# b Many reducers



Let's continue our work with the simplified Redux version of our notes application.

To ease our development, let's change our reducer so that the store gets initialized with a state that contains a couple of notes:

```
const initialState = [
  {
    content: 'reducer defines how redux store works',
    important: true,
    id: 1,
  },
  {
    content: 'state of store can contain any data',
    important: false,
    id: 2,
  },
]

const noteReducer = (state = initialState, action) => {
  // ...
}

// ...
export default noteReducer
```

## Store with complex state

Let's implement filtering for the notes that are displayed to the user. The user interface for the filters will be implemented with radio buttons:


Let's start with a very simple and straightforward implementation:

```
import NewNote from './components/NewNote'
import Notes from './components/Notes'

const App = () => {
  const filterSelected = (value) => {
    console.log(value)
  }

  return (
    <div>
      <NewNote />
      <div>
        all          <input type="radio" name="filter"
          onChange={() => filterSelected('ALL')} />
        important    <input type="radio" name="filter"
          onChange={() => filterSelected('IMPORTANT')} />
        nonimportant <input type="radio" name="filter"
          onChange={() => filterSelected('NONIMPORTANT')} />
      </div>
      <Notes />
    </div>
  )
}
```

Since the *name* attribute of all the radio buttons is the same, they form a *button group* where only one option can be selected.

The buttons have a change handler that currently only prints the string associated with the clicked button to the console.

We decide to implement the filter functionality by storing *the value of the filter* in the redux store in addition to the notes themselves. The state of the store should look like this after making these changes:

```
{
  notes: [
    { content: 'reducer defines how redux store works', important: true, id: 1},
    { content: 'state of store can contain any data', important: false, id: 2}
  ],
  filter: 'IMPORTANT'
}
```

Only the array of notes is stored in the state of the current implementation of our application. In the new implementation, the state object has two properties, *notes* that contains the array of notes and *filter* that contains a string indicating which notes should be displayed to the user.

## Combined reducers

We could modify our current reducer to deal with the new shape of the state. However, a better solution in this situation is to define a new separate reducer for the state of the filter:

```
const filterReducer = (state = 'ALL', action) => {
  switch (action.type) {
    case 'SET_FILTER':
      return action.payload
    default:
      return state
  }
}
```

The actions for changing the state of the filter look like this:

```
{
  type: 'SET_FILTER',
  payload: 'IMPORTANT'
}
```

Let's also create a new `action creator` function. We will write the code for the action creator in a new *src/reducers/filterReducer.js* module:

```
const filterReducer = (state = 'ALL', action) => {
  // ...
}

export const filterChange = filter => {
  return {
    type: 'SET_FILTER',
    payload: filter,
  }
}

export default filterReducer
```

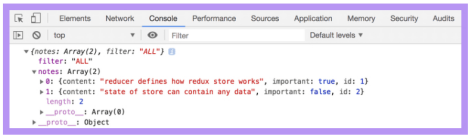
We can create the actual reducer for our application by combining the two existing reducers with the combineReducers function.

Let's define the combined reducer in the *main.jsx* file:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { createStore, combineReducers } from 'redux'
import { Provider } from 'react-redux'
import App from './App'

import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'

const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer
})

const store = createStore(reducer)

console.log(store.getState())

/*
ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <App />
  </Provider>
)*/

ReactDOM.createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <div />
  </Provider>
)
```

Since our application breaks completely at this point, we render an empty *div* element instead of the *App* component.

The state of the store gets printed to the console:




As we can see from the output, the store has the exact shape we wanted it to!

Let's take a closer look at how the combined reducer is created:

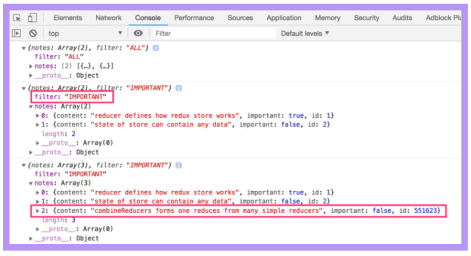
```
const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer,
})
```

The state of the store defined by the reducer above is an object with two properties: *notes* and *filter*. The value of the *notes* property is defined by the *noteReducer*, which does not have to deal with the other properties of the state. Likewise, the *filter* property is managed by the *filterReducer*.

Before we make more changes to the code, let's take a look at how different actions change the state of the store defined by the combined reducer. Let's add the following to the *main.jsx* file:

```
import { createNote } from './reducers/noteReducer'
import { filterChange } from './reducers/filterReducer'
// ...
store.subscribe(() => console.log(store.getState()))
store.dispatch(filterChange('IMPORTANT'))
store.dispatch(createNote('combineReducers forms one reducer from many simple
reducers'))
```

By simulating the creation of a note and changing the state of the filter in this fashion, the state of the store gets logged to the console after every change that is made to the store:

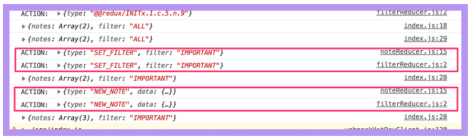



At this point, it is good to become aware of a tiny but important detail. If we add a console log statement *to the beginning of both reducers*:

```
const filterReducer = (state = 'ALL', action) => {
  console.log('ACTION: ', action)
  // ...
```

```
}
```

Based on the console output one might get the impression that every action gets duplicated:




Is there a bug in our code? No. The combined reducer works in such a way that every *action* gets handled in *every* part of the combined reducer. Typically only one reducer is interested in any given action, but there are situations where multiple reducers change their respective parts of the state based on the same action.

## Finishing the filters

Let's finish the application so that it uses the combined reducer. We start by changing the rendering of the application and hooking up the store to the application in the *main.jsx* file:

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

Next, let's fix a bug that is caused by the code expecting the application store to be an array of notes:


It's an easy fix. Because the notes are in the store's field *notes*, we only have to make a little change to the selector function:

```
const Notes = () => {
  const dispatch = useDispatch()
  const notes = useSelector(state => state.notes)

  return(
    <ul>
      {notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() =>
            dispatch(toggleImportanceOf(note.id))
          }
        />
      )}
    </ul>
  )
}
```

Previously the selector function returned the whole state of the store:

```
const notes = useSelector(state => state)
```

And now it returns only its field *notes*

```
const notes = useSelector(state => state.notes)
```

Let's extract the visibility filter into its own *src/components/VisibilityFilter.jsx* component:

```
import { filterChange } from '../reducers/filterReducer'
import { useDispatch } from 'react-redux'

const VisibilityFilter = (props) => {
  const dispatch = useDispatch()

  return (
    <div>
      all
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('ALL'))}
      />
      important
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('IMPORTANT'))}
      />
      nonimportant
      <input
        type="radio"
        name="filter"
        onChange={() => dispatch(filterChange('NONIMPORTANT'))}
      />
    </div>
  )
}

export default VisibilityFilter
```

With the new component *App* can be simplified as follows:

```
import Notes from './components/Notes'
import NewNote from './components/NewNote'
import VisibilityFilter from './components/VisibilityFilter'

const App = () => {
  return (
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
    </div>
  )
}

export default App
```

The implementation is rather straightforward. Clicking the different radio buttons changes the state of the store's *filter* property.

Let's change the *Notes* component to incorporate the filter:

```
const Notes = () => {                                                    copy
  const dispatch = useDispatch()
  const notes = useSelector(state => {
    if ( state.filter === 'ALL' ) {
      return state.notes
    }
    return state.filter  === 'IMPORTANT'
      ? state.notes.filter(note => note.important)
      : state.notes.filter(note => !note.important)
  })

  return(
    <ul>
      {notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() =>
            dispatch(toggleImportanceOf(note.id))
          }
        />
      )}
    </ul>
  )
```

We only make changes to the selector function, which used to be

```
useSelector(state => state.notes)                                        copy
```

Let's simplify the selector by destructuring the fields from the state it receives as a parameter:

```
const notes = useSelector(({ filter, notes }) => {                       copy
  if ( filter === 'ALL' ) {
    return notes
  }
  return filter  === 'IMPORTANT'
    ? notes.filter(note => note.important)
    : notes.filter(note => !note.important)
})
```

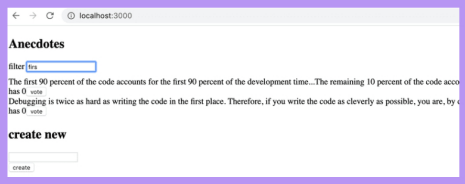
There is a slight cosmetic flaw in our application. Even though the filter is set to *ALL* by default, the associated radio button is not selected. Naturally, this issue can be fixed, but since this is an unpleasant but ultimately harmless bug we will save the fix for later.

The current version of the application can be found on GitHub, branch *part6-2*.

## Exercise 6.9

### 6.9 Better anecdotes, step7

Implement filtering for the anecdotes that are displayed to the user.




Store the state of the filter in the redux store. It is recommended to create a new reducer, action creators, and a combined reducer for the store using the *combineReducers* function.

Create a new *Filter* component for displaying the filter. You can use the following code as a template for the component:

```
const Filter = () => {                                                   copy
  const handleChange = (event) => {
    // input-field value is in variable event.target.value
  }
  const style = {
    marginBottom: 10
  }

  return (
    <div style={style}>
      filter <input onChange={handleChange} />
    </div>
  )
}

export default Filter
```

## Redux Toolkit

As we have seen so far, Redux's configuration and state management implementation requires quite a lot of effort. This is manifested for example in the reducer and action creator-related code which has somewhat repetitive boilerplate code. Redux Toolkit is a library that solves these common Redux-related problems. The library for example greatly simplifies the configuration of the Redux store and offers a large variety of tools to ease state management.

Let's start using Redux Toolkit in our application by refactoring the existing code. First, we will need to install the library:

```
npm install @reduxjs/toolkit                                             copy
```

Next, open the *main.jsx* file which currently creates the Redux store. Instead of Redux's `createStore` function, let's create the store using Redux Toolkit's configureStore function:

```
import React from 'react'                                                copy
import ReactDOM from 'react-dom/client'
import { Provider } from 'react-redux'
import { configureStore } from '@reduxjs/toolkit'
```

```
import App from './App'

import noteReducer from './reducers/noteReducer'
import filterReducer from './reducers/filterReducer'

const store = configureStore({
  reducer: {
    notes: noteReducer,
    filter: filterReducer
  }
})

console.log(store.getState())

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

We already got rid of a few lines of code now that we don't need the `combineReducers` function to create the reducer for the store. We will soon see that the `configureStore` function has many additional benefits such as the effortless integration of development tools and many commonly used libraries without the need for additional configuration.

Let's move on to refactoring the reducers, which brings forth the benefits of the Redux Toolkit. With Redux Toolkit, we can easily create reducer and related action creators using the createSlice function. We can use the `createSlice` function to refactor the reducer and action creators in the *reducers/noteReducer.js* file in the following manner:

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

const initialState = [
  {
    content: 'reducer defines how redux store works',
    important: true,
    id: 1,
  },
  {
    content: 'state of store can contain any data',
    important: false,
    id: 2,
  },
]

const generateId = () =>
  Number((Math.random() * 1000000).toFixed(0))

const noteSlice = createSlice({
  name: 'notes',
  initialState,
  reducers: {
    createNote(state, action) {
      const content = action.payload
      state.push({
        content,
        important: false,
        id: generateId(),
      })
    },
    toggleImportanceOf(state, action) {
      const id = action.payload
      const noteToChange = state.find(n => n.id === id)
      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }
      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    }
  },
})
```

The `createSlice` function's `name` parameter defines the prefix which is used in the action's type values. For example, the `createNote` action defined later will have the type value of `notes/createNote`. It is a good practice to give the parameter a value which is unique among the reducers. This way there won't be unexpected collisions between the application's action type values. The `initialState` parameter defines the reducer's initial state. The `reducers` parameter takes the reducer itself as an object, of which functions handle state changes caused by certain actions. Note that the `action.payload` in the function contains the argument provided by calling the action creator:

```
dispatch(createNote('Redux Toolkit is awesome!'))
```

This dispatch call responds to dispatching the following object:

```
dispatch({ type: 'notes/createNote', payload: 'Redux Toolkit is awesome!' })
```

If you followed closely, you might have noticed that inside the `createNote` action, there seems to happen something that violates the reducers' immutability principle mentioned earlier:

```
createNote(state, action) {
  const content = action.payload

  state.push({
    content,
    important: false,
    id: generateId(),
  })
}
```

We are mutating `state` argument's array by calling the `push` method instead of returning a new instance of the array. What's this all about?

Redux Toolkit utilizes the Immer library with reducers created by `createSlice` function, which makes it possible to mutate the `state` argument inside the reducer. Immer uses the mutated state to produce a new, immutable state and thus the state changes remain immutable. Note that `state` can be changed without "mutating" it, as we have done with the `toggleImportanceOf` action. In this case, the function *returns* the new state. Nevertheless mutating the state will often come in handy especially when a complex state needs to be updated.

The `createSlice` function returns an object containing the reducer as well as the action creators defined by the `reducers` parameter. The reducer can be accessed by the `noteSlice.reducer` property, whereas the action creators by the `noteSlice.actions` property. We can produce the file's exports in the following way:

```
const noteSlice = createSlice(/* ... */)

export const { createNote, toggleImportanceOf } = noteSlice.actions
export default noteSlice.reducer
```

The imports in other files will work just as they did before:

```
import noteReducer, { createNote, toggleImportanceOf } from
'./reducers/noteReducer'
```

We need to alter the action type names in the tests due to the conventions of ReduxToolkit:

```
import noteReducer from './noteReducer'
import deepFreeze from 'deep-freeze'

describe('noteReducer', () => {
  test('returns new state with action notes/createNote', () => {
    const state = []
    const action = {
      type: 'notes/createNote',
      payload: 'the app state is in redux store',
    }

    deepFreeze(state)
    const newState = noteReducer(state, action)

    expect(newState).toHaveLength(1)
    expect(newState.map(s => s.content)).toContainEqual(action.payload)
  })

  test('returns new state with action notes/toggleImportanceOf', () => {
    const state = [
      {
        content: 'the app state is in redux store',
        important: true,
        id: 1
      },
      {
        content: 'state changes are made with actions',
        important: false,
        id: 2
      }]

    const action = {
      type: 'notes/toggleImportanceOf',
      payload: 2
    }

    deepFreeze(state)
    const newState = noteReducer(state, action)

    expect(newState).toHaveLength(2)

    expect(newState).toContainEqual(state[0])

    expect(newState).toContainEqual({
      content: 'state changes are made with actions',
      important: true,
      id: 2
    })
  })
})
```

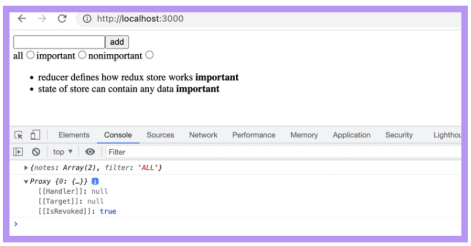
## Redux Toolkit and console.log

As we have learned, console.log is an extremely powerful tool; it often saves us from trouble.

Let's try to print the state of the Redux Store to the console in the middle of the reducer created with the function createSlice:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState,
  reducers: {
    // ...
    toggleImportanceOf(state, action) {
      const id = action.payload

      const noteToChange = state.find(n => n.id === id)

      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }

      console.log(state)

      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    }
  },
})
```

The following is printed to the console

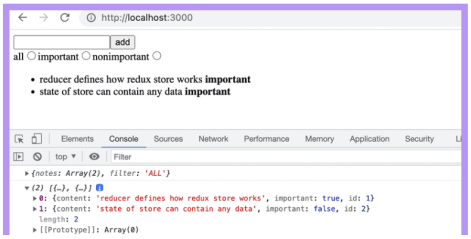



The output is interesting but not very useful. This is about the previously mentioned Immer library used by the Redux Toolkit, which is now used internally to save the state of the Store.

The status can be converted to a human-readable format, e.g. by converting it to a string and back to a JavaScript object as follows:

```
console.log(JSON.parse(JSON.stringify(state)))
```

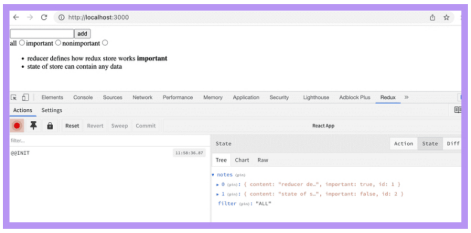
Console output is now human readable



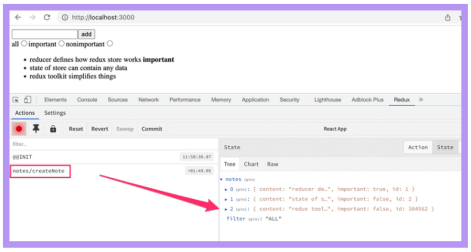
### Redux DevTools

Redux DevTools is a Chrome addon that offers useful development tools for Redux. It can be used for example to inspect the Redux store's state and dispatch actions through the browser's console. When the store is created using Redux Toolkit's `configureStore` function, no additional configuration is needed for Redux DevTools to work.

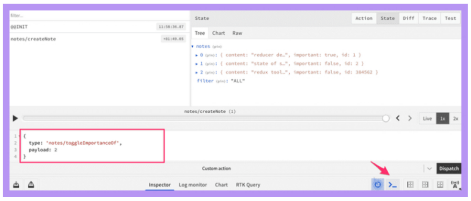
Once the addon is installed, clicking the *Redux* tab in the browser's console should open the development tools:



You can inspect how dispatching a certain action changes the state by clicking the action:



It is also possible to dispatch actions to the store using the development tools:



You can find the code for our current application in its entirety in the *part6-3* branch of this GitHub repository.

### Exercises 6.10.-6.13.

Let's continue working on the anecdote application using Redux that we started in exercise 6.3.

#### 6.10 Better anecdotes, step8

Install Redux Toolkit for the project. Move the Redux store creation into the file *store.js* and use Redux Toolkit's `configureStore` to create the store.

Change the definition of the *filter reducer and action creators* to use the Redux Toolkit's `createSlice` function.

Also, start using Redux DevTools to debug the application's state easier.

#### 6.11 Better anecdotes, step9

Change also the definition of the *anecdote reducer and action creators* to use the Redux Toolkit's `createSlice` function.

#### 6.12 Better anecdotes, step10

The application has a ready-made body for the *Notification* component:

```
const Notification = () => {
  const style = {
    border: 'solid',
    padding: 10,
    borderWidth: 1
  }
  return (
    <div style={style}>
      render here notification...
    </div>
  )
}

export default Notification
```

Extend the component so that it renders the message stored in the Redux store, making the component take the following form:

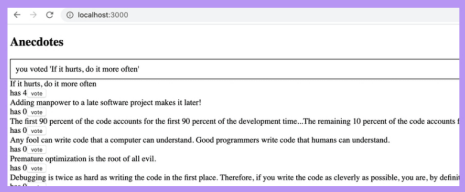
```
import { useSelector } from 'react-redux'

const Notification = () => {
  const notification = useSelector(/* something here */)
  const style = {
    border: 'solid',
    padding: 10,
    borderWidth: 1
  }
  return (
    <div style={style}>
      {notification}
    </div>
  )
}
```

You will have to make changes to the application's existing reducer. Create a separate reducer for the new functionality by using the Redux Toolkit's `createSlice` function.

The application does not have to use the *Notification* component intelligently at this point in the exercises. It is enough for the application to display the initial value set for the message in the *notificationReducer*.

### 6.13 Better anecdotes, step11

Extend the application so that it uses the *Notification* component to display a message for five seconds when the user votes for an anecdote or creates a new anecdote:




It's recommended to create separate action creators for setting and removing notifications.

Propose changes to material

Part 6a
**Previous part**

Part 6c
**Next part**