ReactJS

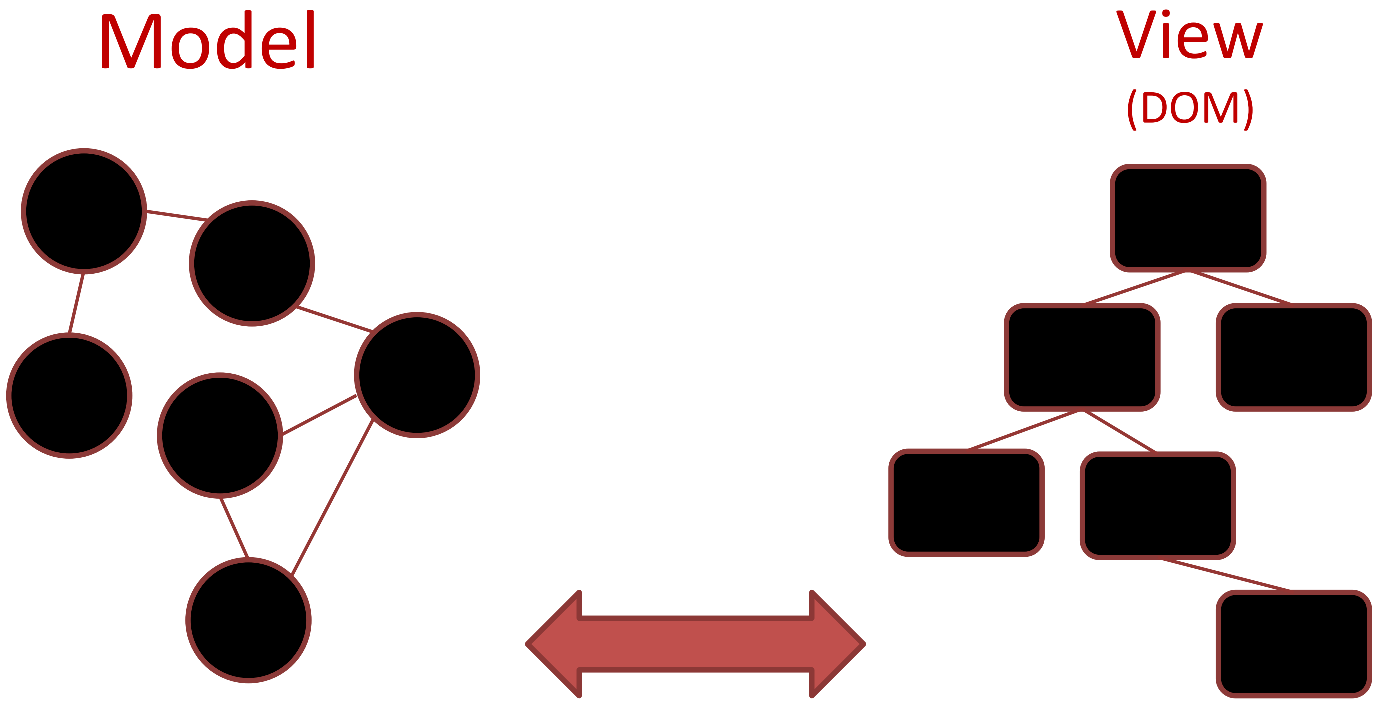
State Management

With Redux

{coding
academy

Overview

- We make the Model (**state**) of the application visible to user by the process we call rendering.
- Commonly, the View allows modifying the Model



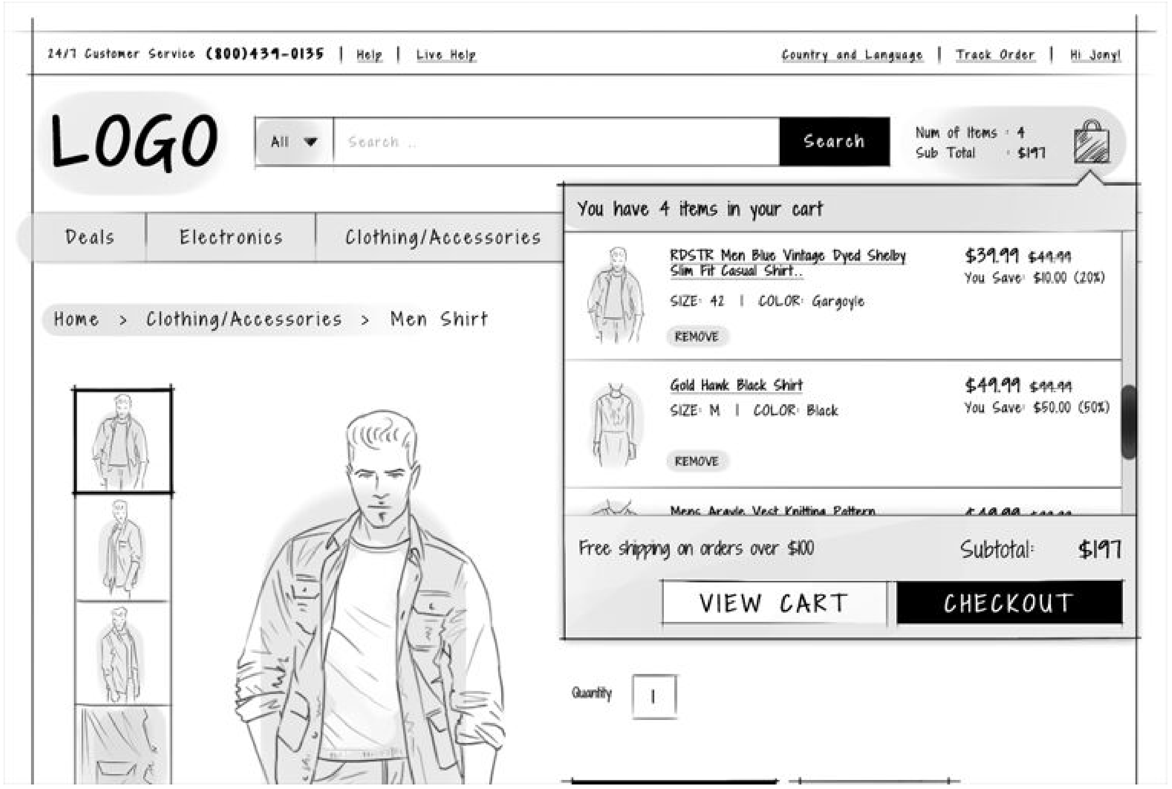
Frontend State

When app grows, there is <u>a lot</u> of that state

- Some of the state comes from the server
- Some of this state is frontend only

Frontend State Examples

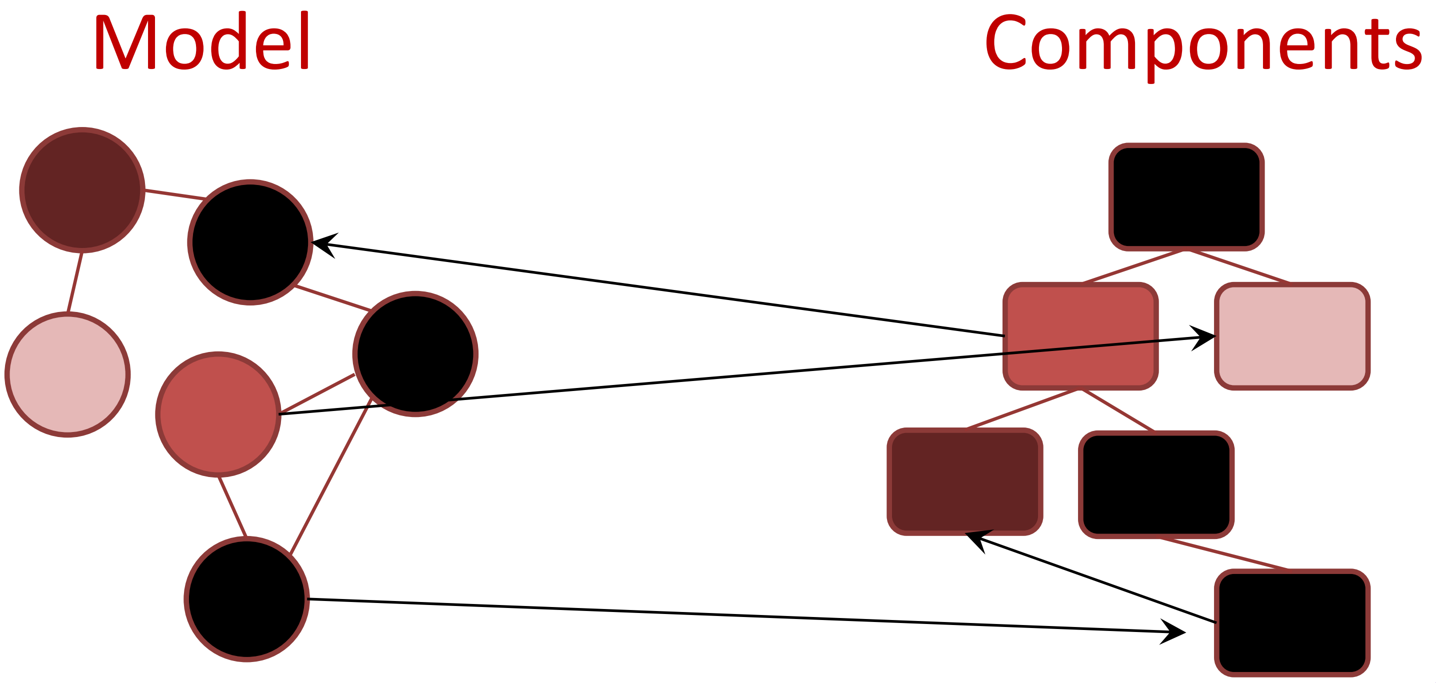
- Logged-in User
- Shopping Cart
- *currentlyPlayingSong*
- *Display Preferences*
- Wizard Step
- Permitted features


24/7 Customer Service (800)434-0135 | Help | Live Help
Country and Language | Track Order | Hi Jony!
LOGO
All
Search ..
Search
Num of Items : 4
Sub Total : $197
Deals
Electronics
Clothing/Accessories
Home > Clothing/Accessories > Men Shirt
You have 4 items in your cart
RDSTR Men Blue Vintage Dyed Shelby Slim Fit Casual Shirt..
SIZE: 42 | COLOR: Gargoyle
REMOVE
$39.99 $49.99
You Save: $10.00 (20%)
Gold Hawk Black Shirt
SIZE: M | COLOR: Black
REMOVE
$49.99 $99.99
You Save: $50.00 (50%)
Mens Argyle Vest Knitting Pattern
Free shipping on orders over $100
Subtotal: $197
VIEW CART
CHECKOUT
Quantity
1

{coding academy

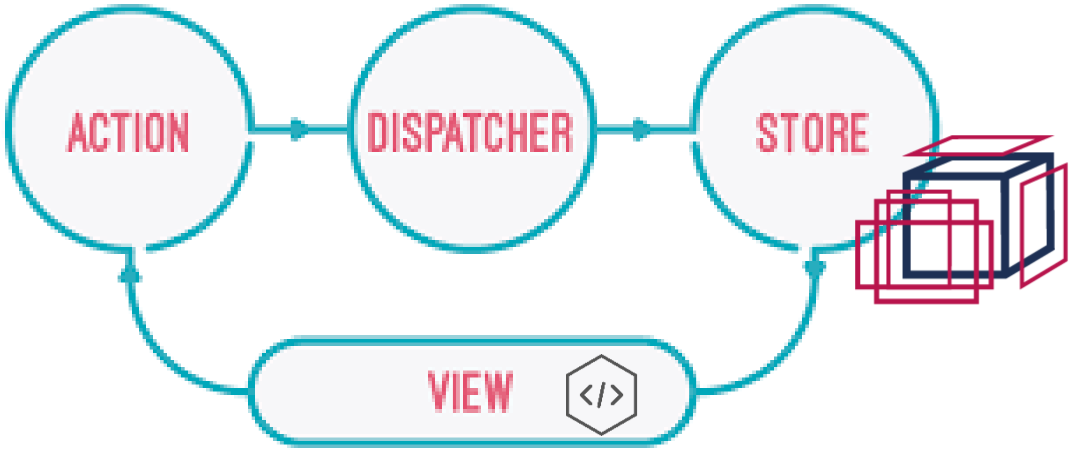
Shared Mutable State is a pain

When something change, we need to sync the changes across the application

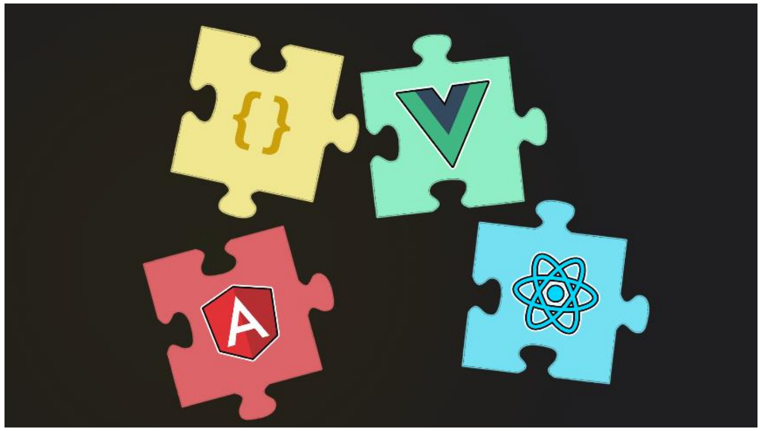


Enters Flux

- Flux is a design idea
- It is about a one-way flow of state
- Where components go through a dispatcher to alter the state of the application

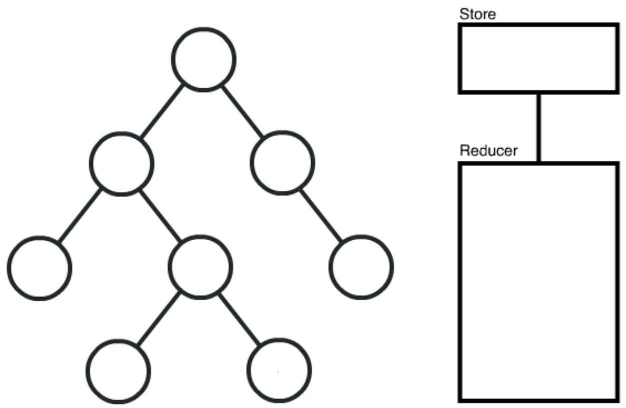


Redux implements Flux



- Redux is a small library that introduced an elegant, yet profoundly simple way to manage application state.
- Redux was born and raised at React
  and the idea was adopted by Angular & Vue.

React - Redux



● State change initited

● State change

- **React-Redux** is the glue library between React and Redux
- Usually, state changes lead to rendering:

The component will be checked for re-rendering when relevant store-state has changed

```
<!-- index.html -->
<script src="lib/react-redux.js"></script>
<script src="lib/redux.js"></script>
```

Wire up the Store

We wrap our `<App>` with `<Provider>` which make the store available to our components.

```
const { Provider } = ReactRedux
import { store } from "./store/store.js"

export function App() {
        return (
            <Provider store={store}>
                <Router>
                    <section className="main-layout app">
                        <AppHeader />
                        <main>
                            <Routes>
                                <Route element={<HomePage />} path="/" />
                                <Route element={<CarIndex />} path="/car" />
                            </Routes>
                        </main>
                        <AppFooter />
                    </section>
                </Router>
            </Provider>
        )
}
```

Store state and the reducer

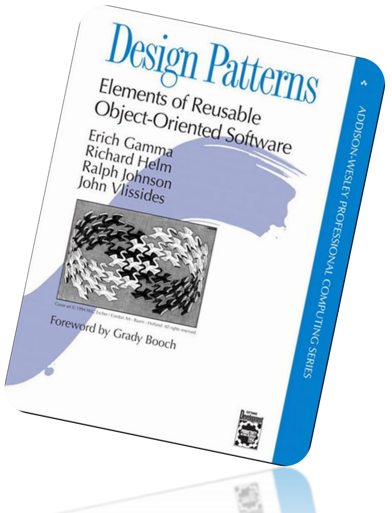
Here is simple store

```
const { createStore } = Redux

const initialState = {
    count: 101
}

function appReducer(state = initialState, cmd) {
    switch (cmd.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 }
        case 'DECREMENT':
            return { ...state, count: state.count - 1 }
        case 'CHANGE_BY':
            return { ...state, count: state.count + cmd.diff }
       default:return  state
}

export const store = createStore(appReducer)
store.subscribe(() => {
    console.log('Current state is:', store.getState())
})
```

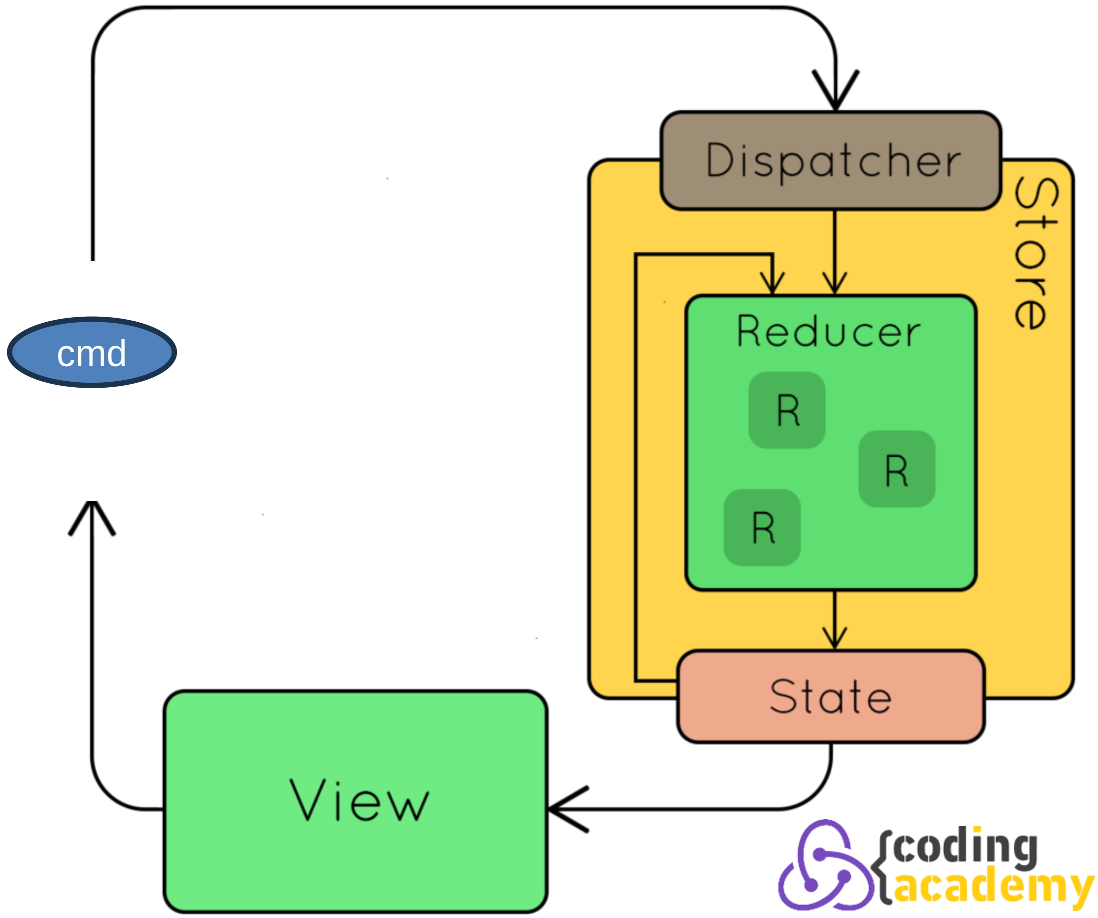
The store

So, we extract the shared state out of the components, and manage it in a global singleton – the store

To change the store state
we dispatch a command (action)



Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



Redux - The Store

- The store holds the state and allows subscribing to changes
- It also provide way to dispatch commands to the reducer

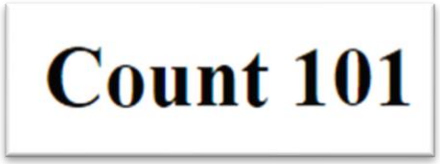
```
const initialState = {
    count: 101
}

// Default value passed to reducer is {count: 101}
function appReducer(state = initialState, cmd) {
    switch (cmd.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 }
        case 'DECREMENT':
            return { ...state, count: state.count - 1 }
        default:return state
}
```

```
store.dispatch({ type: 'INCREMENT' })
// Counter state is: {count: 102}

store.dispatch({ type: 'INCREMENT' })
// Counter state is: {count: 103}

store.dispatch({ type: 'DECREMENT' })
// Counter state is: {count: 102}
```

Selecting state from the store

Here is a component getting some store-state from the store

```
const { useSelector } = ReactRedux
export function Cmp() {
    // Instead of using local state:
    // const [count, setCount] = useState(10)

    // We select some state from the store-state
    const count = useSelector((storeState) => storeState.count)
    return (
        <section>
                Count {count}
        </section >
    )
}
```

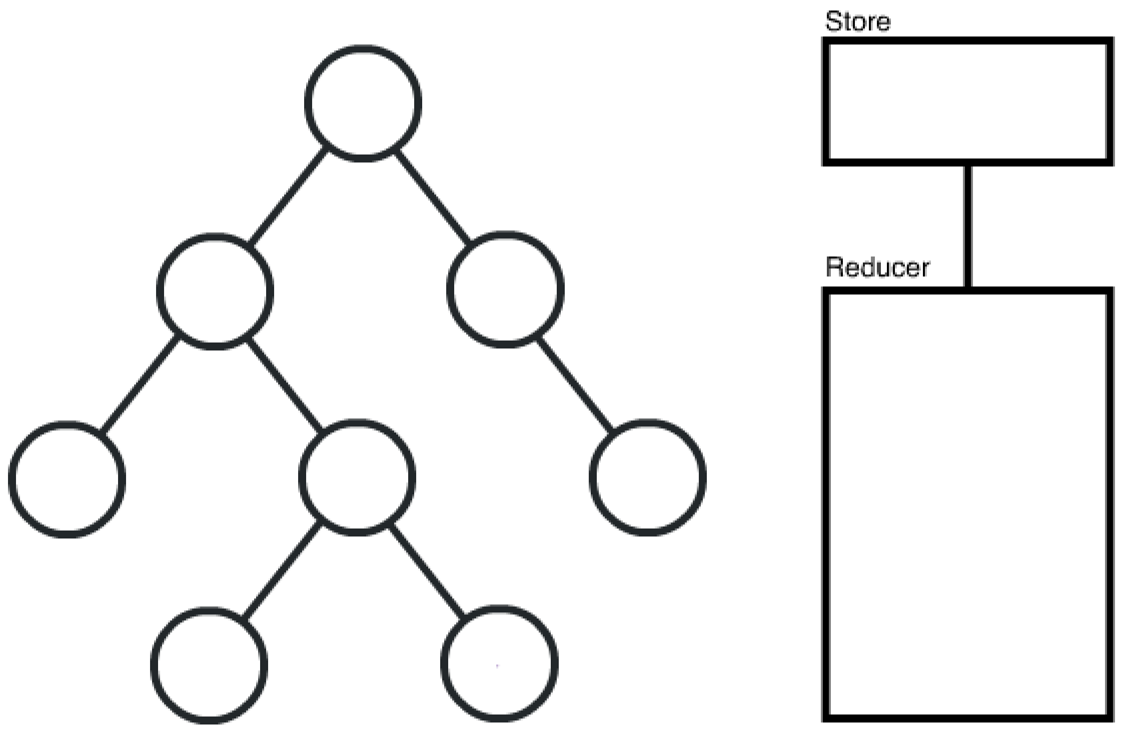

Count 101

Updating store-state

Here is a component that updates some store-state

```
const { useSelector, useDispatch } = ReactRedux
export function Cmp() {

    const dispatch = useDispatch()

    // Select some state from the store-state
    const count = useSelector((storeState) => storeState.count)
    function increment(diff) {
        dispatch({ type: 'CHANGE_BY', diff })
        // Same as:
        // dispatch({ type: 'INCREMENT' })
    }

    return (
        <section>
                Count {count}
                <button onClick={() => {
                    increment(1)
                }}>+</button>
        </section >
    )
}
```

Unidirectional data flow



● State change initited

● State change

Redux – Dispatching command

```
// In some component
const { useDispatch } = ReactRedux

const dispatch = useDispatch()

dispatch({ type: 'INCREMENT'})
dispatch({ type: 'DECREMENT'})
dispatch({ type: 'CHANGE_BY', diff : 10 })
```

- The command is an object with information about the needed state change
- It always has a type and usually some data
- In our components, we can get access to the `dispatch` function with `useDispatch`

Redux - reducers

- When a command is dispatched, the reducer is called, receiving the latest state and the current command

- The reducer performs the command and returns the modified state.

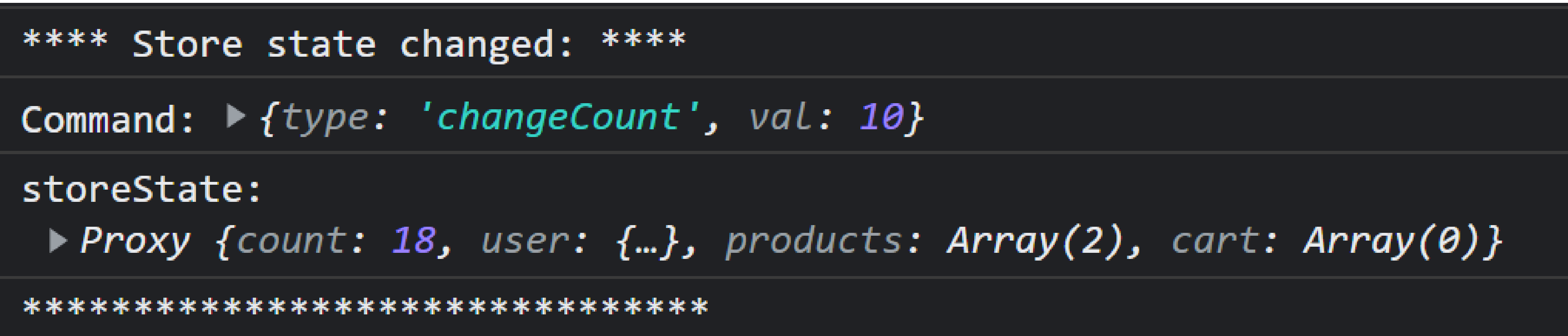
```
function appReducer(state = initialState, cmd) {
    switch (cmd.type) {
        case 'INCREMENT':
            return { ...state, count: state.count + 1 }
        case 'DECREMENT':
            return { ...state, count: state.count - 1 }
        case 'CHANGE_BY':
            return { ...state, count: state.count + cmd.diff }
        default: return state
}
```

{coding
academy

Logging our store changes

We can subscribe to the store and print out the updated state:

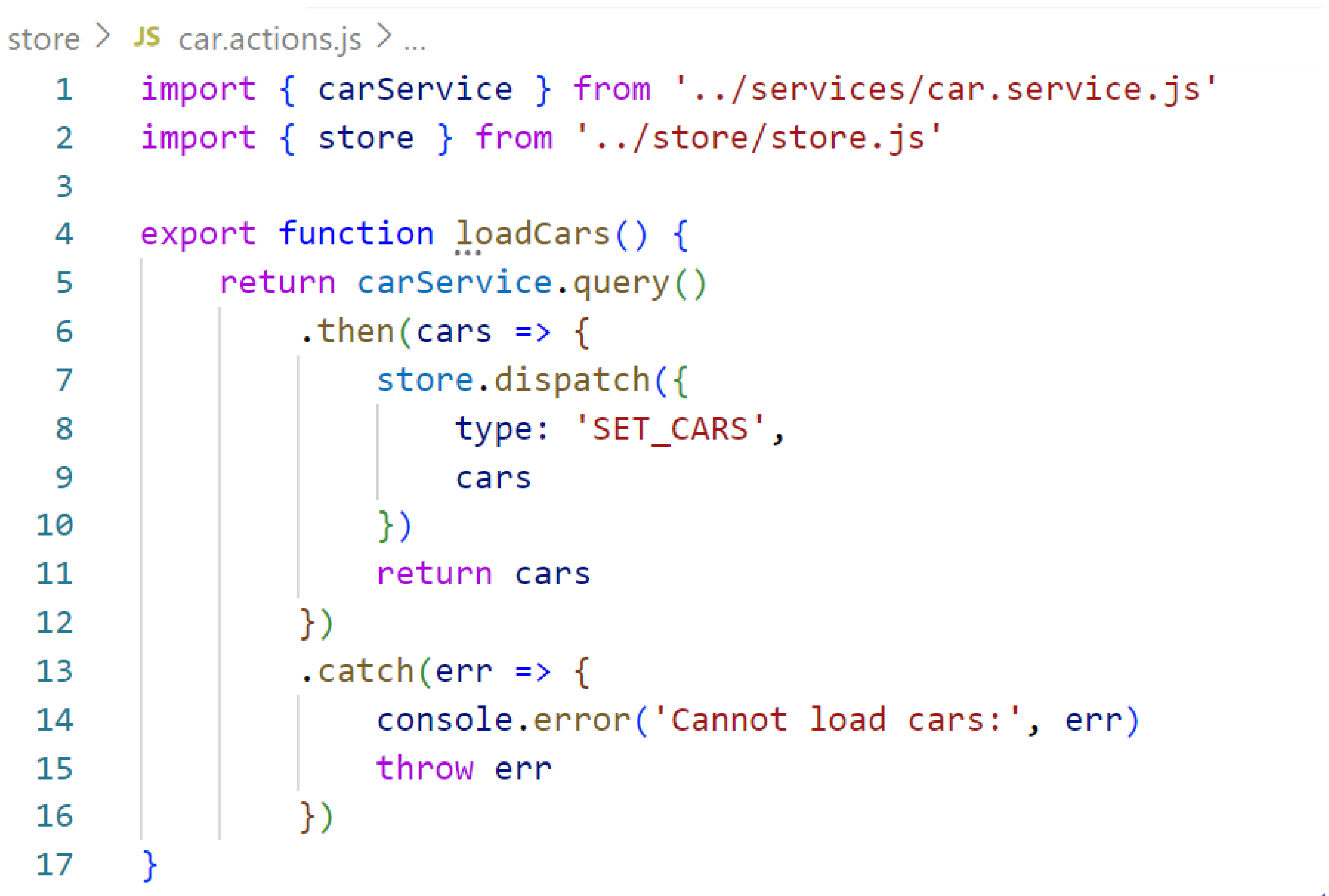
```
store.subscribe(() => {
    console.log('**** Store state changed: ****')
    console.log('storeState:\n', store.getState())
    console.log('*******************************')
})
```



```
**** Store state changed: ****
Command: ▶{type: 'changeCount', val: 10}
storeState:
 ▶Proxy {count: 18, user: {…}, products: Array(2), cart: Array(0)}
*******************************
```

Actions file

We will place our asynchronous calls
in dedicated files:



store > JS car.actions.js > ...

```js
import { carService } from '../services/car.service.js'
import { store } from '../store/store.js'

export function loadCars() {
    return carService.query()
        .then(cars => {
            store.dispatch({
                type: 'SET_CARS',
                cars
            })
            return cars
        })
        .catch(err => {
            console.error('Cannot load cars:', err)
            throw err
        })
}
```

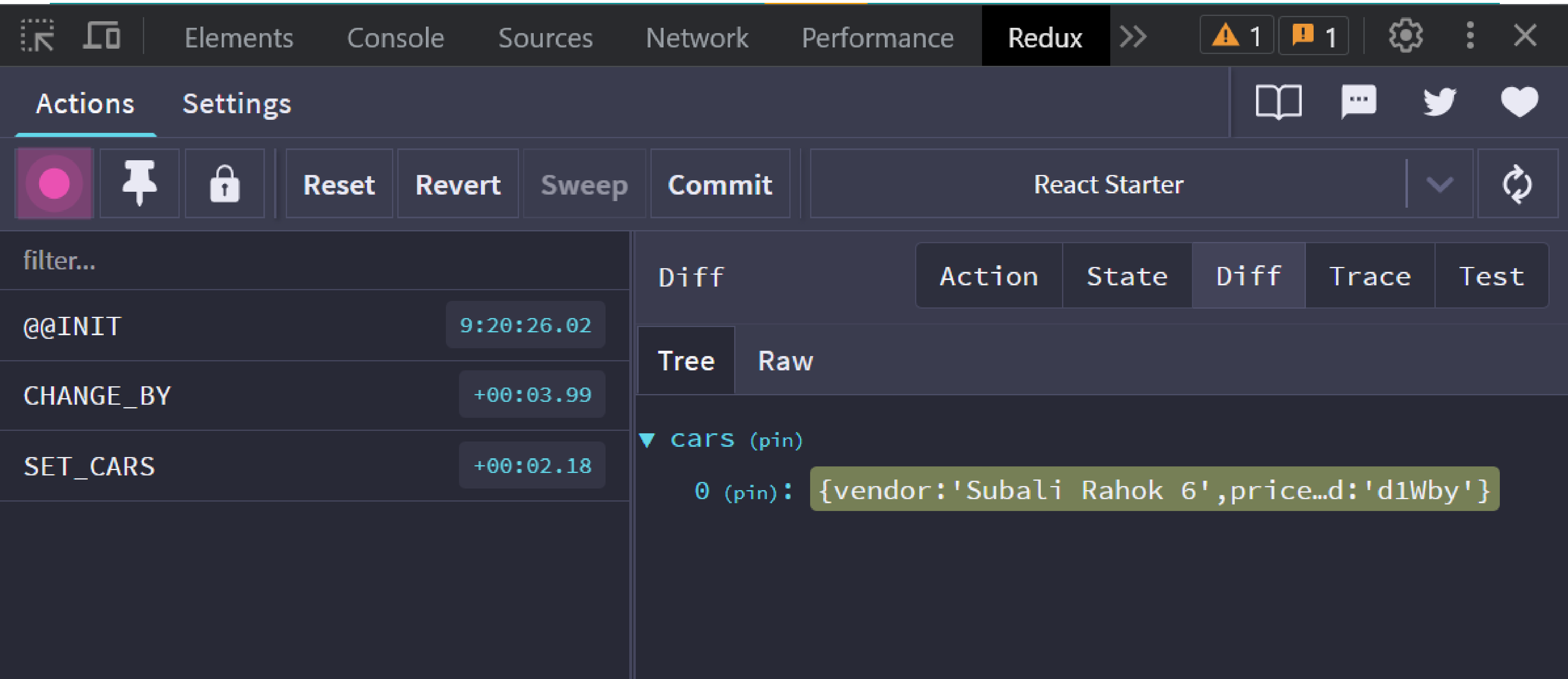
Loading Data

```
import { loadCars} from '../store/car.actions.js'

export function CarIndex() {
    const cars = useSelector(storeState => storeState.cars)

    useEffect(() => {
        loadCars()
    }, [])

    return (
        <div>
            <h3>Cars App</h3>
            <main>
                <pre>{JSON.stringify(cars, null, 2)}</pre>
            </main>
        </div>
    )

}
```

Redux DevTools

Redux DevTools provides action logger and even time travel debugging

```
const middleware = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__()
export const store = createStore(myReducer, middleware)
```



```
@@INIT        9:20:26.02
CHANGE_BY     +00:03.99
SET_CARS      +00:02.18
```

```
▼ cars (pin)
    0 (pin): {vendor:'Subali Rahok 6',price…d:'d1Wby'}
```

Let's add some more command objects:

```
{
    type: 'TOGGLE_CART_SHOWN',
}
{
    type: 'SET_CARS',
    cars
}
{
    type: 'REMOVE_CAR',
    carId
}
{
    type: 'ADD_CAR',
    car
}
{
    type: 'ADD_TO_CART',
    car
}
{
    type: 'REMOVE_FROM_CART',
    carId
}
{
    type: 'CLEAR_CART',
}
```

```
{
    type: 'SET_USER',
    user
}
{
    type: 'SET_USER_SCORE',
    score
}
```

Here are some more actions:

```
export function removeCar(carId) {
    return carService.remove(carId)
        .then(() => {
            store.dispatch({
                type: 'REMOVE_CAR',
                carId
            })
        })
        .catch(err => {
            console.error('Cannot remove car:', err)
            throw err
        })
}
```

Here are some more actions:

```
export function saveCar(car) {
    const type = (car._id) ? 'UPDATE_CAR' : 'ADD_CAR'
    return carService.save(car)
        .then(savedCar => {
            console.log('Saved Car', savedCar)
            store.dispatch({
                type,
                car: savedCar
            })
            return savedCar
        })
        .catch(err => {
            console.error('Cannot save car:', err)
            throw err
        })
}
```

Immutability

Meaning: "being unable to be changed "

Immutability

- When working with redux properly, we work with immutable state
- We do not mutate the state object, instead we return a modified **copy** of it

Mutating array operations

Here we mutate the array,
this is a *big NONO* in redux:

```
// Mutable Updates
function addCar(car) {
    cars.push(car)
}

function removeCar(carId) {
    const idx = cars.findIndex(c => c._id === carId)
    cars.splice(idx, 1)
}

function updateCarPrice(car, price) {
    car.price = price
}
```

None mutating array operations

We create new copies of the objects instead of mutating them

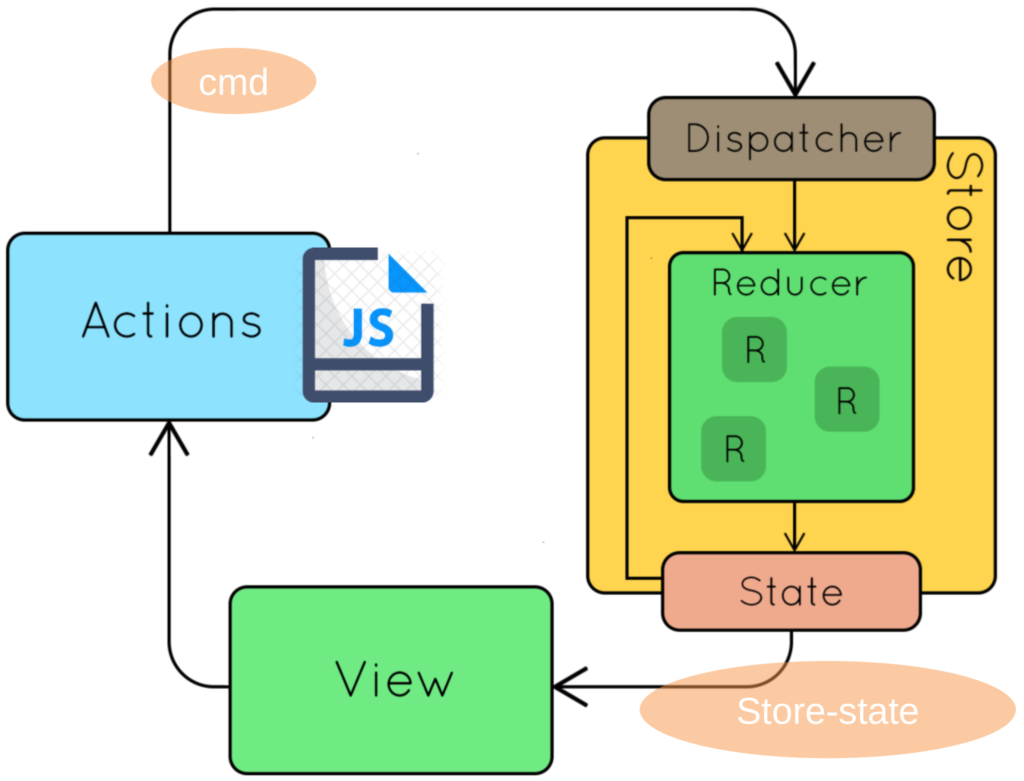
```
// Immutable Updates
function addCar(car) {
    cars = [car, ...cars]
}

function removeCar(carId) {
    cars = cars.filter(c => c._id !== carId)
}

function updateCarPrice(car, price) {
    cars = cars.map(c => {
        if (c._id === car._id) return {...c, price}
        else return c
    })
}
```

Immutability

- In redux we work with immutable state
- See how the reducer returns a modified **copy** of the state:

```
case 'REMOVE_CAR':
    cars = state.cars.filter(car => car._id !== cmd.carId)
    return { ...state, cars }
case 'ADD_CAR':
    cars = [cmd.car, ...state.cars]
    return { ...state, cars }
case 'UPDATE_CAR':
    cars = state.cars.map(currCar =>
        (currCar._id === cmd.car._id) ? cmd.car : currCar)
    return { ...state, cars }
```

Redux – the big picture



- We create the store, providing it with a reducer function
- Components select some store-state and render it
- Asynchronous logic is placed in an actions file
- From which we dispatch command objects to the reducer to update the state

Redux Store is just a change-emitter holding a value

```
function createStore(reducer) {
    let state = reducer(undefined)
    let listeners = []

    function getState() { return state }

    function dispatch(cmd) {
        state = reducer(state, cmd)
        listeners.forEach(listener => listener())
    }

    function subscribe(listener) {
        listeners.push(listener)
        // Return an unsubscribe function
        return () => {
            listeners = listeners.filter(l => l !== listener)
        }
    }
    return {
        getState,
        subscribe,
        dispatch
    }
}
```

Building Bigger Apps

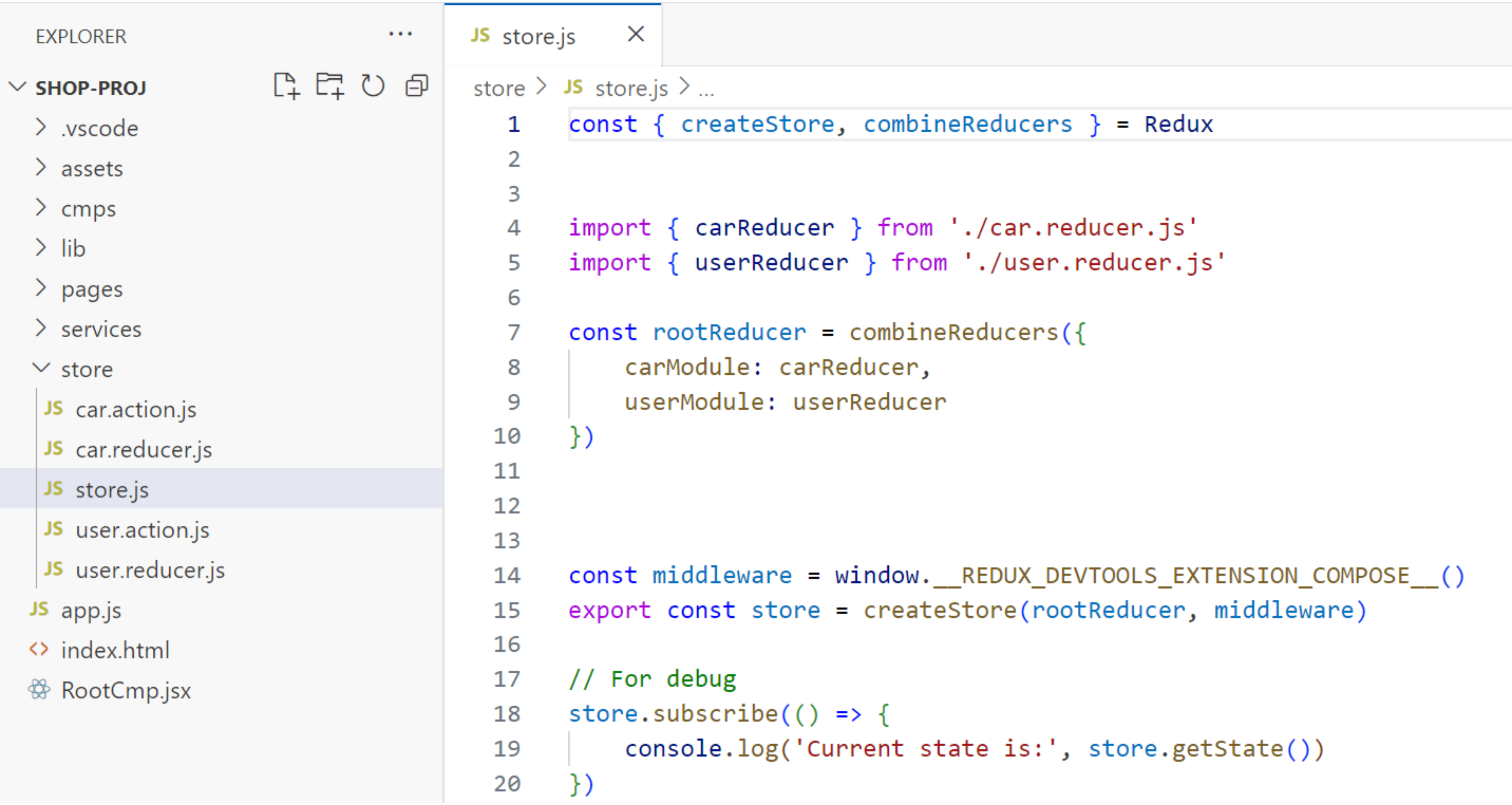
Redux - Modules

Redux - Modules

As app grows, divide the store into modules

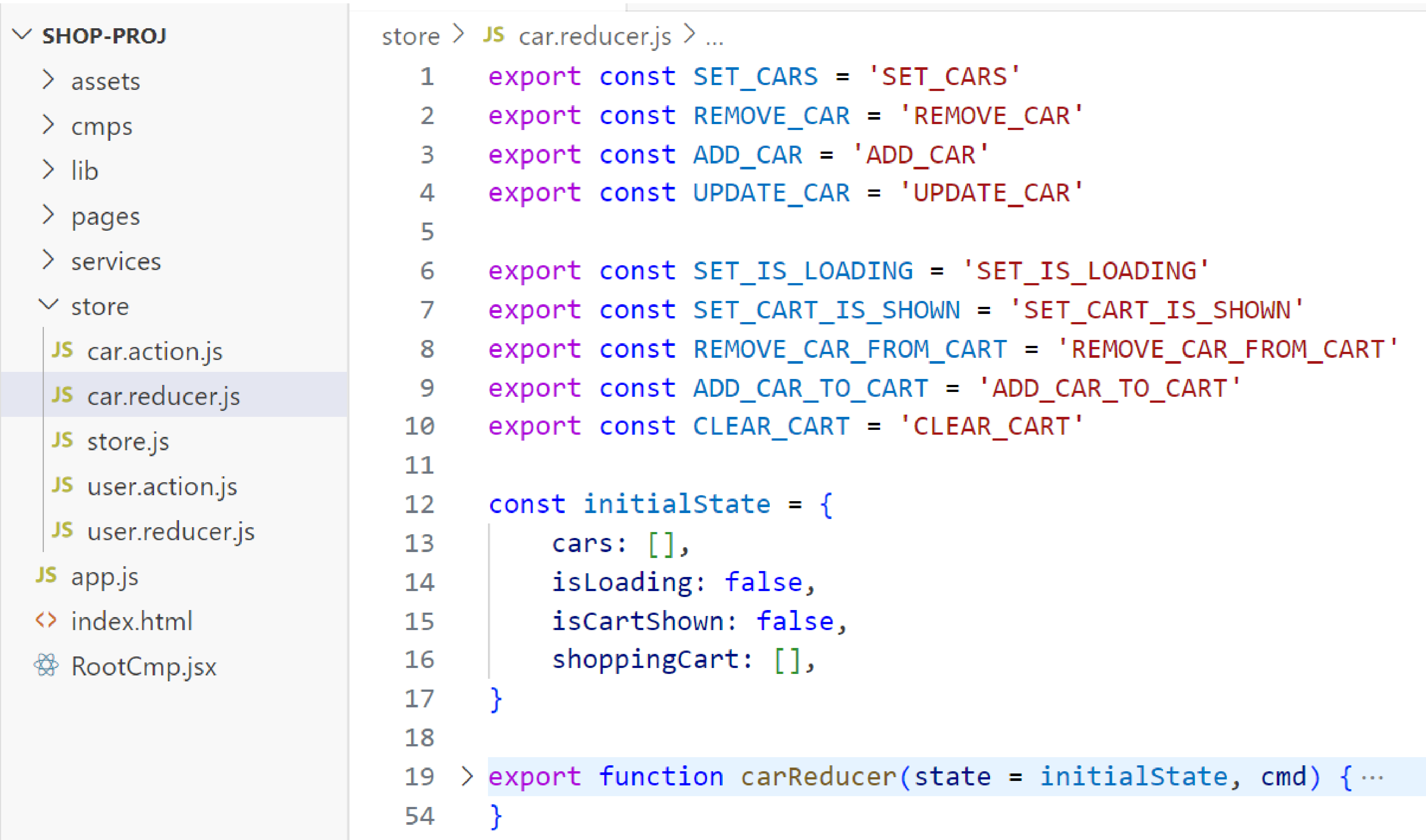
```
import { carReducer } from './car.reducer.js'
import { userReducer } from './user.reducer.js'

const rootReducer = combineReducers({
    carModule: carReducer,
    userModule: userReducer
})

export const store = createStore(rootReducer)
```

Redux - Modules



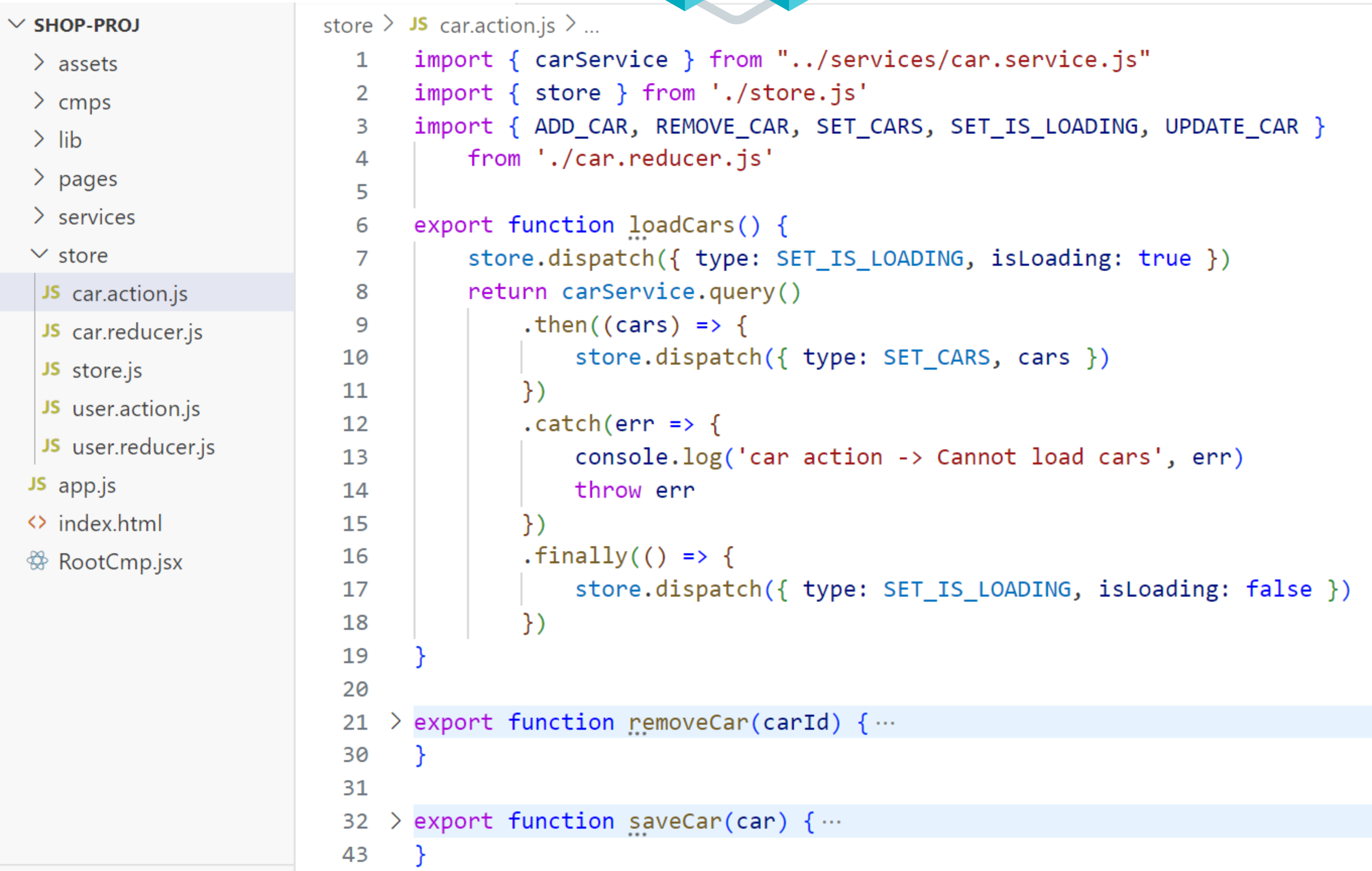
```js
const { createStore, combineReducers } = Redux


import { carReducer } from './car.reducer.js'
import { userReducer } from './user.reducer.js'

const rootReducer = combineReducers({
    carModule: carReducer,
    userModule: userReducer
})



const middleware = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__()
export const store = createStore(rootReducer, middleware)

// For debug
store.subscribe(() => {
    console.log('Current state is:', store.getState())
})
```

Redux - Modules



```
export const SET_CARS = 'SET_CARS'
export const REMOVE_CAR = 'REMOVE_CAR'
export const ADD_CAR = 'ADD_CAR'
export const UPDATE_CAR = 'UPDATE_CAR'

export const SET_IS_LOADING = 'SET_IS_LOADING'
export const SET_CART_IS_SHOWN = 'SET_CART_IS_SHOWN'
export const REMOVE_CAR_FROM_CART = 'REMOVE_CAR_FROM_CART'
export const ADD_CAR_TO_CART = 'ADD_CAR_TO_CART'
export const CLEAR_CART = 'CLEAR_CART'

const initialState = {
    cars: [],
    isLoading: false,
    isCartShown: false,
    shoppingCart: [],
}

export function carReducer(state = initialState, cmd) {…
}
```

Redux - Modules

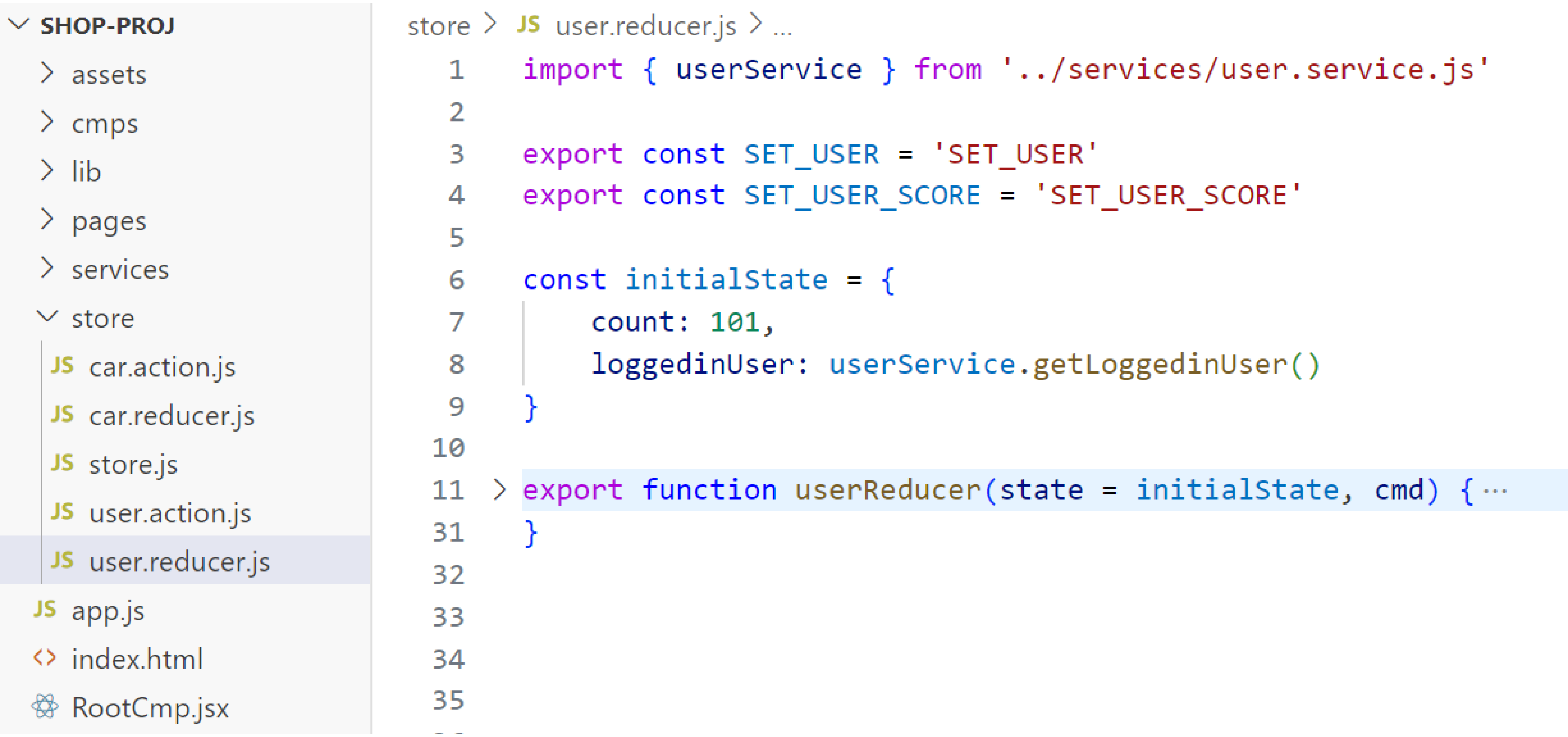


SHOP-PROJ
- assets
- cmps
- lib
- pages
- services
- store
  - car.action.js
  - car.reducer.js
  - store.js
  - user.action.js
  - user.reducer.js
- app.js
- index.html
- RootCmp.jsx

store > JS car.action.js > ...

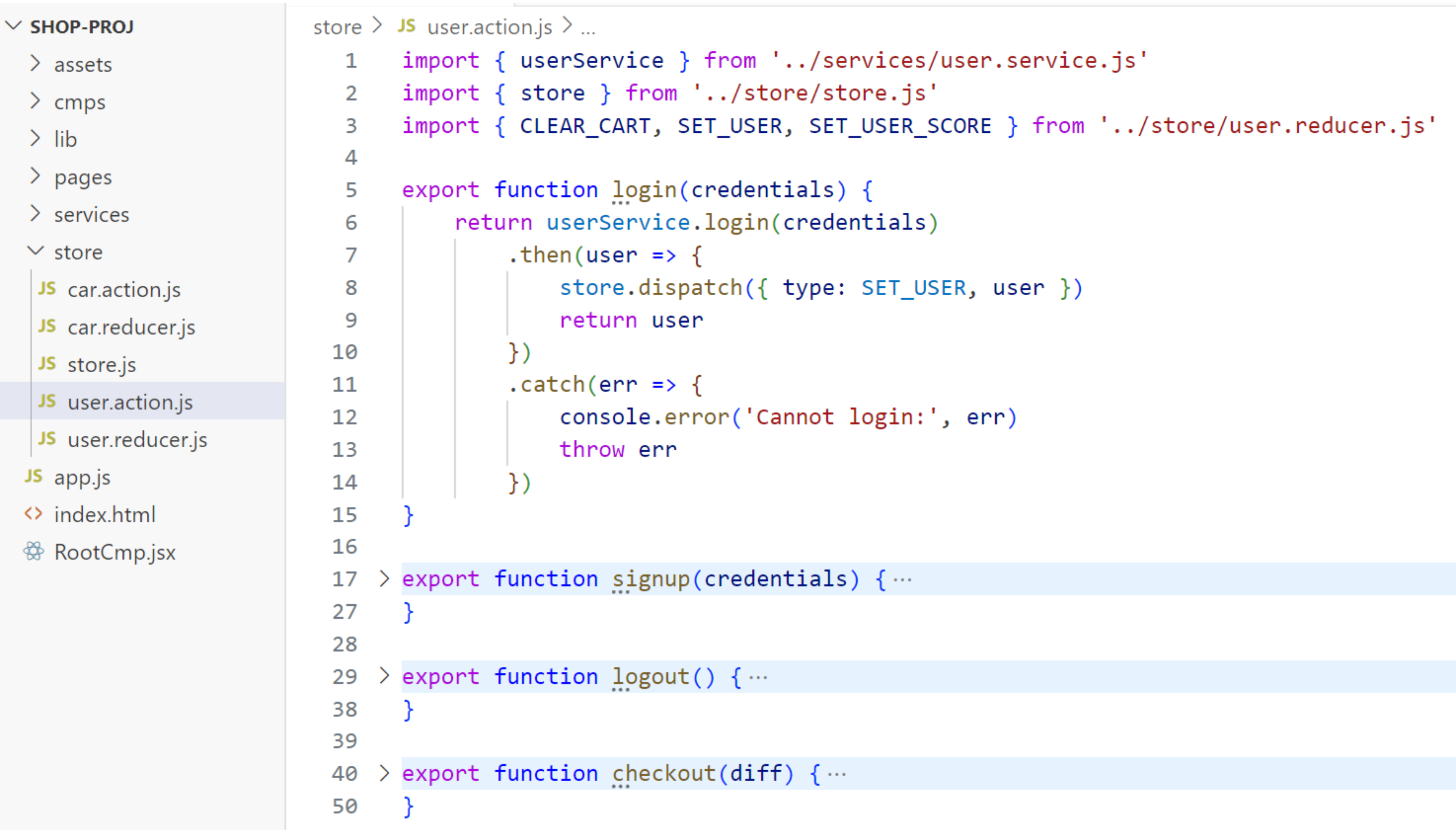
```javascript
import { carService } from "../services/car.service.js"
import { store } from './store.js'
import { ADD_CAR, REMOVE_CAR, SET_CARS, SET_IS_LOADING, UPDATE_CAR }
    from './car.reducer.js'

export function loadCars() {
    store.dispatch({ type: SET_IS_LOADING, isLoading: true })
    return carService.query()
        .then((cars) => {
            store.dispatch({ type: SET_CARS, cars })
        })
        .catch(err => {
            console.log('car action -> Cannot load cars', err)
            throw err
        })
        .finally(() => {
            store.dispatch({ type: SET_IS_LOADING, isLoading: false })
        })
}

export function removeCar(carId) {⋯
}

export function saveCar(car) {⋯
}
```

Redux - Modules



```
import { userService } from '../services/user.service.js'

export const SET_USER = 'SET_USER'
export const SET_USER_SCORE = 'SET_USER_SCORE'

const initialState = {
    count: 101,
    loggedinUser: userService.getLoggedinUser()
}

export function userReducer(state = initialState, cmd) {···
}
```

Redux - Modules



SHOP-PROJ
- assets
- cmps
- lib
- pages
- services
- store
  - car.action.js
  - car.reducer.js
  - store.js
  - user.action.js
  - user.reducer.js
- app.js
- index.html
- RootCmp.jsx

store > user.action.js > ...

```javascript
import { userService } from '../services/user.service.js'
import { store } from '../store/store.js'
import { CLEAR_CART, SET_USER, SET_USER_SCORE } from '../store/user.reducer.js'

export function login(credentials) {
    return userService.login(credentials)
        .then(user => {
            store.dispatch({ type: SET_USER, user })
            return user
        })
        .catch(err => {
            console.error('Cannot login:', err)
            throw err
        })
}

export function signup(credentials) {···
}

export function logout() {···
}

export function checkout(diff) {···
}
```

Our store is ready

```javascript
import { createStore, combineReducers } from 'redux'

import { carReducer } from './car.reducer.js'
import { userReducer } from './user.reducer.js'

const rootReducer = combineReducers({
    carModule: carReducer,
    userModule: userReducer
})


const middleware = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__()
export const store = createStore(rootReducer, middleware)


store.subscribe(() => {
    console.log('**** Store state changed: ****')
    console.log('storeState:\n', store.getState())
    console.log('******************************')
})
```

a comment about Entity edit

In our Edit page, when editing some entity we can either:

- Get it directly from the service
- Place in the store and create a copy for editing (we cannot edit it directly)

The key question is: Is that a shared state?

Updating an entity

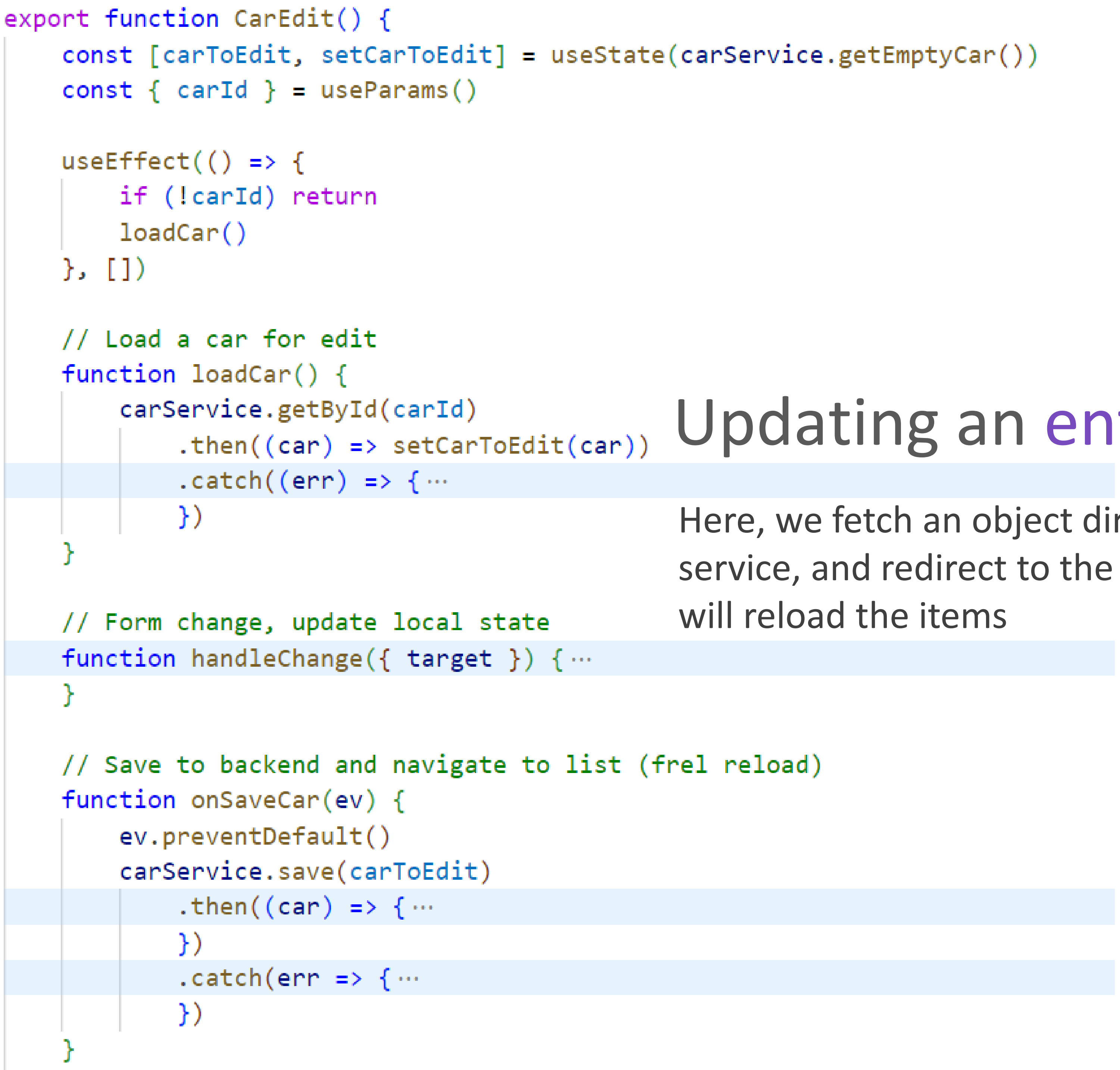
Here, we need to update store-state so we create a copy:

```
// In the component
function onEditCar(car) {
    const price = +prompt('New price?', car.price)
    if (!price || price === car.price) return

    const carToSave = { ...car, price }
    saveCar(carToSave)
        .then((savedCar) => {
            showSuccessMsg(`Car updated to price: $${savedCar.price}`)
        })
        .catch(err => {
            showErrorMsg('Cannot update car')
        })
}
```

```
// in car.actions
export function saveCar(car) {
    const type = car._id ? UPDATE_CAR : ADD_CAR
    return carService.save(car)
        .then(savedCar => {
            store.dispatch({ type, car: savedCar })
            return savedCar
        })
        .catch(err => {
            console.log('car action -> Cannot save car', err)
            throw err
        })
}
```

Updating an entity

Here, we fetch an object directly from the service, and redirect to the list page, that will reload the items



```
export function CarEdit() {
    const [carToEdit, setCarToEdit] = useState(carService.getEmptyCar())
    const { carId } = useParams()

    useEffect(() => {
        if (!carId) return
        loadCar()
    }, [])

    // Load a car for edit
    function loadCar() {
        carService.getById(carId)
            .then((car) => setCarToEdit(car))
            .catch((err) => {…
            })
    }

    // Form change, update local state
    function handleChange({ target }) {…
    }

    // Save to backend and navigate to list (frel reload)
    function onSaveCar(ev) {
        ev.preventDefault()
        carService.save(carToEdit)
            .then((car) => {…
            })
            .catch(err => {…
            })
    }
```

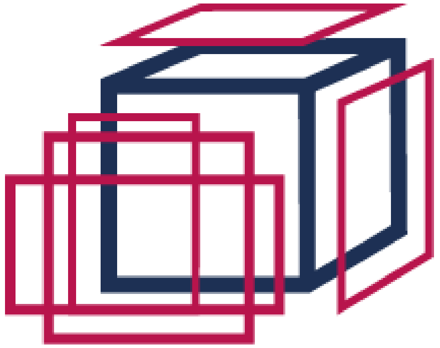
optimistic strategy

To maximize the user experience (UX)
we can use the following strategy:

```
// Normal strategy:
export function removeCar(carId) {
    return carService.remove(carId)
        .then(() => {
            store.dispatch({ type: REMOVE_CAR, carId })
        })
        .catch(err => {
            console.log('car action -> Cannot remove car', err)
            throw err
        })
}

// Optimistic strategy:
export function removeCarOptimistic(carId) {
    store.dispatch({ type: REMOVE_CAR, carId })
    return carService.remove(carId)
        .catch(err => {
            store.dispatch({ type: REMOVE_CAR_UNDO, carId })
            console.log('car action -> Cannot remove car', err)
            throw err
        })
}
```

local state is still allowed

- Components can still have some local state
- But shared-mutable-state belong in the store

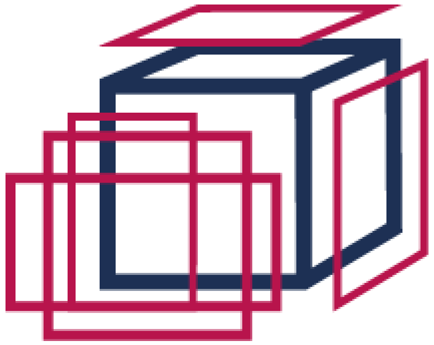


Summary

Redux is a popular and elegant state management library

- The store is a single source of truth.
- It holds one object tree which is the entire shared state of your app
- To mutate the state tree we dispatch a command.
- A command is an object describing what need to happen.
- Commands are passed to reducers - which changes the state accordingly.

Why Bother?

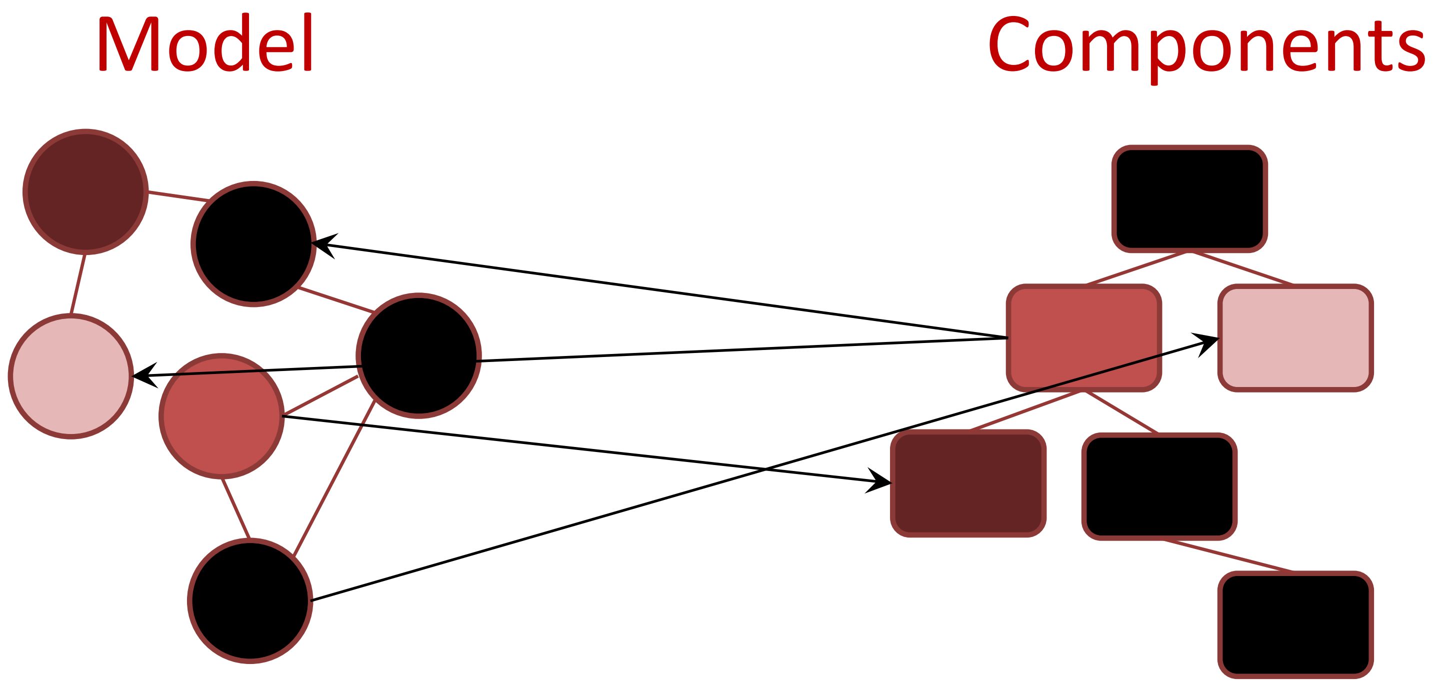


- State Management is about constraints:
  - – Single state tree
  - – Command objects to describe updates
  - – Reducers as pure functions that apply the updates

- But what do we get back?

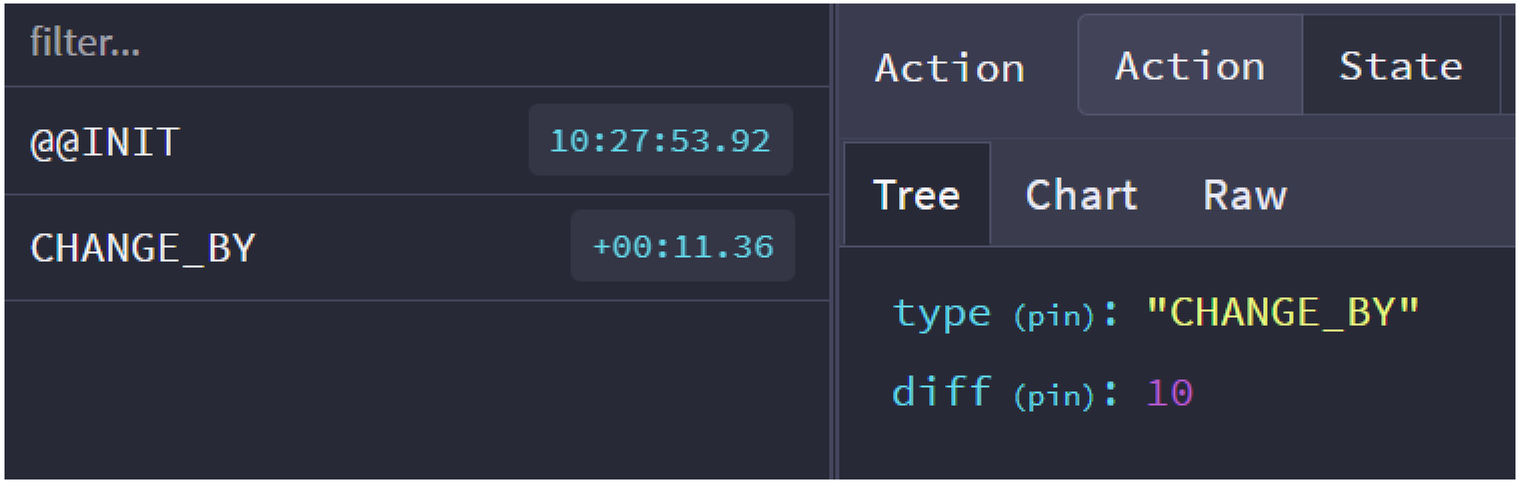
Store State is a single source of truth

When the application grows, syncing changes and rendering effectively become crucial



Debug workflow

- Log commands and states
- Find the bad state
- Check the command (action)
- Fix the reducer

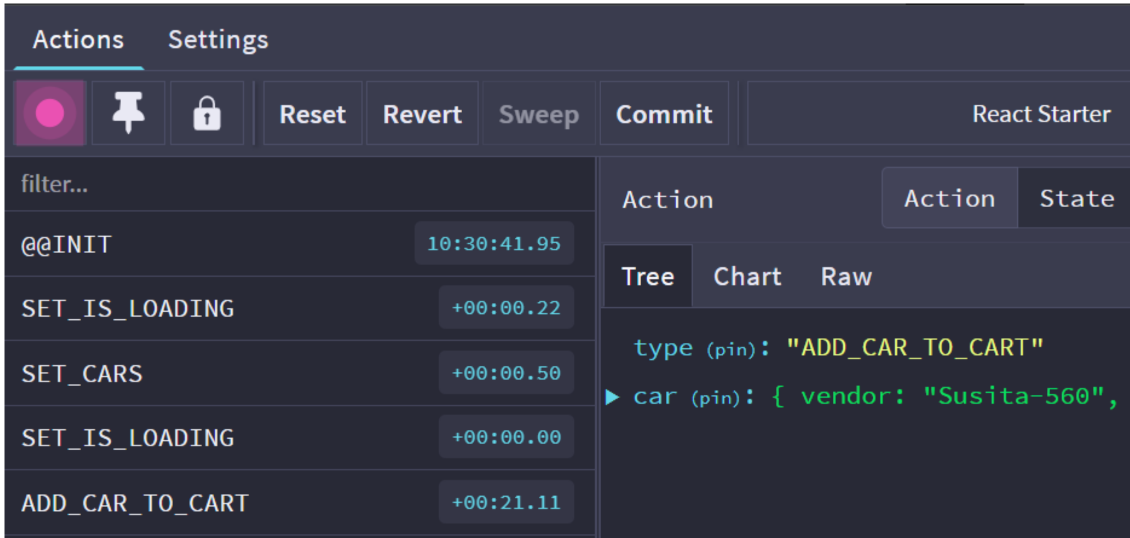


```
filter...
@@INIT        10:27:53.92
CHANGE_BY     +00:11.36
```

```
type (pin): "CHANGE_BY"
diff (pin): 10
```

Everything is Data

- Recording user sessions
  - Good place for collecting analytics
- Error handling
  - Try-catch -> send state and command to logging server
- Optimistic mutations
  – Update the UI immediately, If we get an error from server, undo.
- Collaborative editing

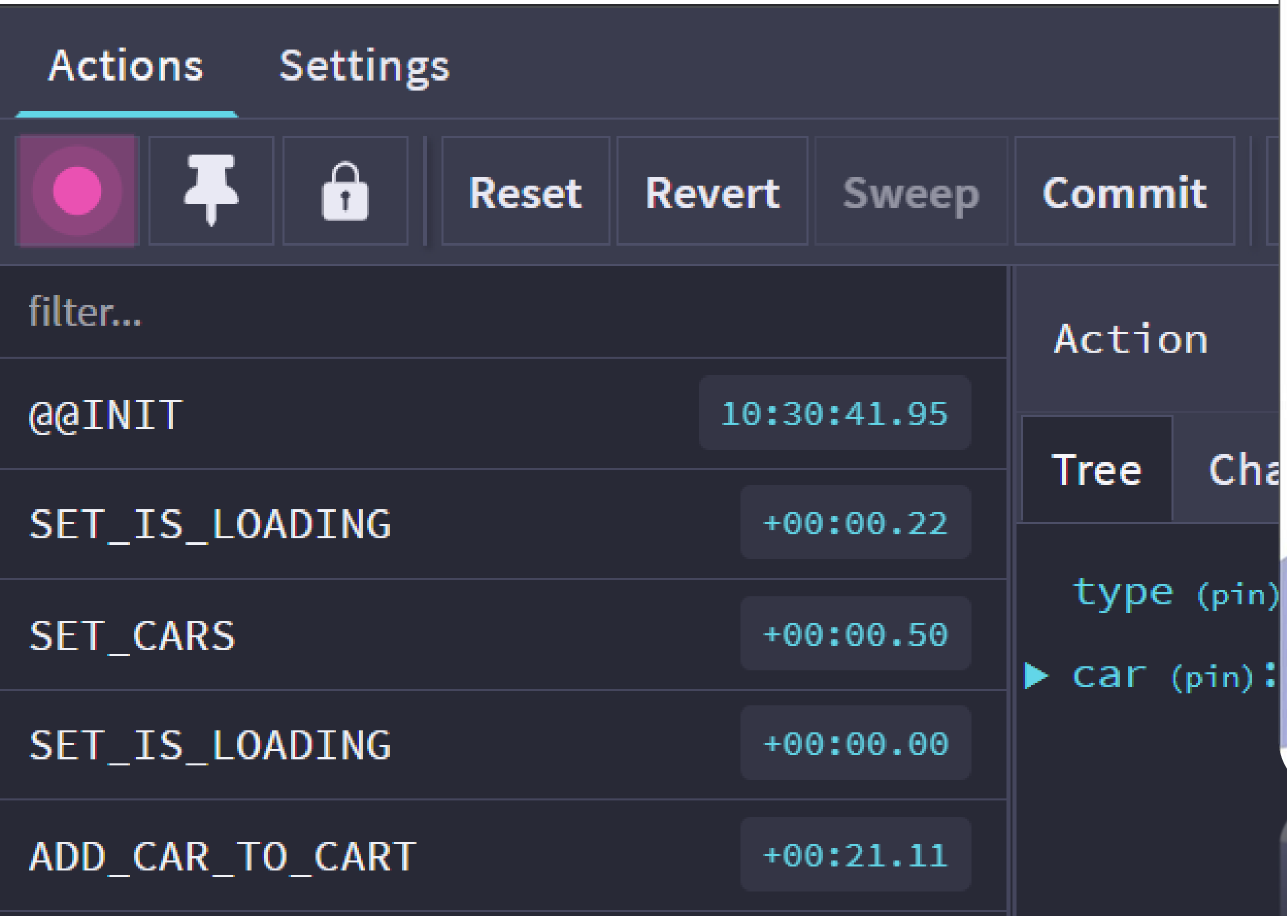


```
@@INIT              10:30:41.95
SET_IS_LOADING      +00:00.22
SET_CARS            +00:00.50
SET_IS_LOADING      +00:00.00
ADD_CAR_TO_CART     +00:21.11
```

```
type (pin): "ADD_CAR_TO_CART"
▶ car (pin): { vendor: "Susita-560",
```

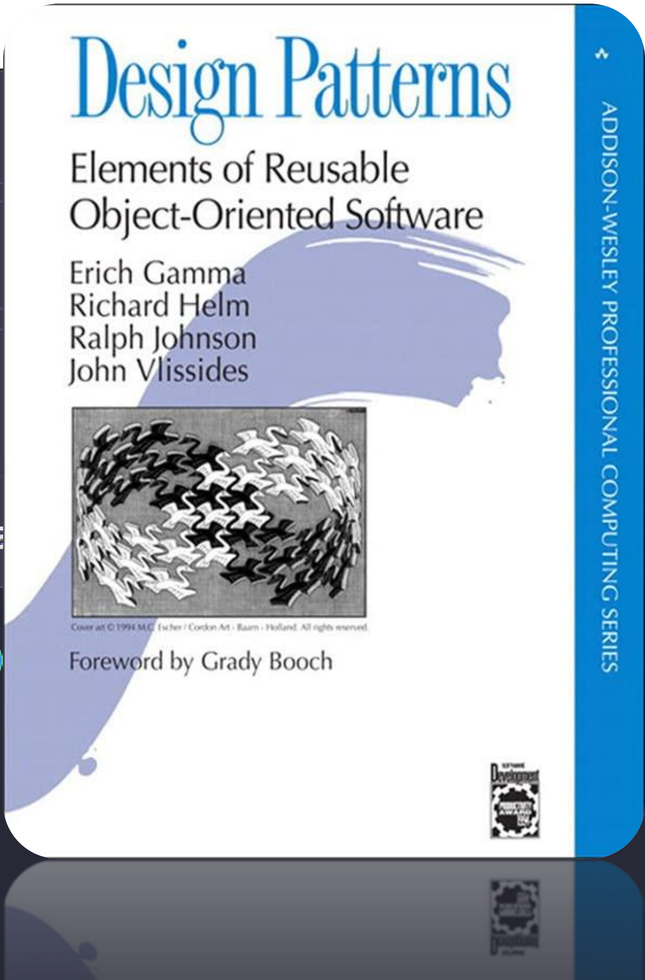
Representing a command with an Object

It's an implementation of the
Command Design Pattern



```
@@INIT              10:30:41.95
SET_IS_LOADING      +00:00.22
SET_CARS            +00:00.50
SET_IS_LOADING      +00:00.00
ADD_CAR_TO_CART     +00:21.11
```

```
type (pin)
▶ car (pin):
```


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Sometimes

Constraints

give you

Features

ReactJS

State Management

With Redux

{coding
academy