b Many reducers

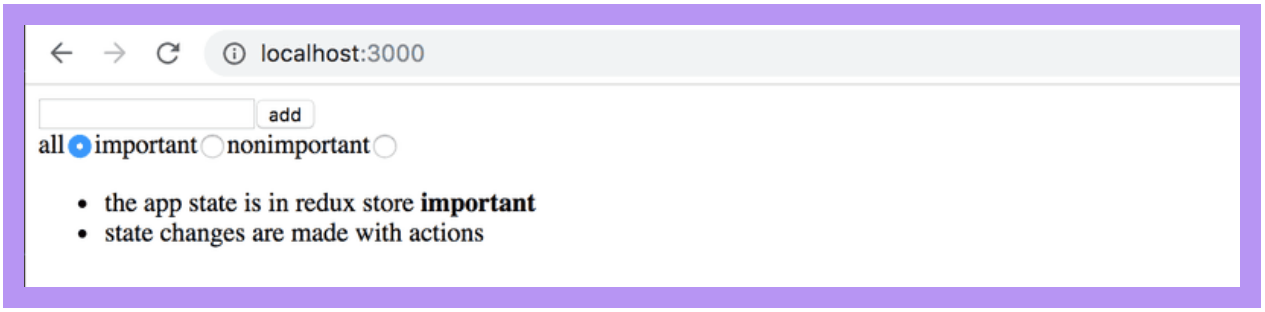
Let's continue our work with the simplified redux version of our notes application.

In order 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 React from 'react'
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.filter
    default:
      return state
  }
}
```

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

```
{
  type: 'SET_FILTER',
  filter: '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',
    filter,
  }
}
```

```
export default filterReducer
```

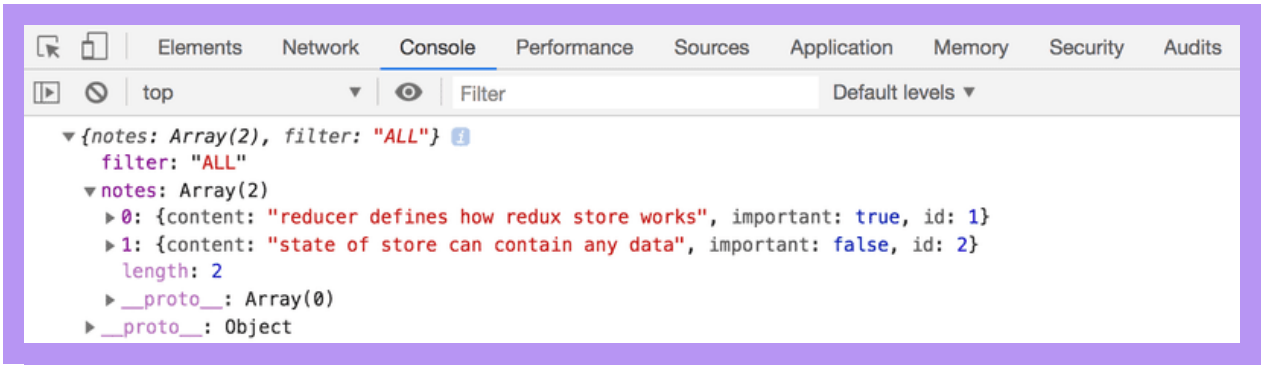
We can create the actual reducer for our application by combining the two existing reducers with the combineReducers function.

Let's define the combined reducer in the *index.js* file:

```
import React from 'react'
import ReactDOM from 'react-dom'
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.render(
  /*
  <Provider store={store}>
    <App />
  </Provider>,
  */
  <div />,
  document.getElementById('root')
)
```

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:

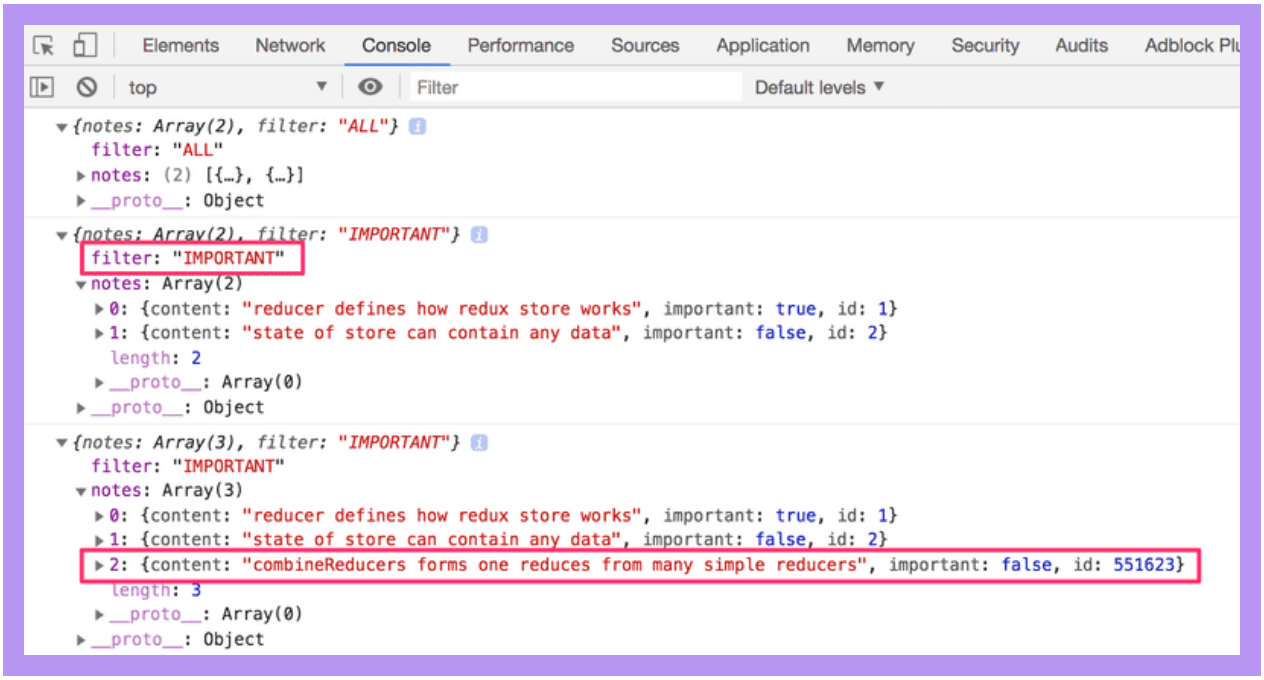
```
const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer,
})
```

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 *index.js* 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'))
```

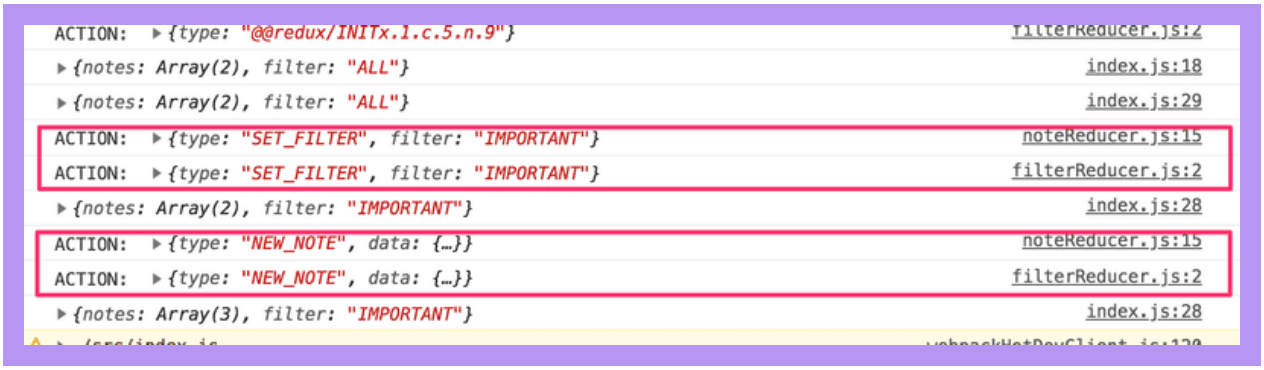
By simulating the creation of a note and changing the state of the filter in this fashion, the state of the store gets logged to the console after every change that is made to the store:



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:



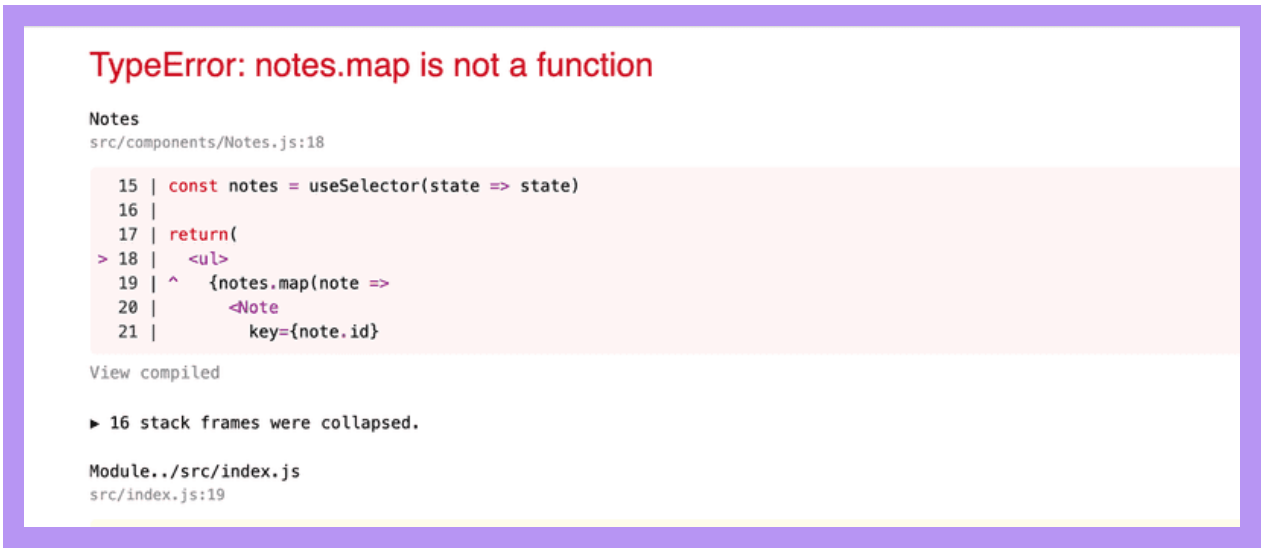
Is there a bug in our code? No. The combined reducer works in such a way that every *action* gets handled in *every* part of the combined reducer. Typically only one reducer is interested in any given action, but there are situations where multiple reducers change their respective parts of the state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

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

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.js* component:

```
import React from 'react'
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 React from 'react'
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 = () => {
  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)
```

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

```
const notes = useSelector(({ filter, notes }) => {
  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.

Redux DevTools

There is an extension Redux DevTools that can be installed on Chrome, in which the state of the Redux-store and the action that changes it can be monitored from the console of the browser.

When debugging, in addition to the browser extension we also have the software library redux-devtools-extension. Let's install it using the command:

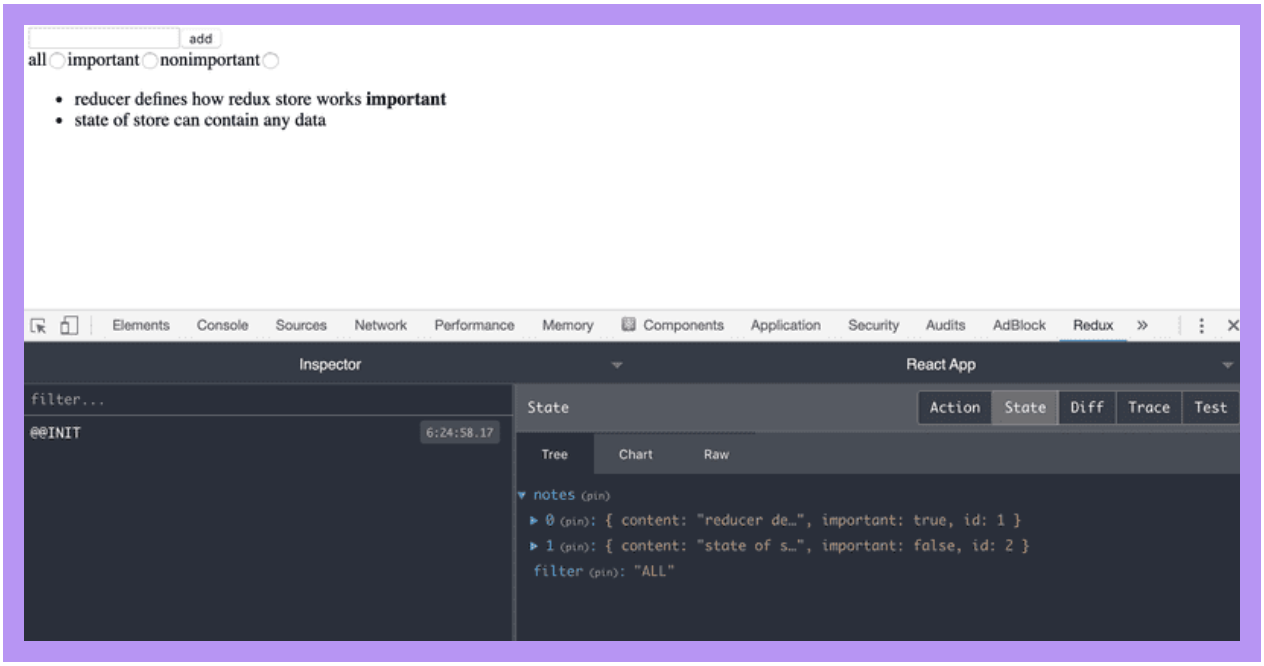
```
npm install --save-dev redux-devtools-extension
```

We'll have to slightly change the definition of the store to get the library up and running:

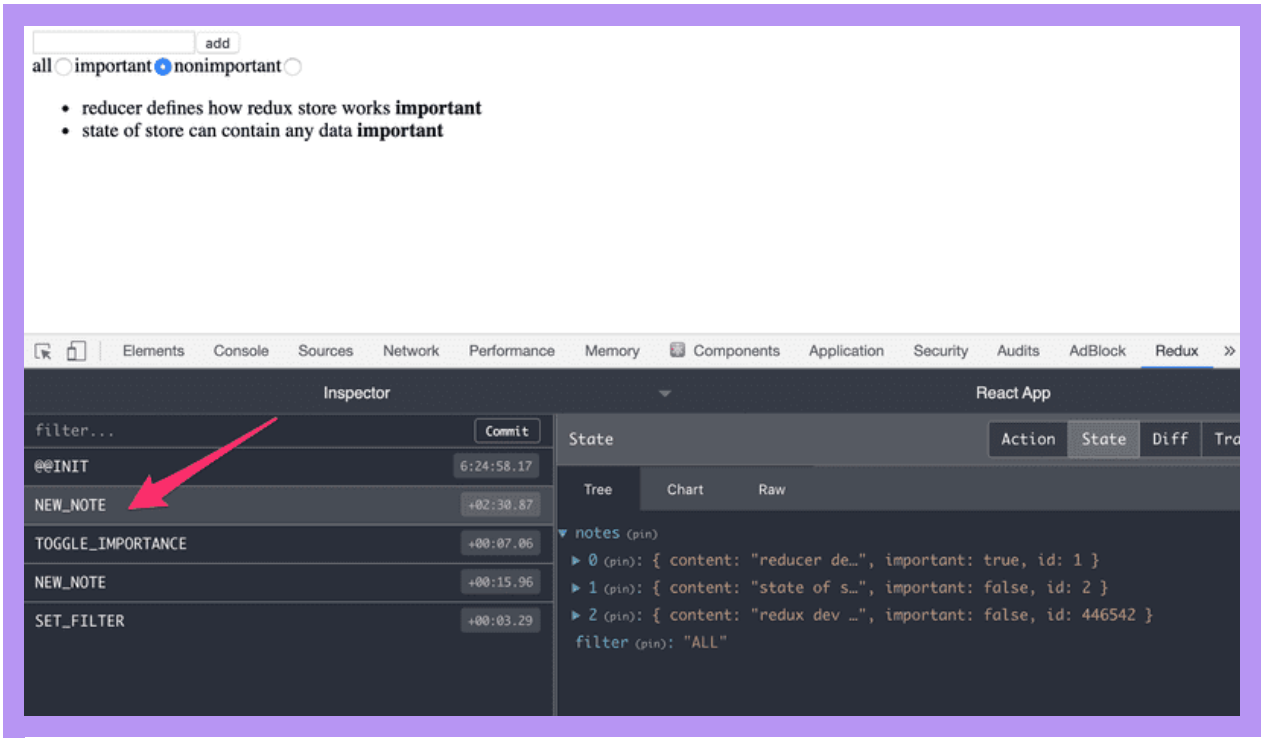
```
// ...
import { createStore, combineReducers } from 'redux'
import { composeWithDevTools } from 'redux-devtools-extension'

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

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

const store = createStore(
```

```
    reducer,
    composeWithDevTools()
  )

  export default store
```

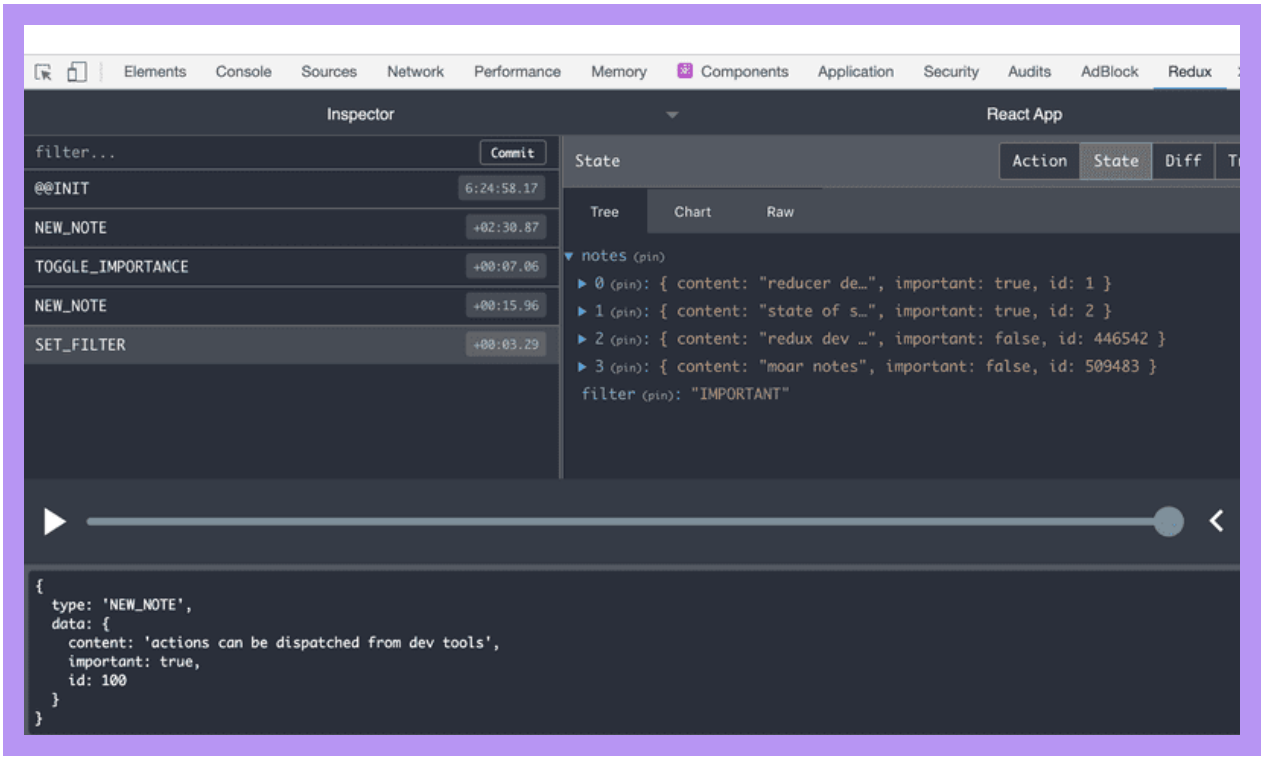
Now when you open the console, the *redux* tab looks like this:



The effect of each action to the store can be easily observed



It's also possible to dispatch actions to the store using the console



You can find the code for our current application in its entirety in the *part6-2* branch of this Github repository.

Exercises 6.9.-6.12.

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

6.9 Better anecdotes, step7

Start using Redux DevTools. Move defining the Redux-store into its own file *store.js*.

6.10 Better anecdotes, step8

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

```
import React from 'react'

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 to take the form:

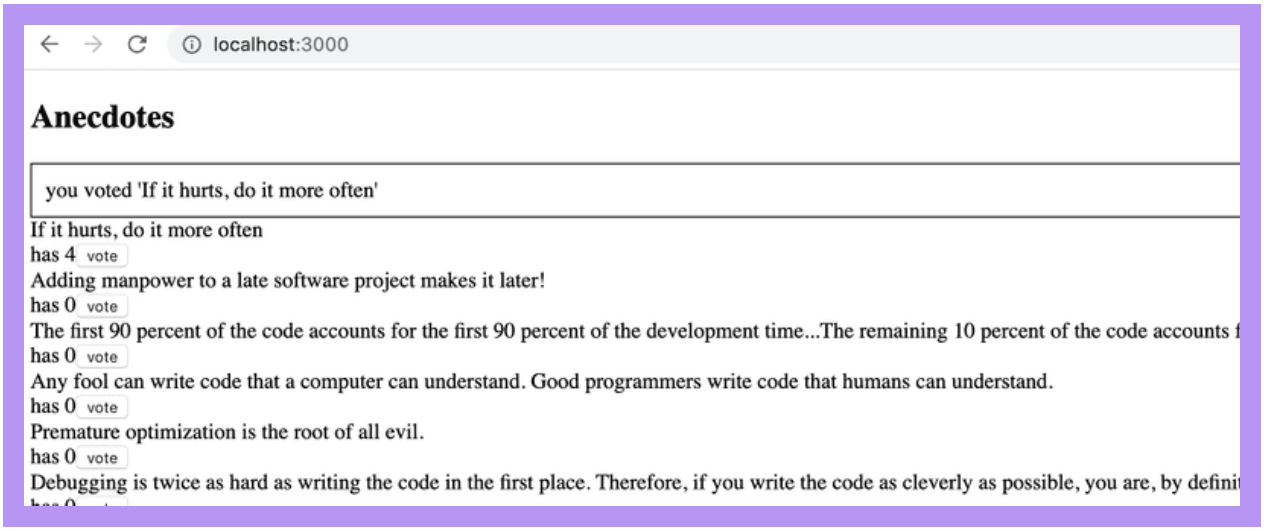
```
import React from 'react'
import { useSelector } from 'react-redux'

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 and refactor the application so that it uses a combined reducer as shown in this part of the course material.

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

6.11 Better anecdotes, step9

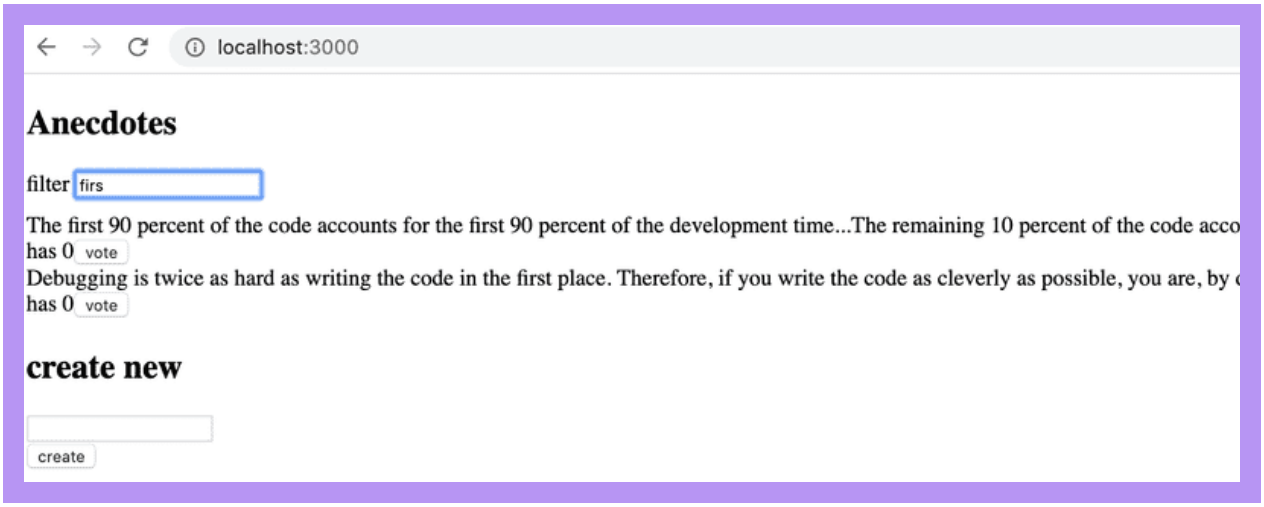
Extend the application so that it uses the *Notification* component to display a message for the duration of five seconds when the user votes for an anecdote or creates a new anecdote:



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

6.12* Better anecdotes, step10

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 and action creators for this purpose.

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

```
import React from 'react'

const Filter = () => {
  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
```

Propose changes to material

Part 6a
Previous part

Part 6c
Next part