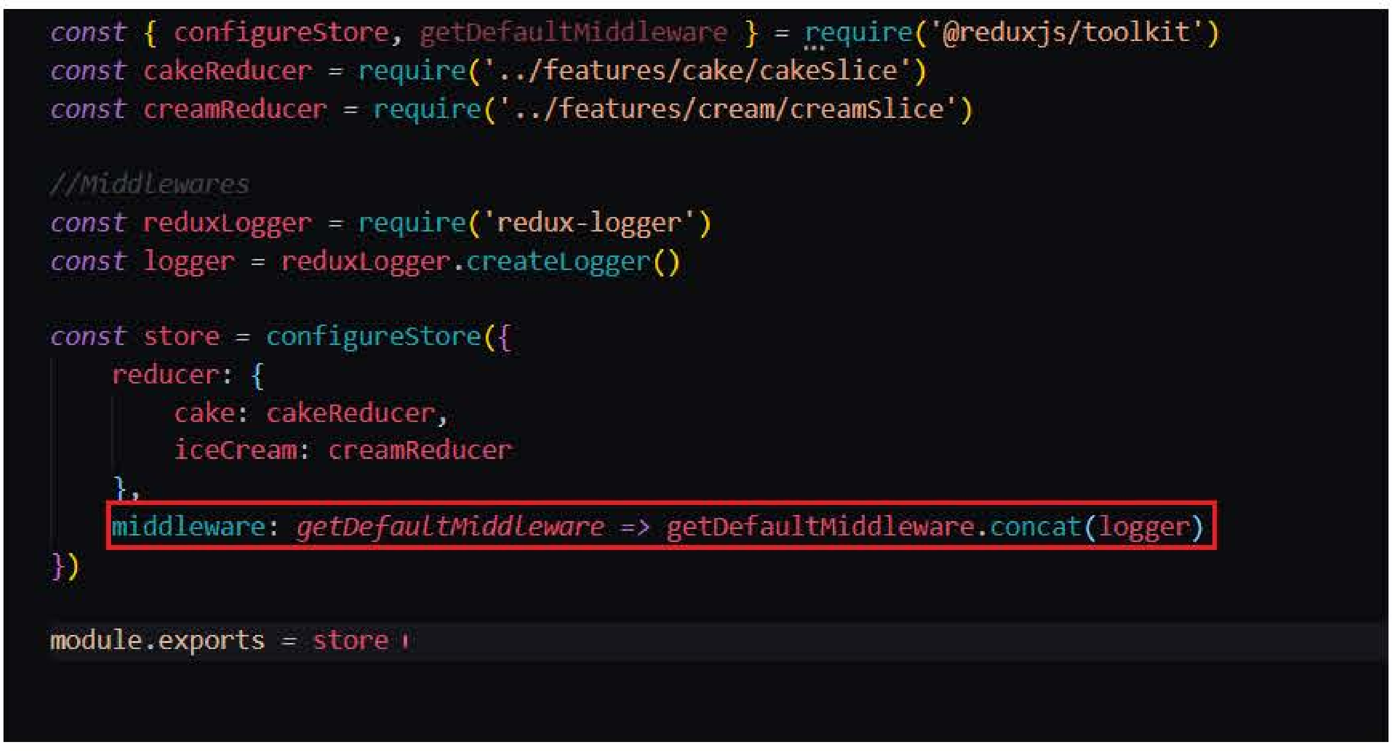
```
const store = require('./app/store')
const {buyCake, restockCake} = require('./features/cake/cakeSlice').cakeActions
const {buyCream, restockCream} = require('./features/cream/creamSlice').creamActions

console.log('Initial State: ', store.getState())

const unsub = store.subscribe(() => {
    console.log('New State: ', store.getState())
})

store.dispatch(buyCake())
store.dispatch(buyCake())
store.dispatch(restockCake(1))
store.dispatch(restockCake(5))


store.dispatch(buyCream())
store.dispatch(buyCream())
store.dispatch(restockCream())
store.dispatch(restockCream())

unsub()
```

index.js

We may observer that we used to frst combine reducers and keep it in rootReducer and then form the store. All of that is handled by the toolkit

BY SHIVKARAMAN

Using Middleware



```
const { configureStore, getDefaultMiddleware } = require('@reduxjs/toolkit')
const cakeReducer = require('../features/cake/cakeSlice')
const creamReducer = require('../features/cream/creamSlice')

//Middlewares
const reduxLogger = require('redux-logger')
const logger = reduxLogger.createLogger()

const store = configureStore({
    reducer: {
        cake: cakeReducer,
        iceCream: creamReducer
    },
    middleware: getDefaultMiddleware => getDefaultMiddleware.concat(logger)
})

module.exports = store
```

The reason why we are using getDefaultMiddleware is because the configureStore by default adds some middlewares. So we are just appending out middle ware to that list

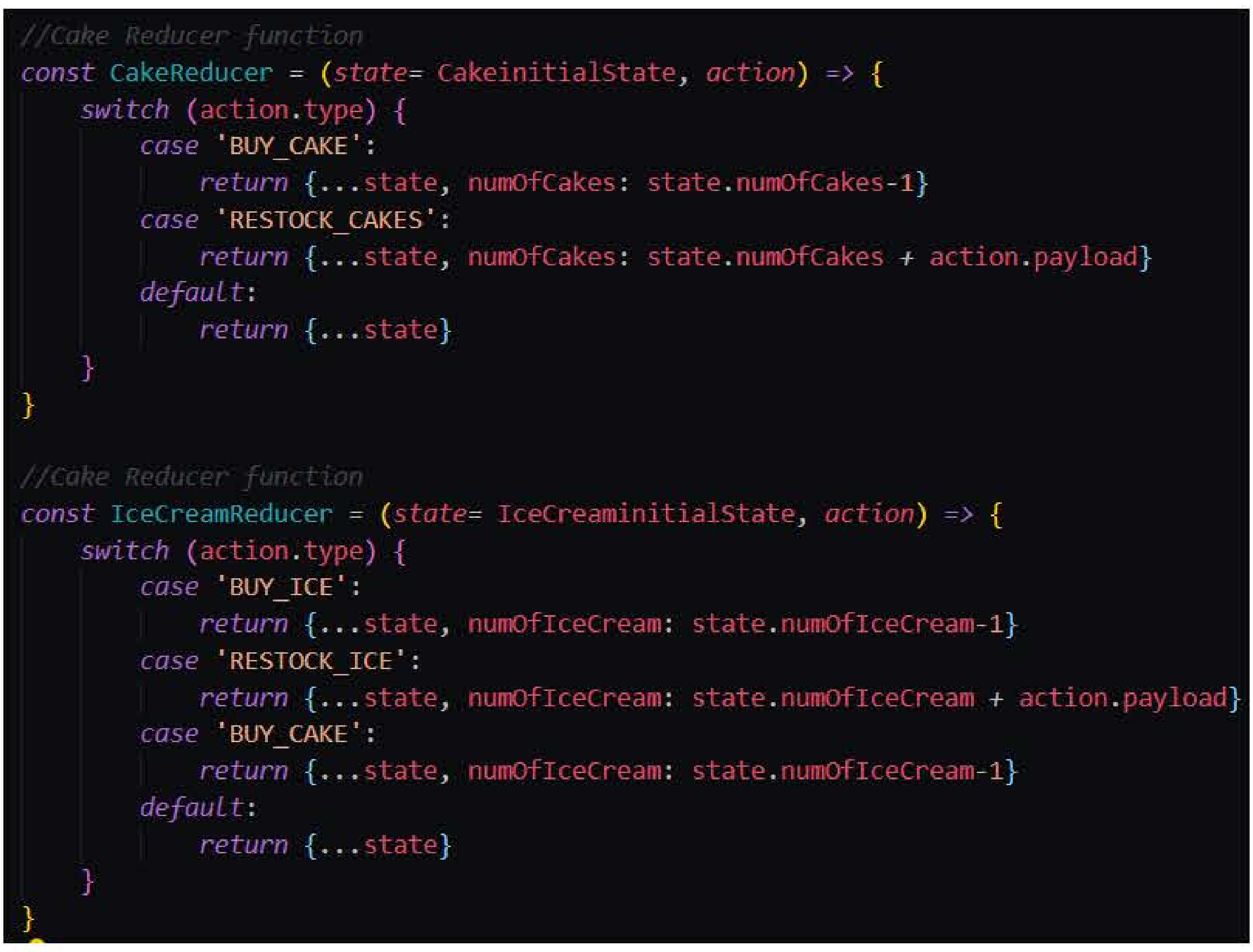
Extra Reducers

Consider the cakeAndIcecream reducer app. Here we had two different reducers and each were handling a different state.

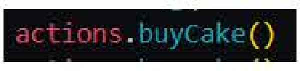
EVEN THOUGH WE HAVE SEPARATE REDUCERS, WHEN WE DISPATCH AN ACTION, BOTH THE REDUCERS RECIEVE THAT ACTION. ONE OF THEM ACTS ON THAT ACTION AND THE OTHER JUST DISCARDS IT.

Demonstration: Way every person who buys a cake gets an ice cream for free.

1. We dan dispatch buyIceCream everytime a cake is baught OR
2. We use the above property: What we can do is we can include a case BUY_CAKE in iceCreamReducer and decrement the no of iceCream



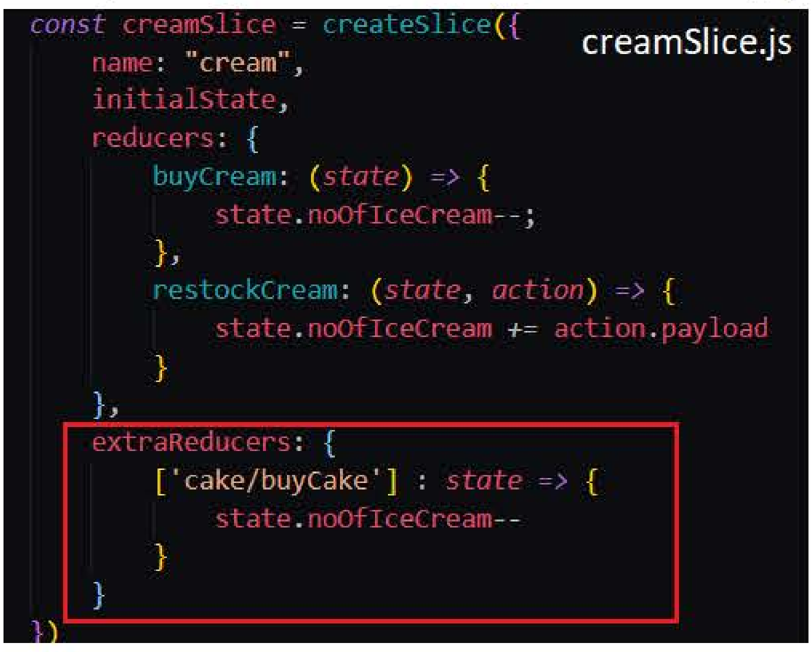
```
//Cake Reducer function
const CakeReducer = (state= CakeinitialState, action) => {
    switch (action.type) {
        case 'BUY_CAKE':
            return {...state, numOfCakes: state.numOfCakes-1}
        case 'RESTOCK_CAKES':
            return {...state, numOfCakes: state.numOfCakes + action.payload}
        default:
            return {...state}
    }
}

//Cake Reducer function
const IceCreamReducer = (state= IceCreaminitialState, action) => {
    switch (action.type) {
        case 'BUY_ICE':
            return {...state, numOfIceCream: state.numOfIceCream-1}
        case 'RESTOCK_ICE':
            return {...state, numOfIceCream: state.numOfIceCream + action.payload}
        case 'BUY_CAKE':
            return {...state, numOfIceCream: state.numOfIceCream-1}
        default:
            return {...state}
    }
}
```



`actions.buyCake()` So now everytime buy cake action is dispatched, both the cakeRecucer and iceCreamRecuder recieves the action. cakeReducer decrements noOfCakes by 1 and iceCreanReducer decrements noOfIceCream by 1

Conclusion: Each reducer can only update its own state but it can respond to any action dispatched in the application.

The above is true in case of **redux** only. But when you use redux-toolkit, the above property does not hold true. A reducer only responds to its actions. So to impliment the above functionality you need to make use of extra reducers

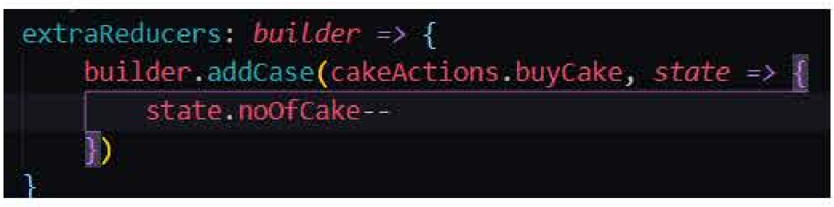


creamSlice.js

```
const creamSlice = createSlice({
    name: "cream",
    initialState,
    reducers: {
        buyCream: (state) => {
            state.noOfIceCream--;
        },
        restockCream: (state, action) => {
            state.noOfIceCream += action.payload
        }
    },
    extraReducers: {
        ['cake/buyCake'] : state => {
            state.noOfIceCream--
        }
    }
})
```

['cake/buyCake'] -> this means buyCake action of the cake slice.

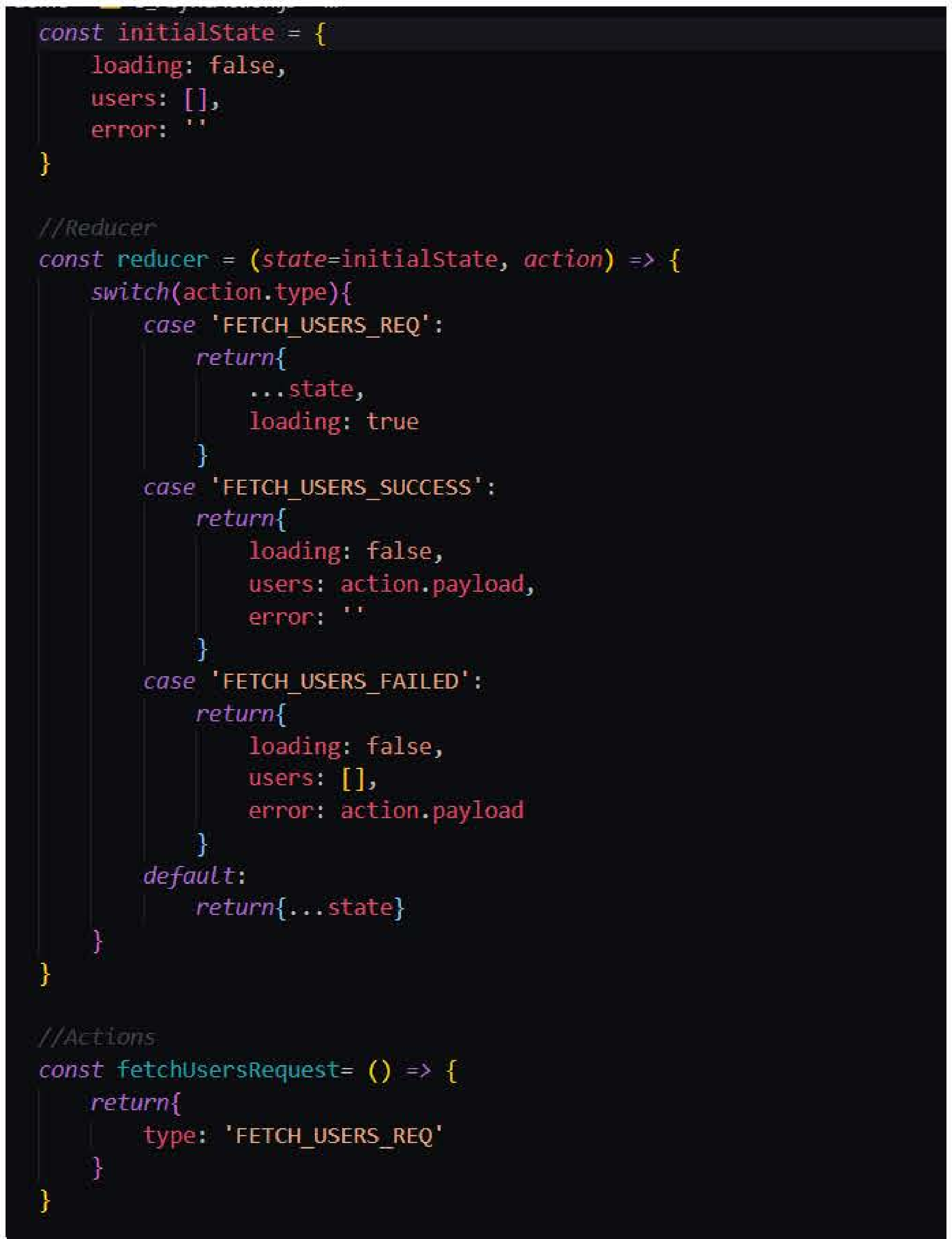
An alternative approach-> here extraRecucer is a function.



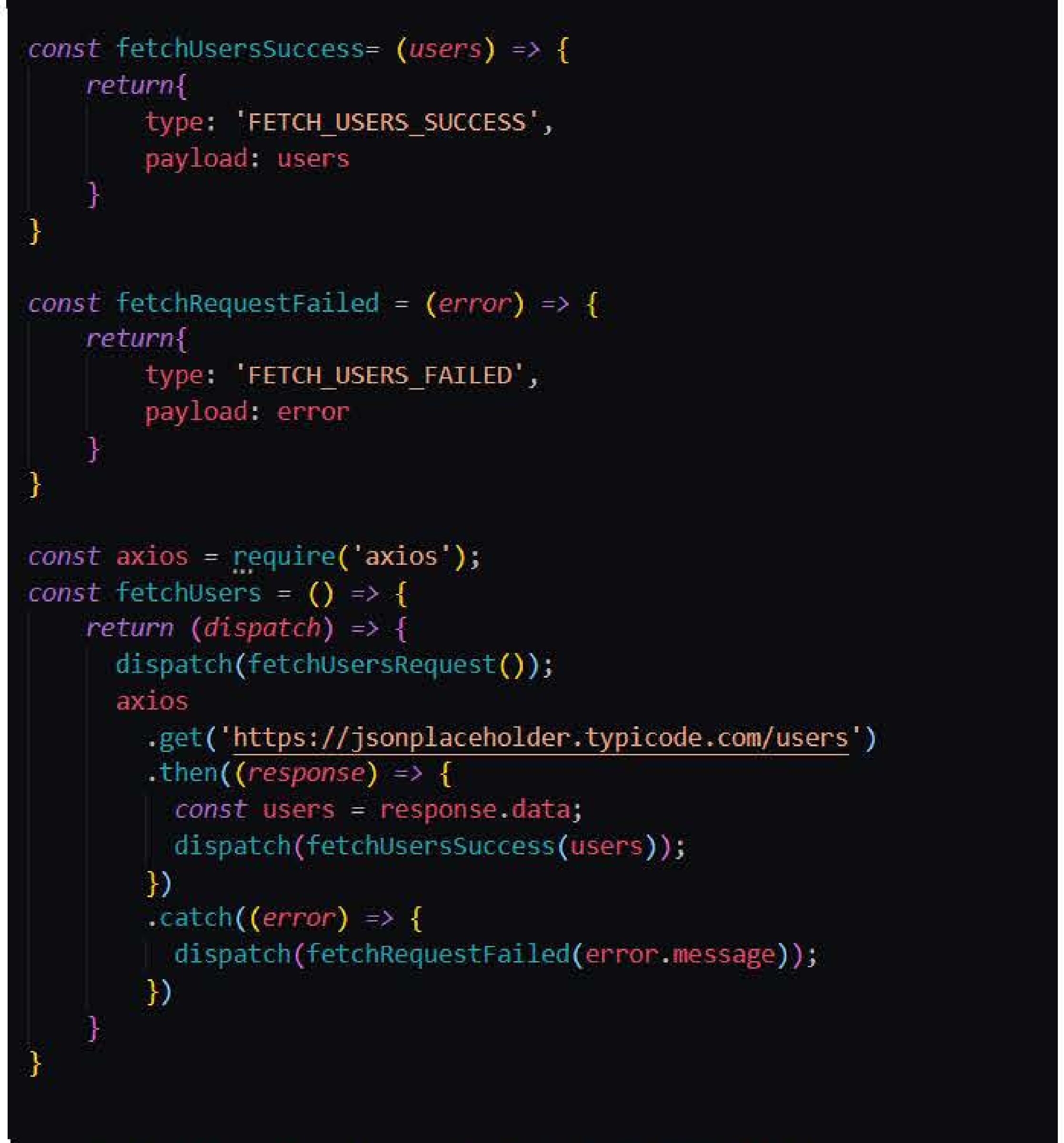
```
extraReducers: builder => {
    builder.addCase(cakeActions.buyCake, state => {
        state.noOfCake--
    })
}
```

It recieves an arguement 'builder'
builder.addCase(action_type, (state, action) => {
        Do something with state and action
})

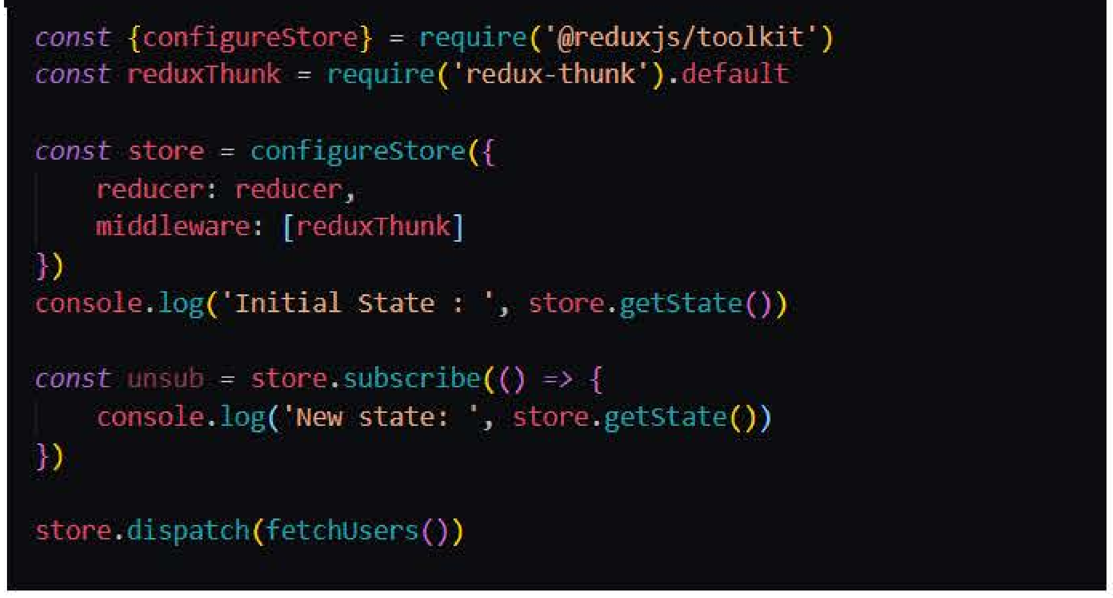
Async Actions using redux-toolkit



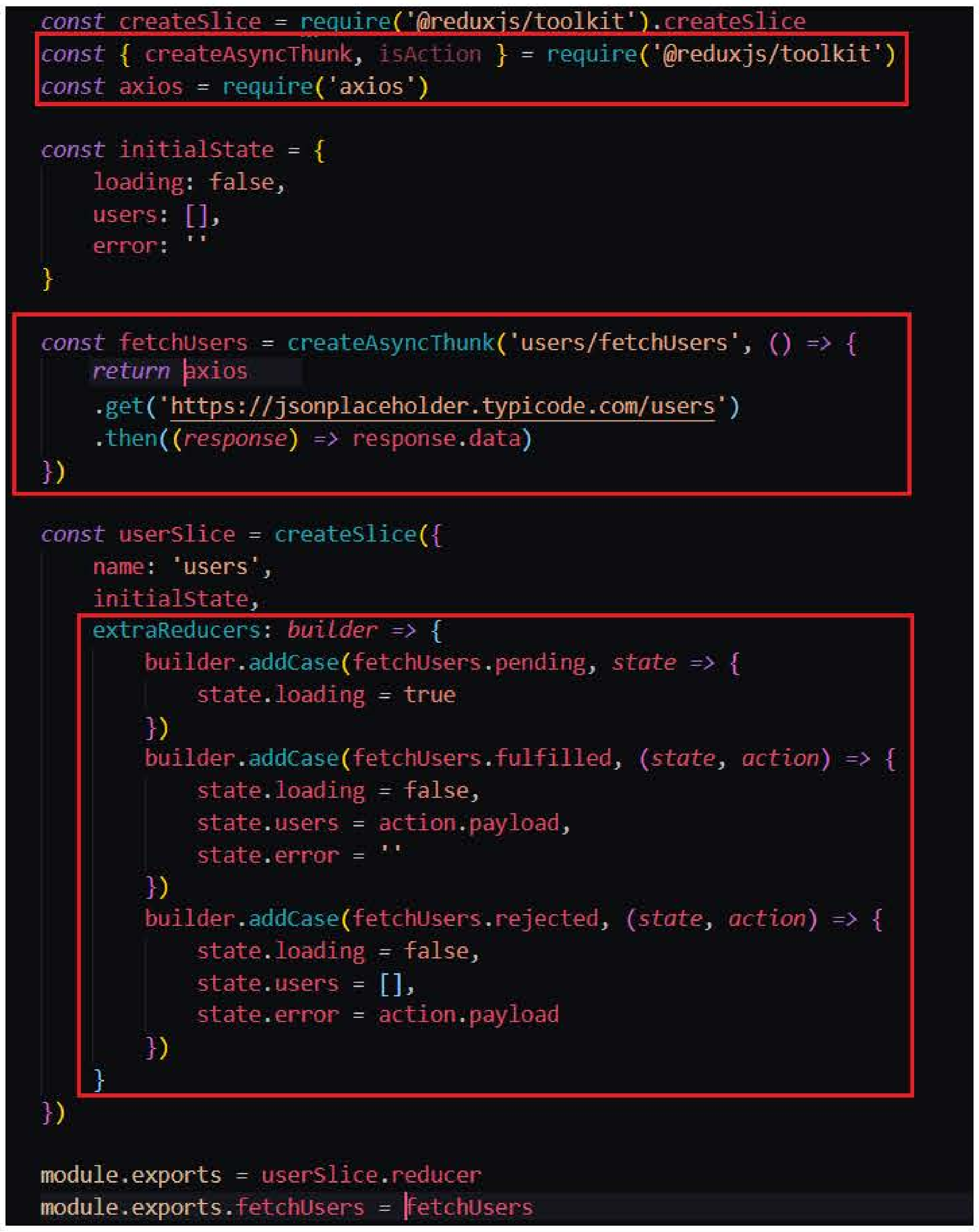
```
const initialState = {
    loading: false,
    users: [],
    error: ''
}

//Reducer
const reducer = (state=initialState, action) => {
    switch(action.type){
        case 'FETCH_USERS_REQ':
            return{
                ...state,
                loading: true
            }
        case 'FETCH_USERS_SUCCESS':
            return{
                loading: false,
                users: action.payload,
                error: ''
            }
        case 'FETCH_USERS_FAILED':
            return{
                loading: false,
                users: [],
                error: action.payload
            }
        default:
            return{...state}
    }
}

//Actions
const fetchUsersRequest= () => {
    return{
        type: 'FETCH_USERS_REQ'
    }
}
```



```
const fetchUsersSuccess= (users) => {
    return{
        type: 'FETCH_USERS_SUCCESS',
        payload: users
    }
}

const fetchRequestFailed = (error) => {
    return{
        type: 'FETCH_USERS_FAILED',
        payload: error
    }
}

const axios = require('axios');
const fetchUsers = () => {
    return (dispatch) => {
      dispatch(fetchUsersRequest());
      axios
        .get('https://jsonplaceholder.typicode.com/users')
        .then((response) => {
          const users = response.data;
          dispatch(fetchUsersSuccess(users));
        })
        .catch((error) => {
          dispatch(fetchRequestFailed(error.message));
        })
    }
}
```



```
const {configureStore} = require('@reduxjs/toolkit')
const reduxThunk = require('redux-thunk').default

const store = configureStore({
    reducer: reducer,
    middleware: [reduxThunk]
})
console.log('Initial State : ', store.getState())

const unsub = store.subscribe(() => {
    console.log('New state: ', store.getState())
})

store.dispatch(fetchUsers())
```



```
const createSlice = require('@reduxjs/toolkit').createSlice
const { createAsyncThunk, isAction } = require('@reduxjs/toolkit')
const axios = require('axios')

const initialState = {
    loading: false,
    users: [],
    error: ''
}

const fetchUsers = createAsyncThunk('users/fetchUsers', () => {
    return axios
    .get('https://jsonplaceholder.typicode.com/users')
    .then((response) => response.data)
})

const userSlice = createSlice({
    name: 'users',
    initialState,
    extraReducers: builder => {
        builder.addCase(fetchUsers.pending, state => {
            state.loading = true
        })
        builder.addCase(fetchUsers.fulfilled, (state, action) => {
            state.loading = false,
            state.users = action.payload,
            state.error = ''
        })
        builder.addCase(fetchUsers.rejected, (state, action) => {
            state.loading = false,
            state.users = [],
            state.error = action.payload
        })
    }
})

module.exports = userSlice.reducer
module.exports.fetchUsers = fetchUsers
```

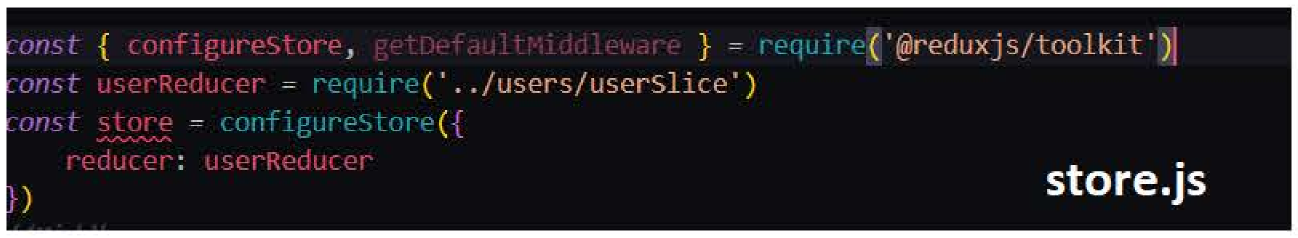
1. The redux-toolkit provides createAsyncThunk functon to implliment creating and dispatching asyc actions.
 createAsyncThunk(action name, func to fetch data)

2. We done need the catch block as error is handles.

3. We also dont need to create lifecycle acions like fetchRequestsSent, fetchRequestSuccess etc.... It is automatically dispatch lifecycle actions depending on the promise returned.

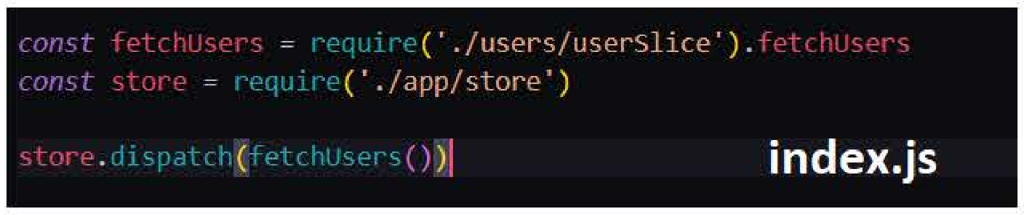
4. It returns either pending, fullfilled , rejected action types

5. The reducers to handle the lifecycle actions are not generated by the slice, we need to create and add it as extra reducers.



```
const { configureStore, getDefaultMiddleware } = require('@reduxjs/toolkit')
const userReducer = require('../users/userSlice')
const store = configureStore({
   reducer: userReducer
})
```

store.js



```
const fetchUsers = require('./users/userSlice').fetchUsers
const store = require('./app/store')

store.dispatch(fetchUsers())
```

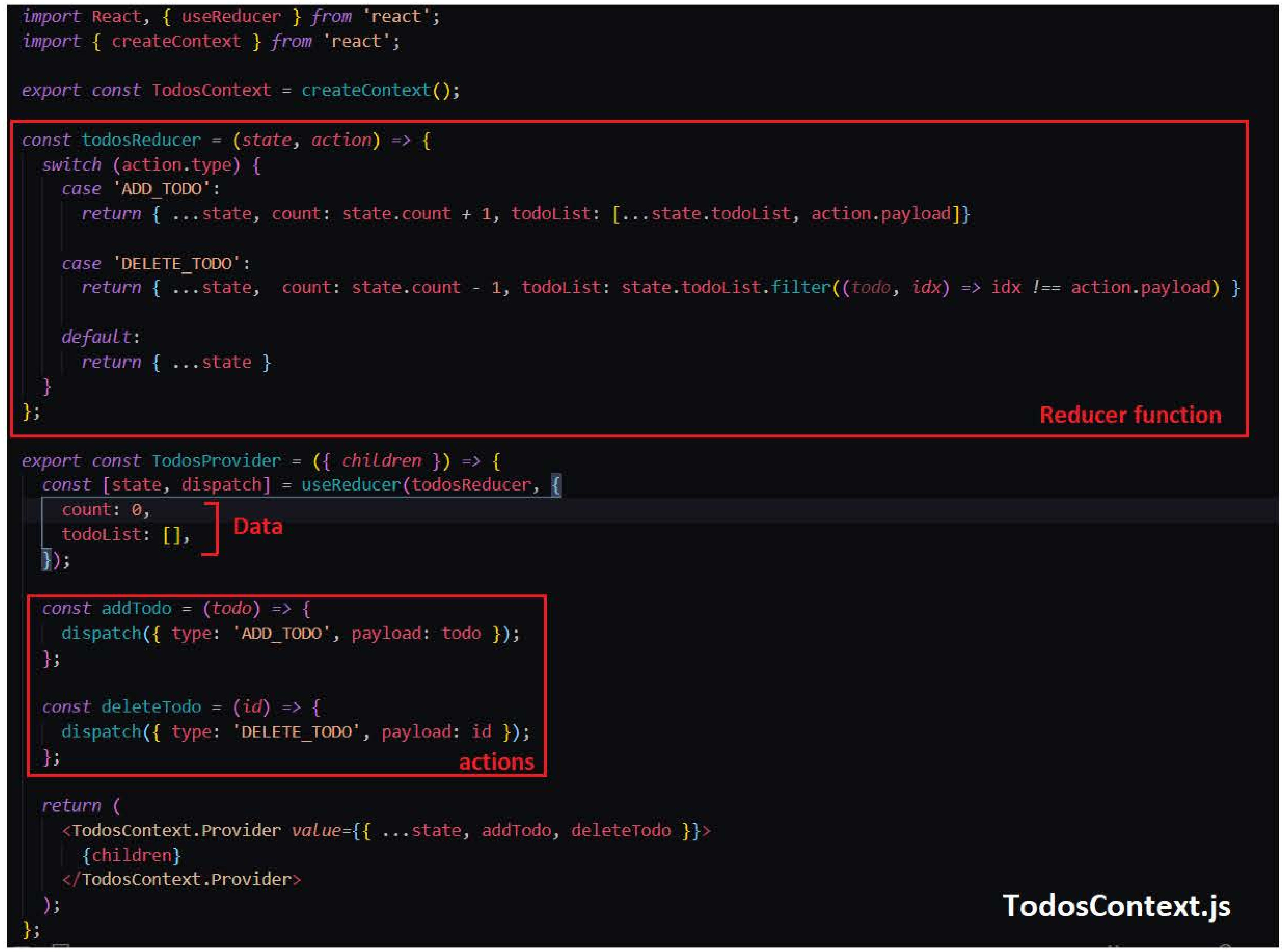
index.js

redux-thunk is used under the hood.
redux-thunk is added as a middleware by the toolkit

WHATEVER WE SAW TILL NOW IS INDEPENDENT OF REACT. TO MAKE IS COMPATABLE WITH REACT, WE JUST NEED TO REPLACE require WITH import AND module.exports WITH EXPORT. We just need to handle import and export

BY SHIVKARAMAN

Without Redux



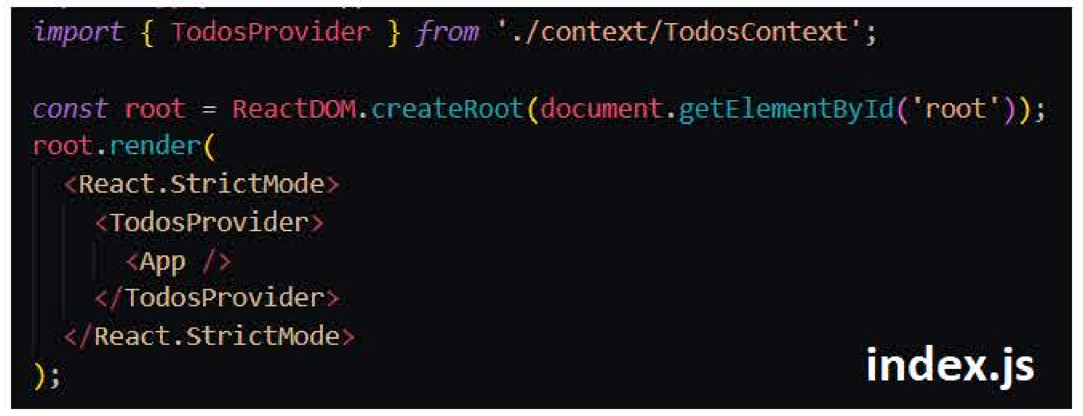
```
import React, { useReducer } from 'react';
import { createContext } from 'react';

export const TodosContext = createContext();

const todosReducer = (state, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return { ...state, count: state.count + 1, todoList: [...state.todoList, action.payload]}

    case 'DELETE_TODO':
      return { ...state,  count: state.count - 1, todoList: state.todoList.filter((todo, idx) => idx !== action.payload) }

    default:
      return { ...state }
  }
};

export const TodosProvider = ({ children }) => {
  const [state, dispatch] = useReducer(todosReducer, {
    count: 0,
    todoList: [],
  });

  const addTodo = (todo) => {
    dispatch({ type: 'ADD_TODO', payload: todo });
  };

  const deleteTodo = (id) => {
    dispatch({ type: 'DELETE_TODO', payload: id });
  };

  return (
    <TodosContext.Provider value={{ ...state, addTodo, deleteTodo }}>
      {children}
    </TodosContext.Provider>
  );
};
```

Reducer function

Data

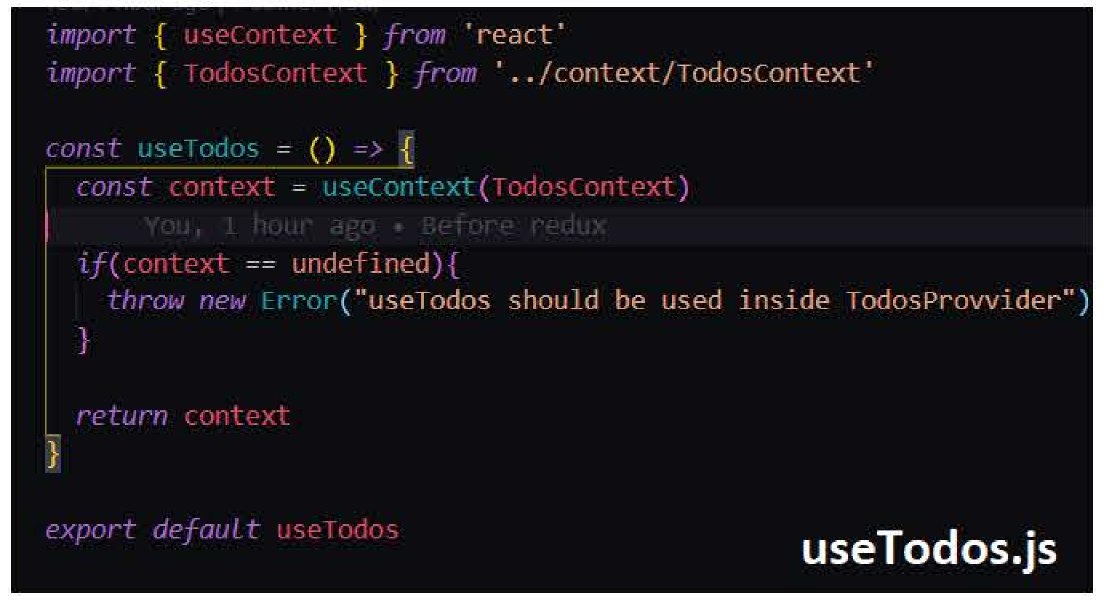
actions

TodosContext.js



```
import { TodosProvider } from './context/TodosContext';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <TodosProvider>
      <App />
    </TodosProvider>
  </React.StrictMode>
);
```

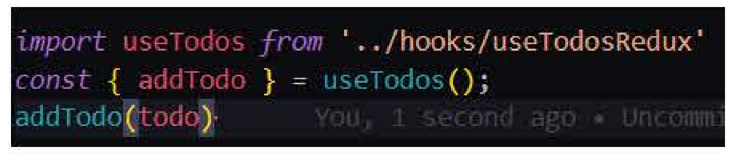
index.js



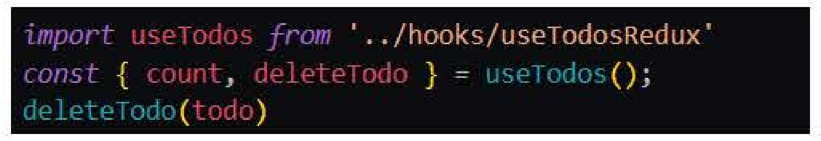
```
import { useContext } from 'react'
import { TodosContext } from '../context/TodosContext'

const useTodos = () => {
  const context = useContext(TodosContext)
      You, 1 hour ago • Before redux
  if(context == undefined){
    throw new Error("useTodos should be used inside TodosProvvider")
  }

  return context
}

export default useTodos
```

useTodos.js

Accessing functions and variables

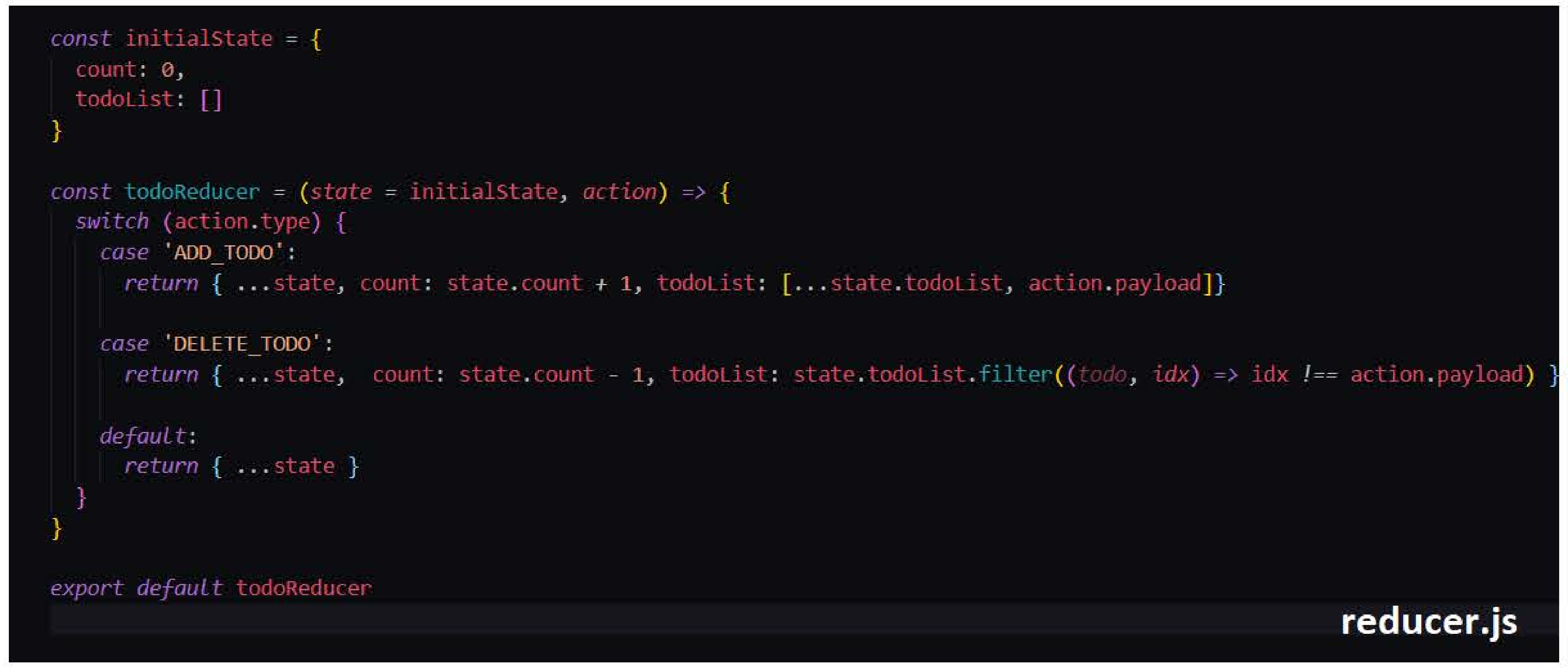


```
import useTodos from '../hooks/useTodosRedux'
const { addTodo } = useTodos();
addTodo(todo)        You, 1 second ago • Uncomm
```



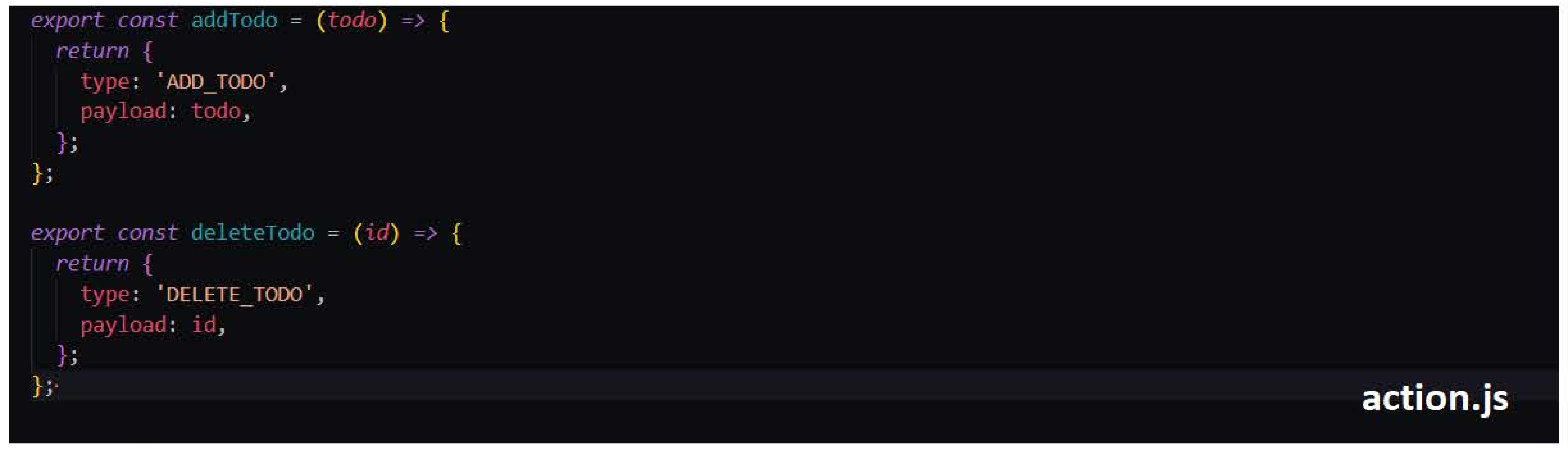
```
import useTodos from '../hooks/useTodosRedux'
const { count, deleteTodo } = useTodos();
deleteTodo(todo)
```

With Redux



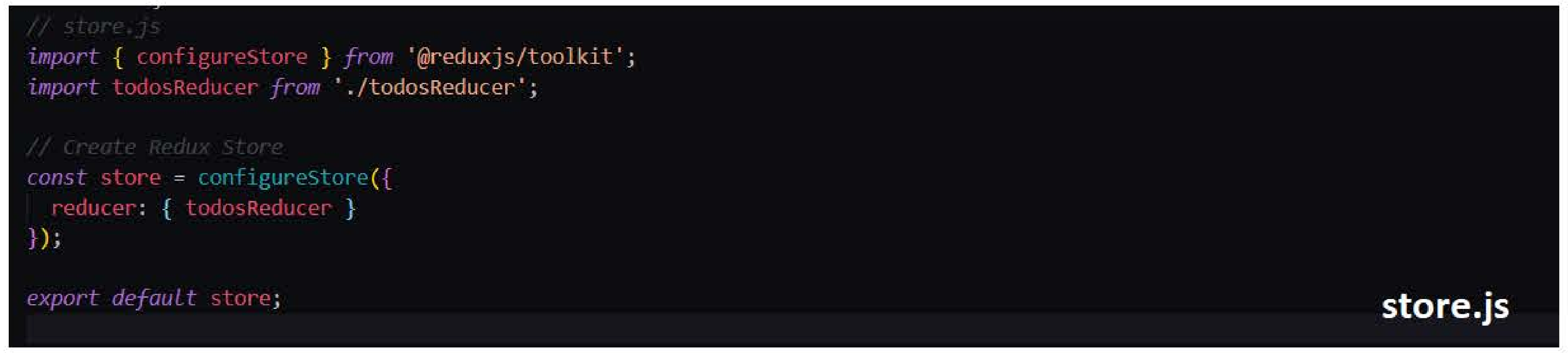
```
const initialState = {
  count: 0,
  todoList: []
}

const todoReducer = (state = initialState, action) => {
  switch (action.type) {
    case 'ADD_TODO':
      return { ...state, count: state.count + 1, todoList: [...state.todoList, action.payload]}

    case 'DELETE_TODO':
      return { ...state,  count: state.count - 1, todoList: state.todoList.filter((todo, idx) => idx !== action.payload) }

    default:
      return { ...state }
  }
}

export default todoReducer
```

reducer.js



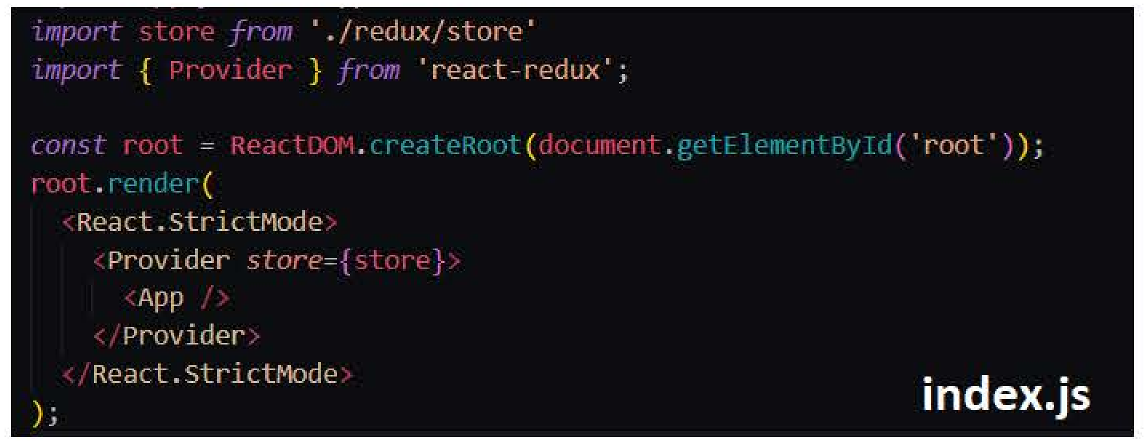
```
export const addTodo = (todo) => {
  return {
    type: 'ADD_TODO',
    payload: todo,
  };
};

export const deleteTodo = (id) => {
  return {
    type: 'DELETE_TODO',
    payload: id,
  };
};
```

action.js



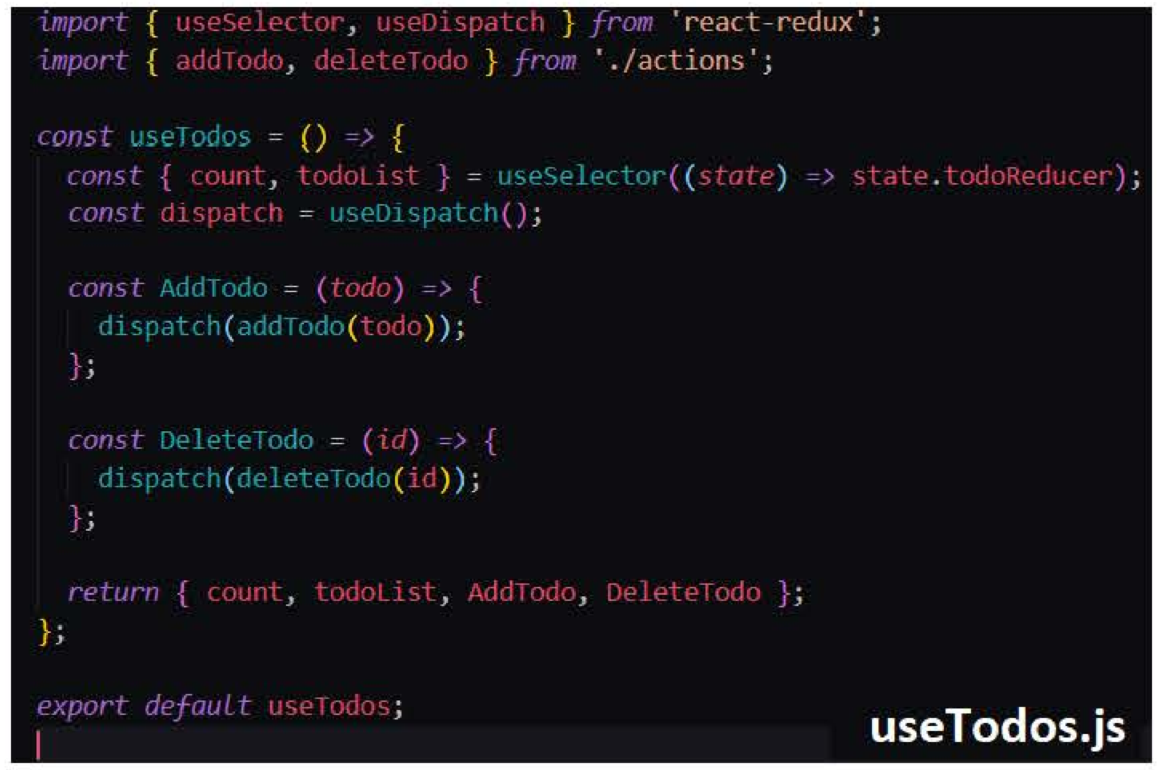
```
// store.js
import { configureStore } from '@reduxjs/toolkit';
import todosReducer from './todosReducer';

// Create Redux Store
const store = configureStore({
  reducer: { todosReducer }
});

export default store;
```

store.js



```
import store from './redux/store'
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

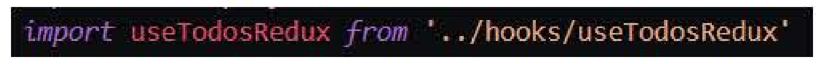
index.js



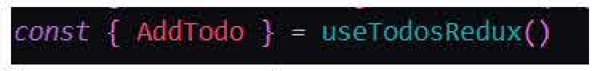
```
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, deleteTodo } from './actions';

const useTodos = () => {
  const { count, todoList } = useSelector((state) => state.todoReducer);
  const dispatch = useDispatch();

  const AddTodo = (todo) => {
    dispatch(addTodo(todo));
  };

  const DeleteTodo = (id) => {
    dispatch(deleteTodo(id));
  };

  return { count, todoList, AddTodo, DeleteTodo };
};

export default useTodos;
```

useTodos.js

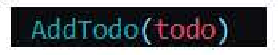
Acccessing Functions



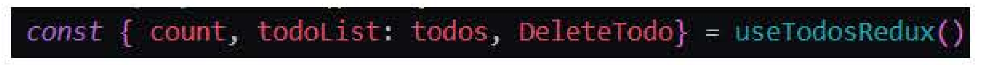
```
import useTodosRedux from '../hooks/useTodosRedux'
```



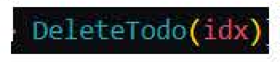
```
const { AddTodo } = useTodosRedux()
```



```
AddTodo(todo)
```

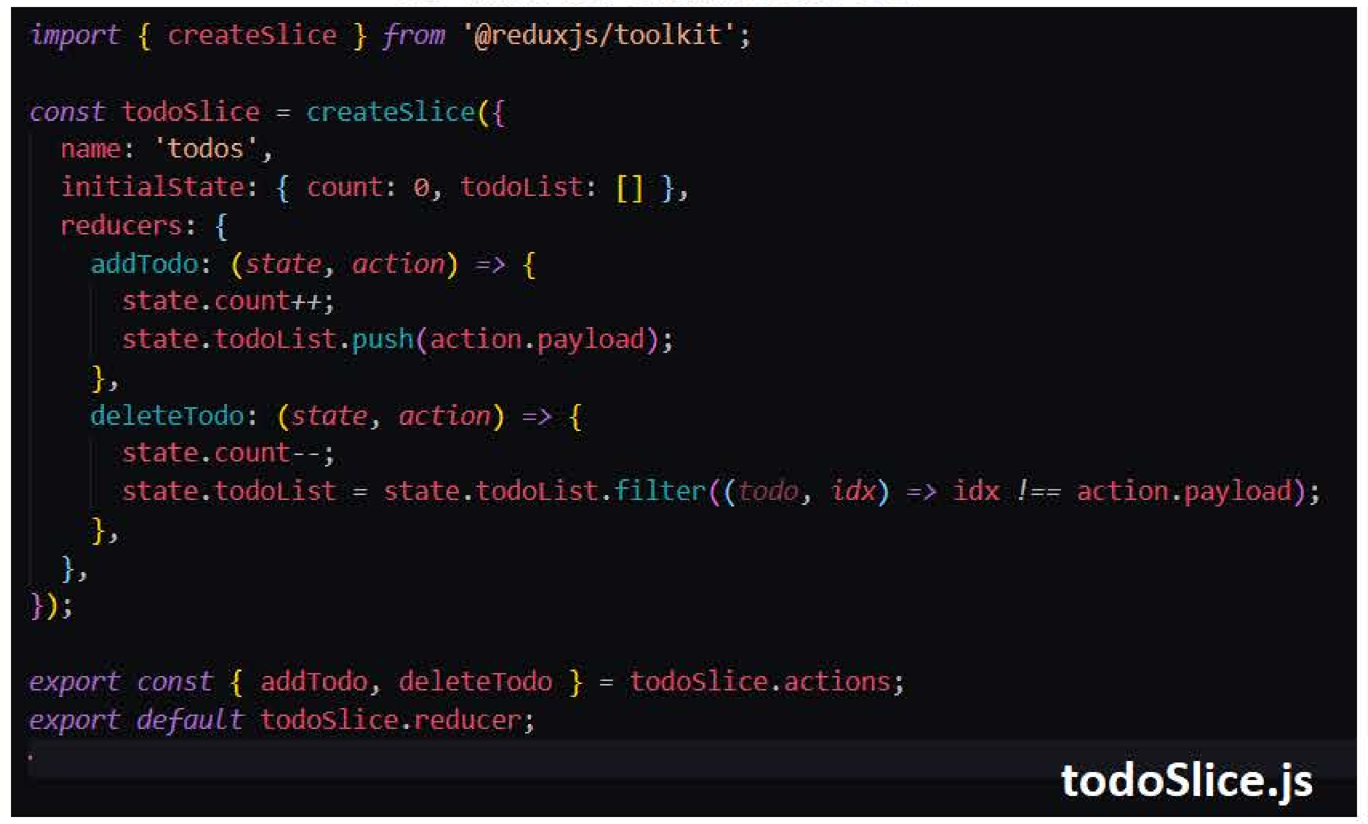


```
const { count, todoList: todos, DeleteTodo} = useTodosRedux()
```



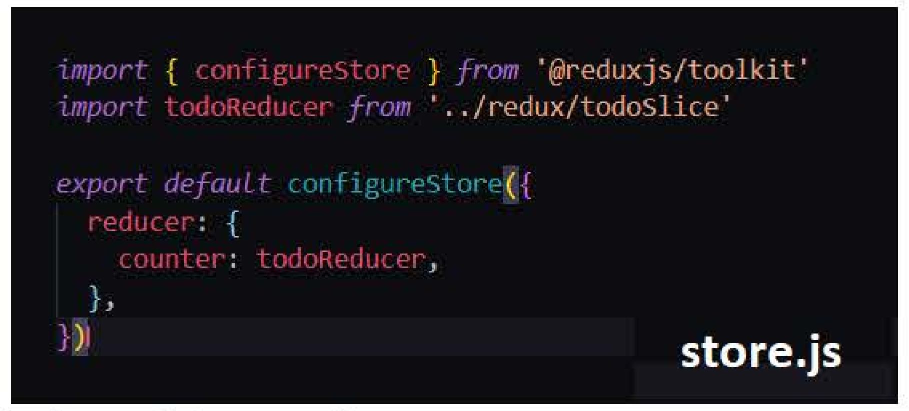
```
DeleteTodo(idx)
```

With createSlice



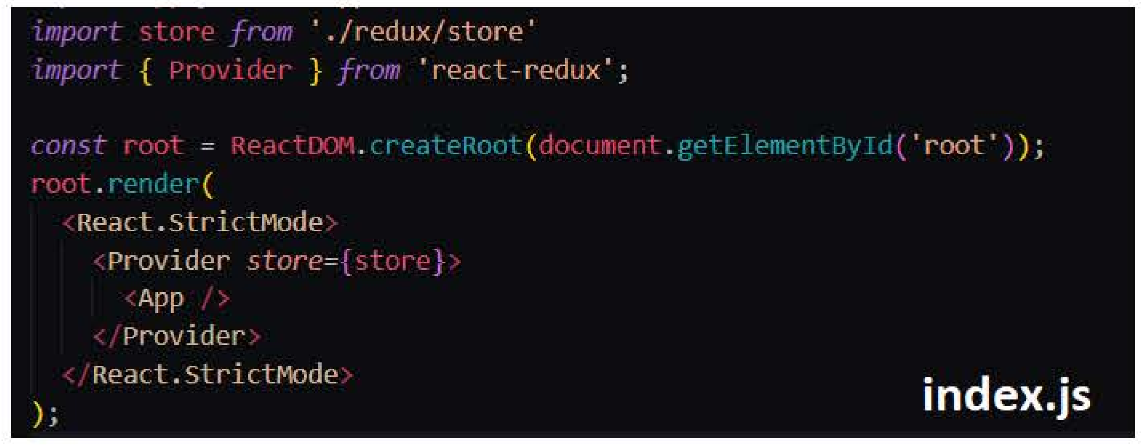
```
import { createSlice } from '@reduxjs/toolkit';

const todoSlice = createSlice({
  name: 'todos',
  initialState: { count: 0, todoList: [] },
  reducers: {
    addTodo: (state, action) => {
      state.count++;
      state.todoList.push(action.payload);
    },
    deleteTodo: (state, action) => {
      state.count--;
      state.todoList = state.todoList.filter((todo, idx) => idx !== action.payload);
    },
  },
});

export const { addTodo, deleteTodo } = todoSlice.actions;
export default todoSlice.reducer;
```

todoSlice.js



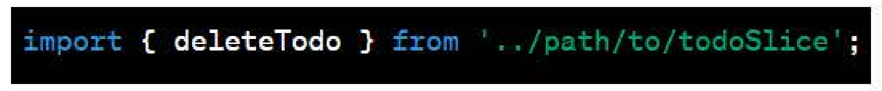
```
import { configureStore } from '@reduxjs/toolkit'
import todoReducer from '../redux/todoSlice'

export default configureStore({
  reducer: {
    counter: todoReducer,
  },
})
```

store.js

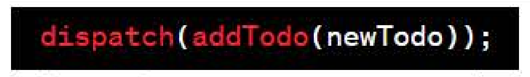


```
import store from './redux/store'
import { Provider } from 'react-redux';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

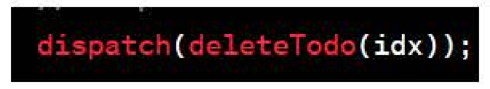
index.js



```
import { deleteTodo } from '../path/to/todoSlice';
```



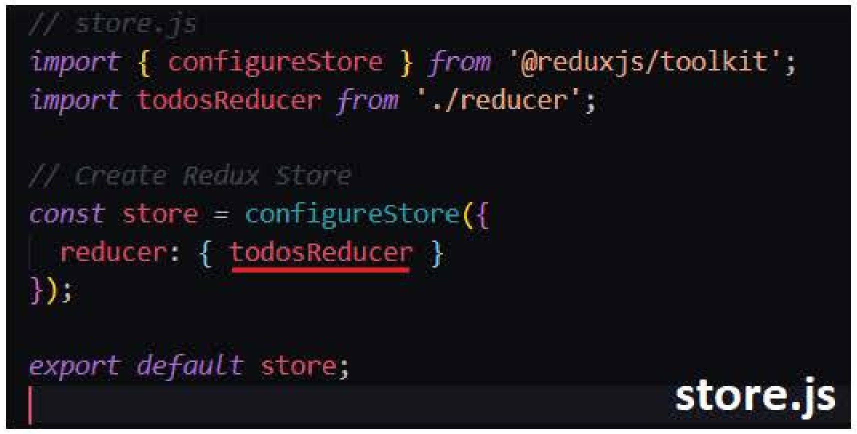
```
dispatch(addTodo(newTodo));
```



```
dispatch(deleteTodo(idx));
```

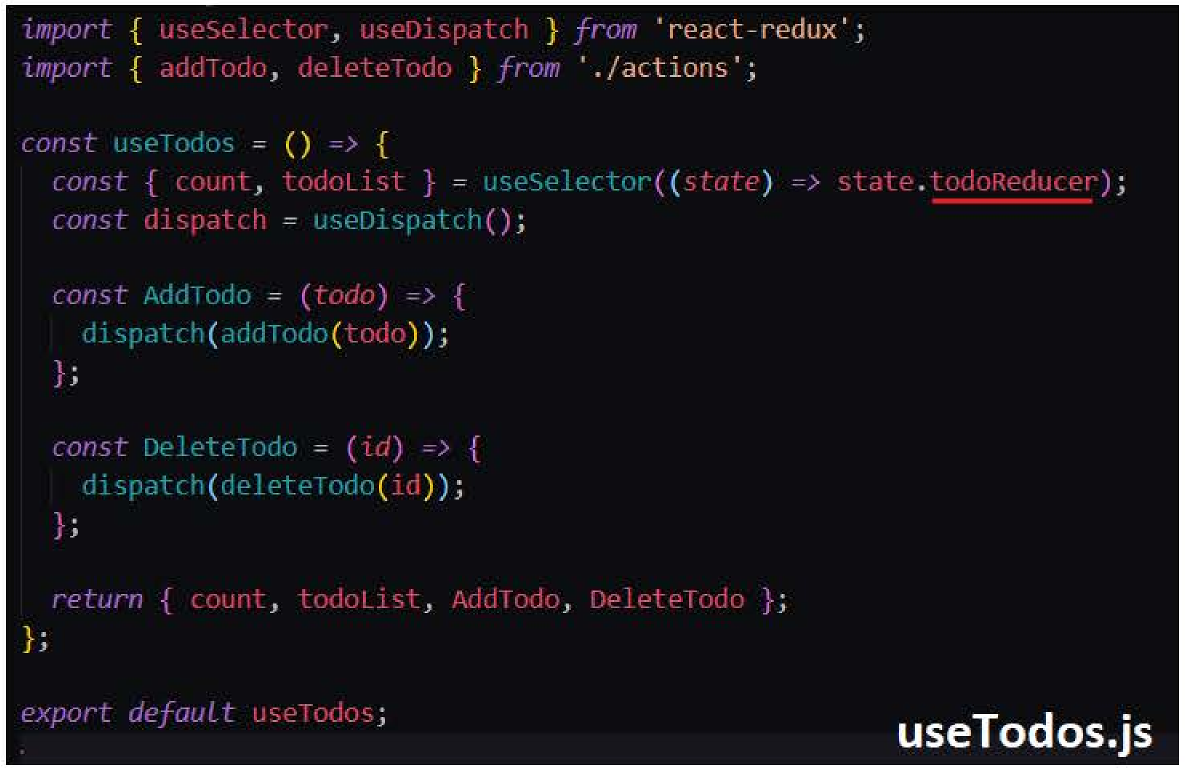
Accessing the functions is same as in the atherr approach

BY SHIVKARAMAN



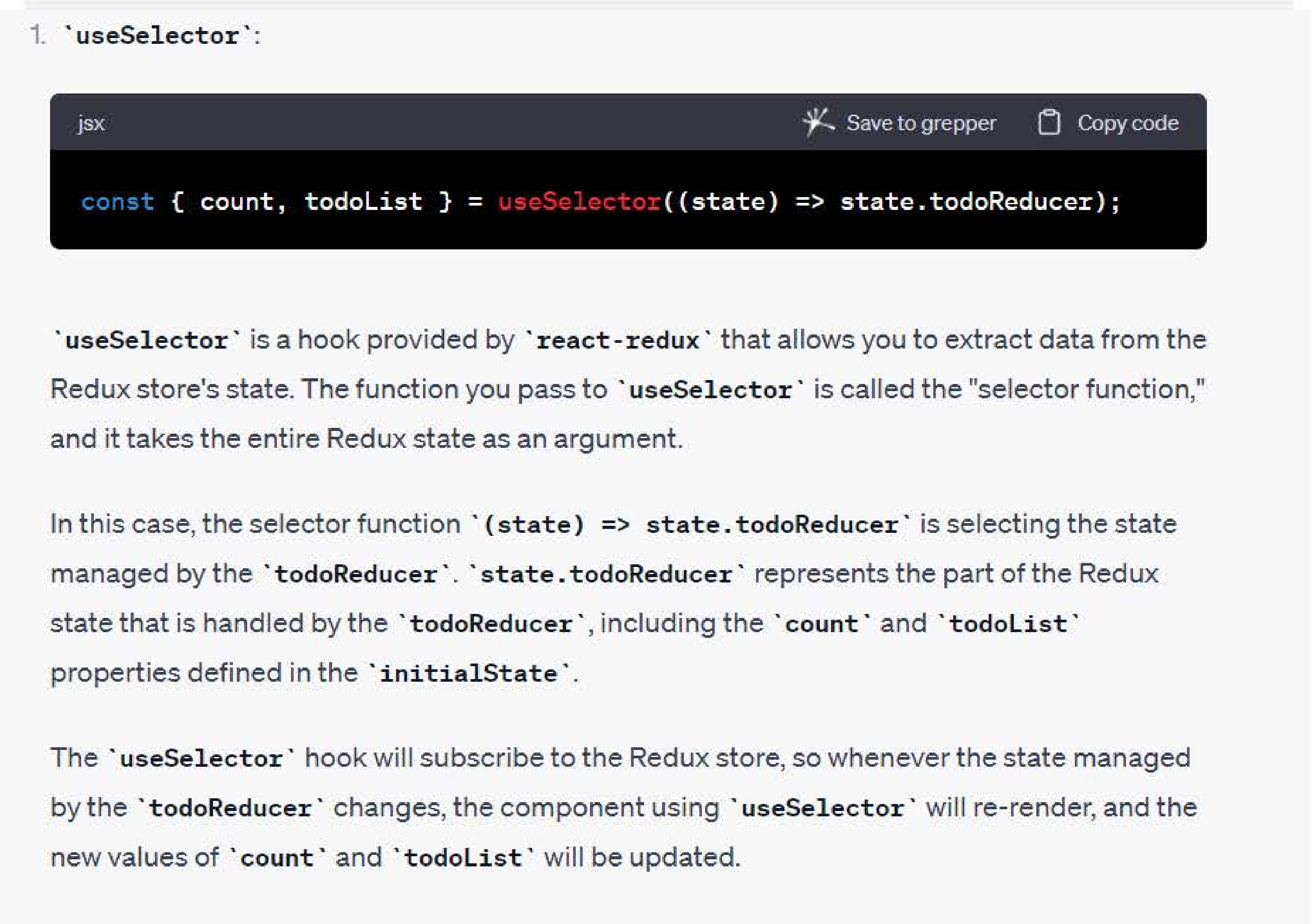
```js
// store.js
import { configureStore } from '@reduxjs/toolkit';
import todosReducer from './reducer';

// Create Redux Store
const store = configureStore({
  reducer: { todosReducer }
});

export default store;
```

store.js



```js
import { useSelector, useDispatch } from 'react-redux';
import { addTodo, deleteTodo } from './actions';

const useTodos = () => {
  const { count, todoList } = useSelector((state) => state.todoReducer);
  const dispatch = useDispatch();

  const AddTodo = (todo) => {
    dispatch(addTodo(todo));
  };

  const DeleteTodo = (id) => {
    dispatch(deleteTodo(id));
  };

  return { count, todoList, AddTodo, DeleteTodo };
};

export default useTodos;
.
```

useTodos.js



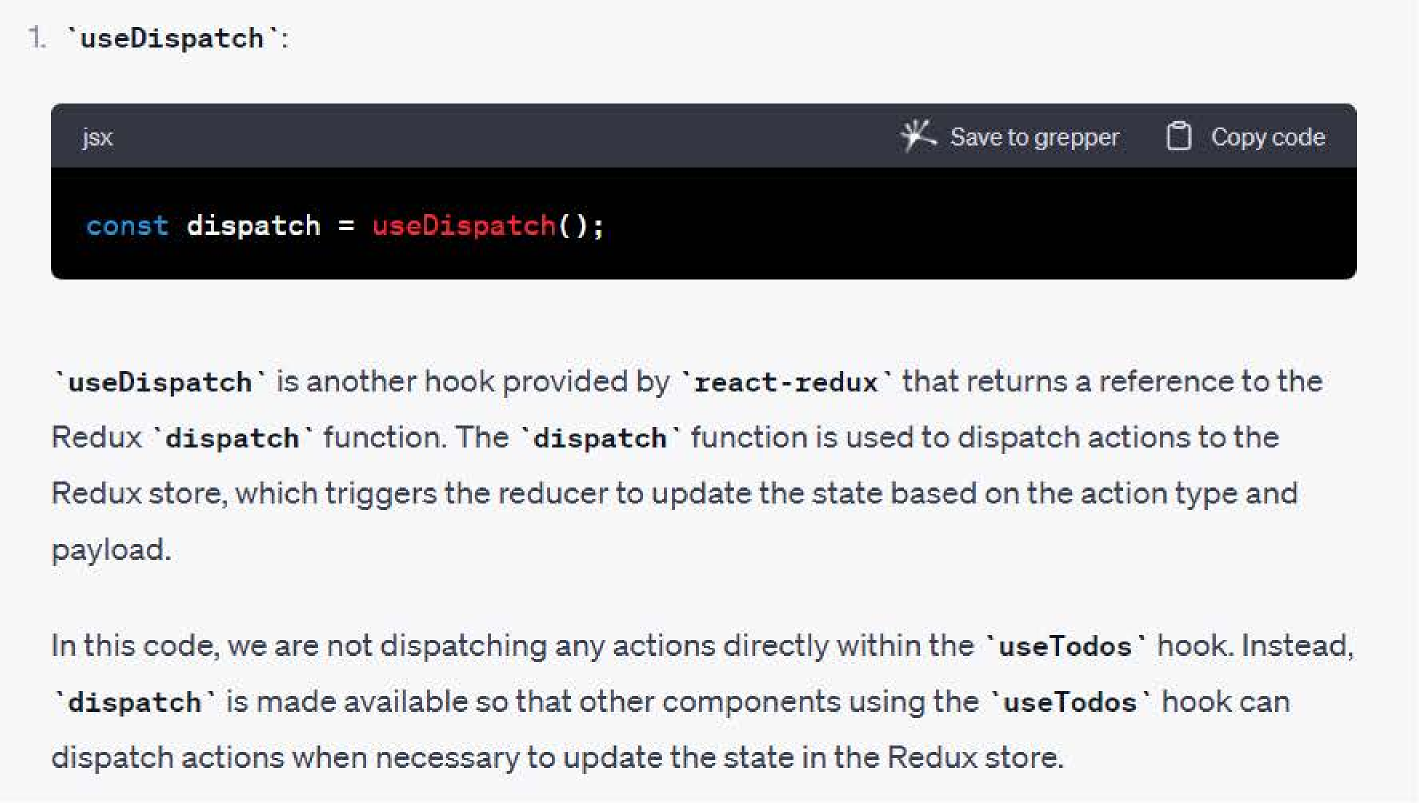
1. `useSelector`:

```jsx
const { count, todoList } = useSelector((state) => state.todoReducer);
```

`useSelector` is a hook provided by `react-redux` that allows you to extract data from the Redux store's state. The function you pass to `useSelector` is called the "selector function," and it takes the entire Redux state as an argument.

In this case, the selector function `(state) => state.todoReducer` is selecting the state managed by the `todoReducer`. `state.todoReducer` represents the part of the Redux state that is handled by the `todoReducer`, including the `count` and `todoList` properties defined in the `initialState`.

The `useSelector` hook will subscribe to the Redux store, so whenever the state managed by the `todoReducer` changes, the component using `useSelector` will re-render, and the new values of `count` and `todoList` will be updated.

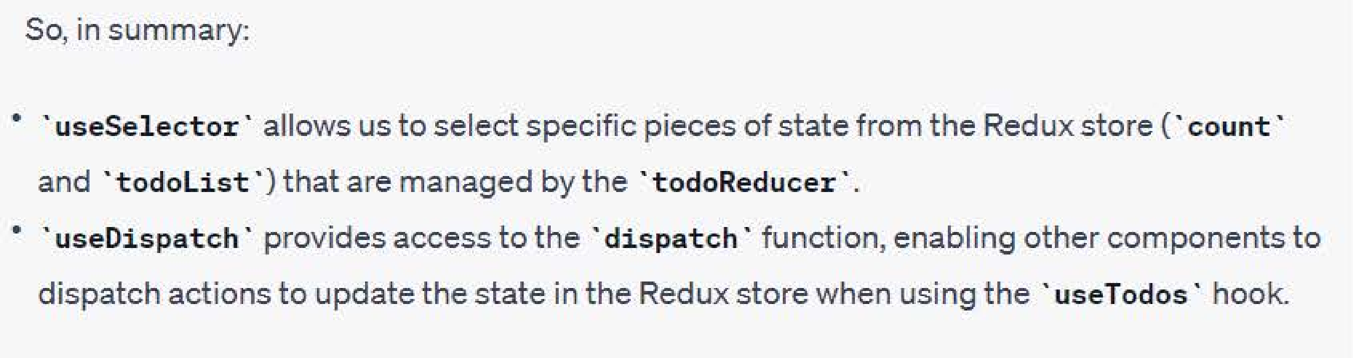


1. `useDispatch`:

```jsx
const dispatch = useDispatch();
```

`useDispatch` is another hook provided by `react-redux` that returns a reference to the Redux `dispatch` function. The `dispatch` function is used to dispatch actions to the Redux store, which triggers the reducer to update the state based on the action type and payload.

In this code, we are not dispatching any actions directly within the `useTodos` hook. Instead, `dispatch` is made available so that other components using the `useTodos` hook can dispatch actions when necessary to update the state in the Redux store.



So, in summary:

- `useSelector` allows us to select specific pieces of state from the Redux store (`count` and `todoList`) that are managed by the `todoReducer`.
- `useDispatch` provides access to the `dispatch` function, enabling other components to dispatch actions to update the state in the Redux store when using the `useTodos` hook.

BY SHIVKARAMAN