{() => fs}

Fullstack > Part 6 > Flux-architecture and Redux

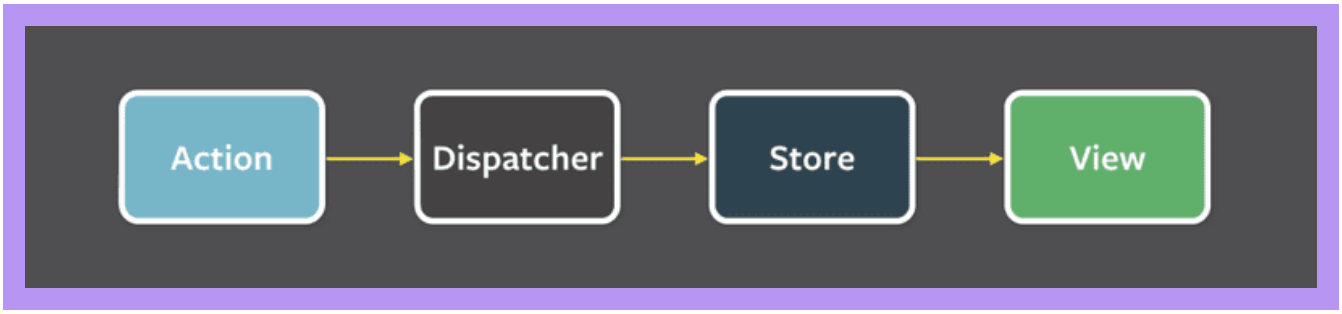
a Flux-architecture and Redux

So far, we have followed the state management conventions recommended by React. We have placed the state and the functions for handling it in the higher level of the component structure of the application. Quite often most of the app state and state altering functions reside directly in the root component. The state and its handler methods have then been passed to other components with props. This works up to a certain point, but when applications grow larger, state management becomes challenging.

Flux-architecture

Already years ago Facebook developed the Flux-architecture to make state management of React apps easier. In Flux, the state is separated from the React components and into its own *stores*. State in the store is not changed directly, but with different *actions*.

When an action changes the state of the store, the views are rerendered:



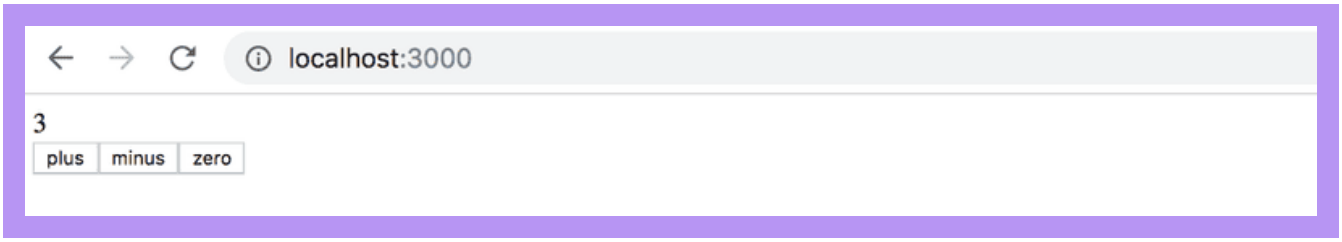
If some action on the application, for example pushing a button, causes the need to change the state, the change is made with an action. This causes re-rendering the view again:

Flux offers a standard way for how and where the application's state is kept and how it is modified.

Redux

Facebook has an implementation for Flux, but we will be using the Redux library. It works with the same principle but is a bit simpler. Facebook also uses Redux now instead of their original Flux.

We will get to know Redux by implementing a counter application yet again:



Create a new Vite application and install redux with the command

```
npm install redux
```

copy

As in Flux, in Redux the state is also stored in a store.

The whole state of the application is stored in *one* JavaScript object in the store. Because our application only needs the value of the counter, we will save it straight to the store. If the state was more complicated, different things in the state would be saved as separate fields of the object.

The state of the store is changed with actions. Actions are objects, which have at least a field determining the *type* of the action. Our application needs for example the following action:

```
{
  type: 'INCREMENT'
}
```

copy

If there is data involved with the action, other fields can be declared as needed. However, our counting app is so simple that the actions are fine with just the type field.

The impact of the action to the state of the application is defined using a reducer. In practice, a reducer is a function that is given the current state and an action as parameters. It *returns* a new state.

Let's now define a reducer for our application:

```
const counterReducer = (state, action) => {
  if (action.type === 'INCREMENT') {
    return state + 1
  } else if (action.type === 'DECREMENT') {
    return state - 1
  } else if (action.type === 'ZERO') {
    return 0
  }

  return state
}
```

The first parameter is the *state* in the store. The reducer returns a *new state* based on the `action` type. So, e.g. when the type of Action is *INCREMENT*, the state gets the old value plus one. If the type of Action is *ZERO* the new value of state is zero.

Let's change the code a bit. We have used if-else statements to respond to an action and change the state. However, the switch statement is the most common approach to writing a reducer.

Let's also define a default value of 0 for the parameter *state*. Now the reducer works even if the store state has not been primed yet.

```
const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'ZERO':
      return 0
    default: // if none of the above matches, code comes here
      return state
  }
}
```

The reducer is never supposed to be called directly from the application's code. It is only given as a parameter to the `createStore` function which creates the store:

```
import { createStore } from 'redux'

const counterReducer = (state = 0, action) => {
  // ...
}

const store = createStore(counterReducer)
```

The store now uses the reducer to handle *actions*, which are *dispatched* or 'sent' to the store with its `dispatch` method.

```
store.dispatch({ type: 'INCREMENT' })
```

copy

You can find out the state of the store using the method `getState`.

For example the following code:

```
const store = createStore(counterReducer)
console.log(store.getState())
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
console.log(store.getState())
store.dispatch({ type: 'ZERO' })
store.dispatch({ type: 'DECREMENT' })
console.log(store.getState())
```

copy

would print the following to the console

```
0
3
-1
```

copy

because at first, the state of the store is 0. After three *INCREMENT* actions the state is 3. In the end, after the *ZERO* and *DECREMENT* actions, the state is -1.

The third important method that the store has is `subscribe`, which is used to create callback functions that the store calls whenever an action is dispatched to the store.

If, for example, we would add the following function to subscribe, *every change in the store* would be printed to the console.

```
store.subscribe(() => {
  const storeNow = store.getState()
  console.log(storeNow)
})
```

copy

so the code

```
const store = createStore(counterReducer)

store.subscribe(() => {
  const storeNow = store.getState()
  console.log(storeNow)
})

store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'INCREMENT' })
store.dispatch({ type: 'ZERO' })
store.dispatch({ type: 'DECREMENT' })
```

would cause the following to be printed

```
1
2
3
0
-1
```

The code of our counter application is the following. All of the code has been written in the same file (`main.jsx`), so *store* is directly available for the React code. We will get to know better ways to structure React/Redux code later.

```
import React from 'react'
import ReactDOM from 'react-dom/client'

import { createStore } from 'redux'

const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return state + 1
    case 'DECREMENT':
      return state - 1
    case 'ZERO':
      return 0
    default:
      return state
  }
}

const store = createStore(counterReducer)

const App = () => {
  return (
    <div>
```

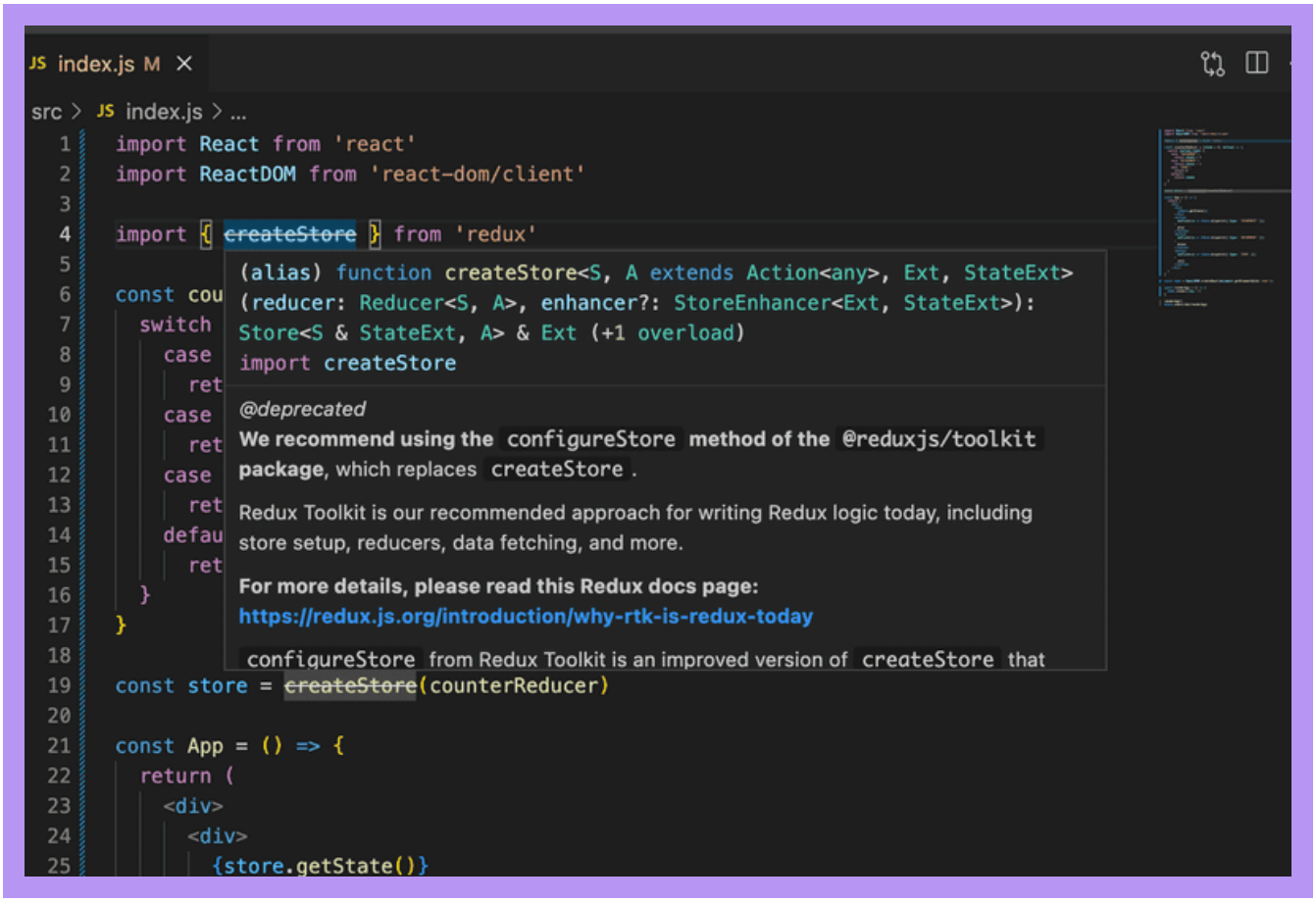
```
        <div>
          {store.getState()}
        </div>
        <button
          onClick={e => store.dispatch({ type: 'INCREMENT' })}
        >
          plus
        </button>
        <button
          onClick={e => store.dispatch({ type: 'DECREMENT' })}
        >
          minus
        </button>
        <button
          onClick={e => store.dispatch({ type: 'ZERO' })}
        >
          zero
        </button>
      </div>
    )
}

const root = ReactDOM.createRoot(document.getElementById('root'))

const renderApp = () => {
  root.render(<App />)
}

renderApp()
store.subscribe(renderApp)
```

There are a few notable things in the code. *App* renders the value of the counter by asking it from the store with the method `store.getState()`. The action handlers of the buttons *dispatch* the right actions to the store.

When the state in the store is changed, React is not able to automatically re-render the application. Thus we have registered a function `renderApp`, which renders the whole app, to listen for changes in the store with the `store.subscribe` method. Note that we have to immediately call the `renderApp` method. Without the call, the first rendering of the app would never happen.

A note about the use of createStore

The most observant will notice that the name of the function createStore is overlined. If you move the mouse over the name, an explanation will appear



The full explanation is as follows

> *We recommend using the configureStore method of the @reduxjs/toolkit package, which replaces createStore.*
>
> *Redux Toolkit is our recommended approach for writing Redux logic today, including store setup, reducers, data fetching, and more.*
>
> *For more details, please read this Redux docs page: https://redux.js.org/introduction/why-rtk-is-redux-today*
>
> *configureStore from Redux Toolkit is an improved version of createStore that simplifies setup and helps avoid common bugs.*
>
> *You should not be using the redux core package by itself today, except for learning purposes. The createStore method from the core redux package will not be removed, but we encourage all users to migrate to using Redux Toolkit for all Redux code.*

So, instead of the function *createStore*, it is recommended to use the slightly more "advanced" function *configureStore*, and we will also use it when we have achieved the basic functionality of Redux.

Side note: *createStore* is defined as "deprecated", which usually means that the feature will be removed in some newer version of the library. The explanation above and this discussion reveal that *createStore* will not be removed, and it has been given the status *deprecated*, perhaps with slightly incorrect

reasons. So the function is not obsolete, but today there is a more preferable, new way to do almost the same thing.

Redux-notes

We aim to modify our note application to use Redux for state management. However, let's first cover a few key concepts through a simplified note application.

The first version of our application is the following

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    state.push(action.payload)
    return state
  }

  return state
}

const store = createStore(noteReducer)

store.dispatch({
  type: 'NEW_NOTE',
  payload: {
    content: 'the app state is in redux store',
    important: true,
    id: 1
  }
})

store.dispatch({
  type: 'NEW_NOTE',
  payload: {
    content: 'state changes are made with actions',
    important: false,
    id: 2
  }
})

const App = () => {
  return(
    <div>
      <ul>
        {store.getState().map(note=>
          <li key={note.id}>
            {note.content} <strong>{note.important ? 'important' : ''}</strong>
          </li>
        )}
        </ul>
    </div>
```

```
  )
}
```

So far the application does not have the functionality for adding new notes, although it is possible to do so by dispatching *NEW_NOTE* actions.

Now the actions have a type and a field *payload*, which contains the note to be added:

```
{
  type: 'NEW_NOTE',
  payload: {
    content: 'state changes are made with actions',
    important: false,
    id: 2
  }
}
```

The choice of the field name is not random. The general convention is that actions have exactly two fields, *type* telling the type and *payload* containing the data included with the Action.

Pure functions, immutable

The initial version of the reducer is very simple:

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    state.push(action.payload)
    return state
  }

  return state
}
```

The state is now an Array. *NEW_NOTE*-type actions cause a new note to be added to the state with the `push` method.

The application seems to be working, but the reducer we have declared is bad. It breaks the basic assumption that reducers must be pure functions.

Pure functions are such, that they *do not cause any side effects* and they must always return the same response when called with the same parameters.

We added a new note to the state with the method `state.push(action.payload)` which *changes* the state of the state-object. This is not allowed. The problem is easily solved by using the concat method, which creates a *new array*, which contains all the elements of the old array and the new element:

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    return state.concat(action.payload)
  }

  return state
}
```

A reducer state must be composed of immutable objects. If there is a change in the state, the old object is not changed, but it is *replaced with a new, changed, object*. This is exactly what we did with the new reducer: the old array is replaced with the new one.

Let's expand our reducer so that it can handle the change of a note's importance:

```
{
  type: 'TOGGLE_IMPORTANCE',
  payload: {
    id: 2
  }
}
```

Since we do not have any code which uses this functionality yet, we are expanding the reducer in the 'test-driven' way. Let's start by creating a test for handling the action *NEW_NOTE*.

We have to first configure the Jest testing library for the project. Let us install the following dependencies:

```
npm install --save-dev jest @babel/preset-env @babel/preset-react eslint-plugin-jest
```

Next we'll create the file *.babelrc*, with the following content:

```
{
  "presets": [
    "@babel/preset-env",
    ["@babel/preset-react", { "runtime": "automatic" }]
  ]
}
```

Let us expand *package.json* with a script for running the tests:

```
{
  // ...
```

```
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx --report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview",
    "test": "jest"
  },
  // ...
}
```

And finally, *.eslintrc.cjs* needs to be altered as follows:

```
module.exports = {
  root: true,
  env: {
    browser: true,
    es2020: true,
    "jest/globals": true
  },
  // ...
}
```

To make testing easier, we'll first move the reducer's code to its own module, to the file *src/reducers/noteReducer.js*. We'll also add the library deep-freeze, which can be used to ensure that the reducer has been correctly defined as an immutable function. Let's install the library as a development dependency:

```
npm install --save-dev deep-freeze
```

The test, which we define in file *src/reducers/noteReducer.test.js*, has the following content:

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

describe('noteReducer', () => {
  test('returns new state with action NEW_NOTE', () => {
    const state = []
    const action = {
      type: 'NEW_NOTE',
      payload: {
        content: 'the app state is in redux store',
        important: true,
        id: 1
      }
    }

    deepFreeze(state)
```

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

    expect(newState).toHaveLength(1)
    expect(newState).toContainEqual(action.payload)
  })
})
```

The *deepFreeze(state)* command ensures that the reducer does not change the state of the store given to it as a parameter. If the reducer uses the `push` command to manipulate the state, the test will not pass

```
FAIL src/reducers/noteReducer.test.js
  noteReducer
    ✕ returns new state with action NEW_NOTE (16ms)

  ● noteReducer › returns new state with action NEW_NOTE

    TypeError: Cannot add property 0, object is not extensible
        at Array.push (<anonymous>)

       5 |
       6 |   if (action.type === 'NEW_NOTE') {
    >  7 |     state.push(action.data)
         |           ^
       8 |     return state
       9 |   }
      10 |

      at push (src/reducers/noteReducer.js:7:11)
      at Object.it (src/reducers/noteReducer.test.js:17:22)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 1 total
```

Now we'll create a test for the *TOGGLE_IMPORTANCE* action:

```
test('returns new state with action TOGGLE_IMPORTANCE', () => {
  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: 'TOGGLE_IMPORTANCE',
    payload: {
      id: 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
  })
})
```

So the following action

```
{
  type: 'TOGGLE_IMPORTANCE',
  payload: {
    id: 2
  }
}
```

has to change the importance of the note with the id 2.

The reducer is expanded as follows

```
const noteReducer = (state = [], action) => {
  switch(action.type) {
    case 'NEW_NOTE':
      return state.concat(action.payload)
    case 'TOGGLE_IMPORTANCE': {
      const id = action.payload.id
      const noteToChange = state.find(n => n.id === id)
      const changedNote = {
        ...noteToChange,
        important: !noteToChange.important
      }
      return state.map(note =>
        note.id !== id ? note : changedNote
      )
     }
    default:
```

```
      return state
  }
}
```

We create a copy of the note whose importance has changed with the syntax familiar from part 2, and replace the state with a new state containing all the notes which have not changed and the copy of the changed note *changedNote*.

Let's recap what goes on in the code. First, we search for a specific note object, the importance of which we want to change:

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

copy

then we create a new object, which is a *copy* of the original note, only the value of the *important* field has been changed to the opposite of what it was:

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

copy

A new state is then returned. We create it by taking all of the notes from the old state except for the desired note, which we replace with its slightly altered copy:

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

copy

Array spread syntax

Because we now have quite good tests for the reducer, we can refactor the code safely.

Adding a new note creates the state returned from the Array's `concat` function. Let's take a look at how we can achieve the same by using the JavaScript array spread syntax:

```
const noteReducer = (state = [], action) => {
  switch(action.type) {
    case 'NEW_NOTE':
      return [...state, action.payload]
    case 'TOGGLE_IMPORTANCE':
      // ...
    default:
    return state
```

copy

```
  }
}
```

The spread -syntax works as follows. If we declare

```
const numbers = [1, 2, 3]
```

`...numbers` breaks the array up into individual elements, which can be placed in another array.

```
[...numbers, 4, 5]
```

and the result is an array *[1, 2, 3, 4, 5]*.

If we would have placed the array to another array without the spread

```
[numbers, 4, 5]
```

the result would have been *[[1, 2, 3], 4, 5]*.

When we take elements from an array by destructuring, a similar-looking syntax is used to *gather* the rest of the elements:

```
const numbers = [1, 2, 3, 4, 5, 6]

const [first, second, ...rest] = numbers

console.log(first)     // prints 1
console.log(second)    // prints 2
console.log(rest)      // prints [3, 4, 5, 6]
```

Exercises 6.1.-6.2.

Let's make a simplified version of the unicafe exercise from part 1. Let's handle the state management with Redux.

You can take the code from this repository https://github.com/fullstack-hy2020/unicafe-redux for the base of your project.

Start by removing the git configuration of the cloned repository, and by installing dependencies

```
cd unicafe-redux   // go to the directory of cloned repository
rm -rf .git
npm install
```

6.1: Unicafe Revisited, step 1

Before implementing the functionality of the UI, let's implement the functionality required by the store.

We have to save the number of each kind of feedback to the store, so the form of the state in the store is:

```
{
  good: 5,
  ok: 4,
  bad: 2
}
```

The project has the following base for a reducer:

```
const initialState = {
  good: 0,
  ok: 0,
  bad: 0
}

const counterReducer = (state = initialState, action) => {
  console.log(action)
  switch (action.type) {
    case 'GOOD':
      return state
    case 'OK':
      return state
    case 'BAD':
      return state
    case 'ZERO':
      return state
    default: return state
  }

}

export default counterReducer
```

and a base for its tests

```
import deepFreeze from 'deep-freeze'
import counterReducer from './reducer'

describe('unicafe reducer', () => {
  const initialState = {
    good: 0,
    ok: 0,
    bad: 0
  }

  test('should return a proper initial state when called with undefined state', () => {
    const state = {}
    const action = {
      type: 'DO_NOTHING'
    }

    const newState = counterReducer(undefined, action)
    expect(newState).toEqual(initialState)
  })

  test('good is incremented', () => {
    const action = {
      type: 'GOOD'
    }
    const state = initialState

    deepFreeze(state)
    const newState = counterReducer(state, action)
    expect(newState).toEqual({
      good: 1,
      ok: 0,
      bad: 0
    })
  })
})
```

Implement the reducer and its tests.

In the tests, make sure that the reducer is an *immutable function* with the *deep-freeze* library. Ensure that the provided first test passes, because Redux expects that the reducer returns the original state when it is called with a first parameter - which represents the previous *state* - with the value *undefined*.

Start by expanding the reducer so that both tests pass. Then add the rest of the tests, and finally the functionality that they are testing.

A good model for the reducer is the redux-notes example above.

6.2: Unicafe Revisited, step2

Now implement the actual functionality of the application.

Your application can have a modest appearance, nothing else is needed but buttons and the number of reviews for each type:

Uncontrolled form

Let's add the functionality for adding new notes and changing their importance:

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

const App = () => {
  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    store.dispatch({
      type: 'NEW_NOTE',
      payload: {
        content,
        important: false,
        id: generateId()
      }
    })
  }

  const toggleImportance = (id) => {
    store.dispatch({
      type: 'TOGGLE_IMPORTANCE',
      payload: { id }
    })
  }

  return (
    <div>
      <form onSubmit={addNote}>
        <input name="note" />
        <button type="submit">add</button>
      </form>
      <ul>
        {store.getState().map(note =>
          <li
            key={note.id}
            onClick={() => toggleImportance(note.id)}
          >
            {note.content} <strong>{note.important ? 'important' : ''}</strong>
          </li>
        )}
```

```
      </ul>
    </div>
  )
}
```

The implementation of both functionalities is straightforward. It is noteworthy that we *have not* bound the state of the form fields to the state of the *App* component like we have previously done. React calls this kind of form uncontrolled.

> Uncontrolled forms have certain limitations (for example, dynamic error messages or disabling the submit button based on input are not possible). However they are suitable for our current needs.

You can read more about uncontrolled forms here.

The method for adding new notes is simple, it just dispatches the action for adding notes:

```
addNote = (event) => {
  event.preventDefault()
  const content = event.target.note.value
  event.target.note.value = ''
  store.dispatch({
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
  })
}
```

We can get the content of the new note straight from the form field. Because the field has a name, we can access the content via the event object *event.target.note.value*.

```
<form onSubmit={addNote}>
  <input name="note" />
  <button type="submit">add</button>
</form>
```

A note's importance can be changed by clicking its name. The event handler is very simple:

```
toggleImportance = (id) => {
  store.dispatch({
    type: 'TOGGLE_IMPORTANCE',
    payload: { id }
```

```
  })
}
```

Action creators

We begin to notice that, even in applications as simple as ours, using Redux can simplify the frontend code. However, we can do a lot better.

React components don't need to know the Redux action types and forms. Let's separate creating actions into separate functions:

```
const createNote = (content) => {
  return {
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
  }
}

const toggleImportanceOf = (id) => {
  return {
    type: 'TOGGLE_IMPORTANCE',
    payload: { id }
  }
}
```

Functions that create actions are called action creators.

The *App* component does not have to know anything about the inner representation of the actions anymore, it just gets the right action by calling the creator function:

```
const App = () => {
  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    store.dispatch(createNote(content))

  }

  const toggleImportance = (id) => {
    store.dispatch(toggleImportanceOf(id))
  }
```

```
  // ...
}
```

Forwarding Redux Store to various components

Aside from the reducer, our application is in one file. This is of course not sensible, and we should separate *App* into its module.

Now the question is, how can the *App* access the store after the move? And more broadly, when a component is composed of many smaller components, there must be a way for all of the components to access the store. There are multiple ways to share the Redux store with the components. First, we will look into the newest, and possibly the easiest way, which is using the hooks API of the react-redux library.

First, we install react-redux

```
npm install react-redux
```

copy

Next, we move the `App` component into its own file `App.jsx`. Let's see how this affects the rest of the application files.

`main.jsx` becomes:

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

import App from './App'
import noteReducer from './reducers/noteReducer'

const store = createStore(noteReducer)

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

copy

Note, that the application is now defined as a child of a Provider component provided by the react-redux library. The application's store is given to the Provider as its attribute *store*.

Defining the action creators has been moved to the file *reducers/noteReducer.js* where the reducer is defined. That file looks like this:

```
const noteReducer = (state = [], action) => {
  // ...
}

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

export const createNote = (content) => {
  return {
    type: 'NEW_NOTE',
    payload: {
      content,
      important: false,
      id: generateId()
    }
  }
}

export const toggleImportanceOf = (id) => {
  return {
    type: 'TOGGLE_IMPORTANCE',
    payload: { id }
  }
}

export default noteReducer
```

If the application has many components which need the store, the *App* component must pass *store* as props to all of those components.

The module now has multiple export commands.

The reducer function is still returned with the *export default* command, so the reducer can be imported the usual way:

```
import noteReducer from './reducers/noteReducer'
```

A module can have only *one default export*, but multiple "normal" exports

```
export const createNote = (content) => {
  // ...
}

export const toggleImportanceOf = (id) => {
  // ...
}
```

Normally (not as defaults) exported functions can be imported with the curly brace syntax:

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

copy

Code for the *App* component

```
import { createNote, toggleImportanceOf } from './reducers/noteReducer'
import { useSelector, useDispatch } from 'react-redux'

const App = () => {
  const dispatch = useDispatch()
  const notes = useSelector(state => state)

  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  const toggleImportance = (id) => {
    dispatch(toggleImportanceOf(id))
  }

  return (
    <div>
      <form onSubmit={addNote}>
        <input name="note" />
        <button type="submit">add</button>
      </form>
      <ul>
        {notes.map(note =>
          <li
            key={note.id}
            onClick={() => toggleImportance(note.id)}
          >
            {note.content} <strong>{note.important ? 'important' : ''}</strong>
          </li>
        )}
      </ul>
    </div>
  )
}

export default App
```

copy

There are a few things to note in the code. Previously the code dispatched actions by calling the dispatch method of the Redux store:

```
store.dispatch({
  type: 'TOGGLE_IMPORTANCE',
  payload: { id }
})
```

Now it does it with the *dispatch* function from the `useDispatch` hook.

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

const App = () => {
  const dispatch = useDispatch()
  // ...

  const toggleImportance = (id) => {
    dispatch(toggleImportanceOf(id))
  }

  // ...
}
```

The *useDispatch* hook provides any React component access to the dispatch function of the Redux store defined in *main.jsx*. This allows all components to make changes to the state of the Redux store.

The component can access the notes stored in the store with the `useSelector` -hook of the react-redux library.

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

const App = () => {
  // ...
  const notes = useSelector(state => state)
  // ...
}
```

useSelector receives a function as a parameter. The function either searches for or selects data from the Redux store. Here we need all of the notes, so our selector function returns the whole state:

```
state => state
```

which is a shorthand for:

```
(state) => {
  return state
}
```

Usually, selector functions are a bit more interesting and return only selected parts of the contents of the Redux store. We could for example return only notes marked as important:

```
const importantNotes = useSelector(state => state.filter(note => note.important))
```

The current version of the application can be found on GitHub, branch *part6-0*.

More components

Let's separate creating a new note into a component.

```
import { useDispatch } from 'react-redux'
import { createNote } from '../reducers/noteReducer'

const NewNote = () => {
  const dispatch = useDispatch()

  const addNote = (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  return (
    <form onSubmit={addNote}>
      <input name="note" />
      <button type="submit">add</button>
    </form>
  )
}

export default NewNote
```

Unlike in the React code we did without Redux, the event handler for changing the state of the app (which now lives in Redux) has been moved away from the *App* to a child component. The logic for changing the state in Redux is still neatly separated from the whole React part of the application.

We'll also separate the list of notes and displaying a single note into their own components (which will both be placed in the *Notes.jsx* file):

```
import { useDispatch, useSelector } from 'react-redux'
import { toggleImportanceOf } from '../reducers/noteReducer'

const Note = ({ note, handleClick }) => {
  return(
    <li onClick={handleClick}>
      {note.content}
      <strong> {note.important ? 'important' : ''}</strong>
    </li>
  )
}

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

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

export default Notes
```

The logic for changing the importance of a note is now in the component managing the list of notes.

There is not much code left in *App*:

```
const App = () => {

  return (
    <div>
      <NewNote />
      <Notes />
    </div>
  )
}
```

Note, responsible for rendering a single note, is very simple and is not aware that the event handler it gets as props dispatches an action. These kinds of components are called presentational in React

terminology.

Notes, on the other hand, is a container component, as it contains some application logic: it defines what the event handlers of the *Note* components do and coordinates the configuration of *presentational* components, that is, the *Note*s.

The code of the Redux application can be found on GitHub, on the branch *part6-1*.

Exercises 6.3.-6.8.

Let's make a new version of the anecdote voting application from part 1. Take the project from this repository https://github.com/fullstack-hy2020/redux-anecdotes as the base of your solution.

If you clone the project into an existing git repository, *remove the git configuration of the cloned application:*

```
cd redux-anecdotes  // go to the cloned repository
rm -rf .git
```

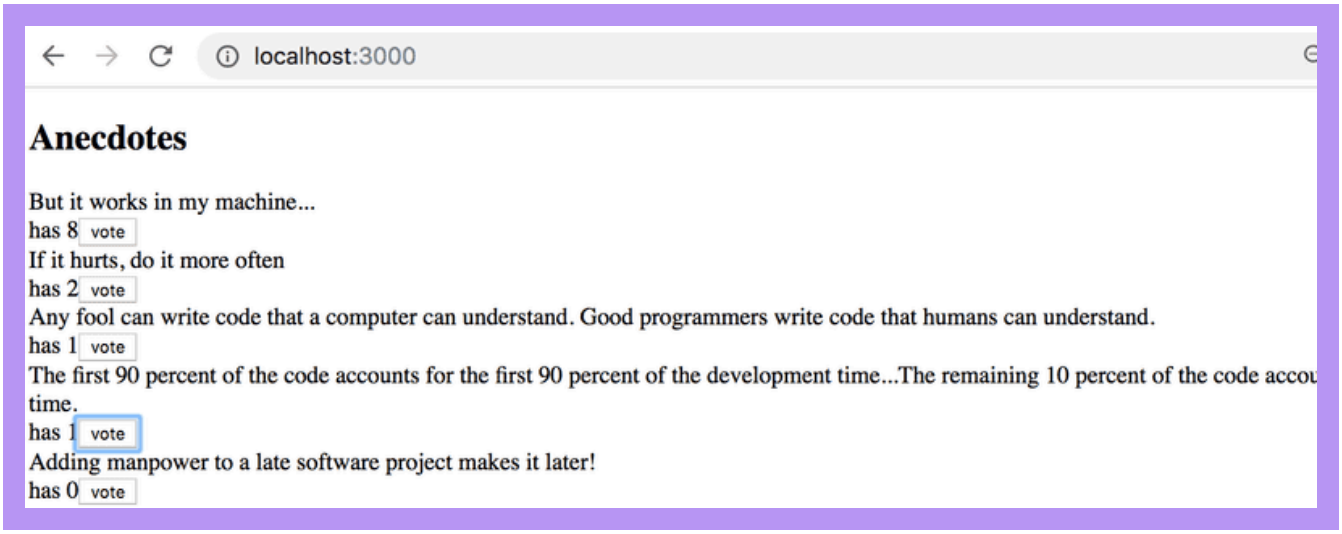
copy

The application can be started as usual, but you have to install the dependencies first:

```
npm install
npm run dev
```

copy

After completing these exercises, your application should look like this:



6.3: Anecdotes, step 1

Implement the functionality for voting anecdotes. The number of votes must be saved to a Redux store.

6.4: Anecdotes, step 2

Implement the functionality for adding new anecdotes.

You can keep the form uncontrolled like we did earlier.

6.5: Anecdotes, step 3

Make sure that the anecdotes are ordered by the number of votes.

6.6: Anecdotes, step 4

If you haven't done so already, separate the creation of action-objects to action creator-functions and place them in the *src/reducers/anecdoteReducer.js* file, so do what we have been doing since the chapter action creators.

6.7: Anecdotes, step 5

Separate the creation of new anecdotes into a component called *AnecdoteForm*. Move all logic for creating a new anecdote into this new component.

6.8: Anecdotes, step 6

Separate the rendering of the anecdote list into a component called *AnecdoteList*. Move all logic related to voting for an anecdote to this new component.

Now the *App* component should look like this:

```
import AnecdoteForm from './components/AnecdoteForm'
import AnecdoteList from './components/AnecdoteList'

const App = () => {
  return (
    <div>
      <h2>Anecdotes</h2>
      <AnecdoteList />
      <AnecdoteForm />
    </div>
  )
}

export default App
```

copy

Propose changes to material

Part 5
Previous part

Part 6b
Next part

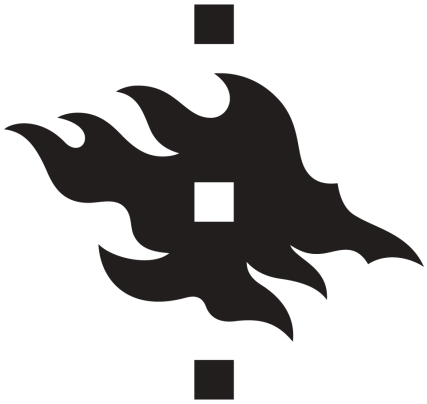
About course

Course contents

FAQ

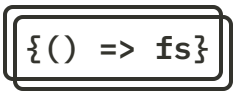
Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON



b Many reducers

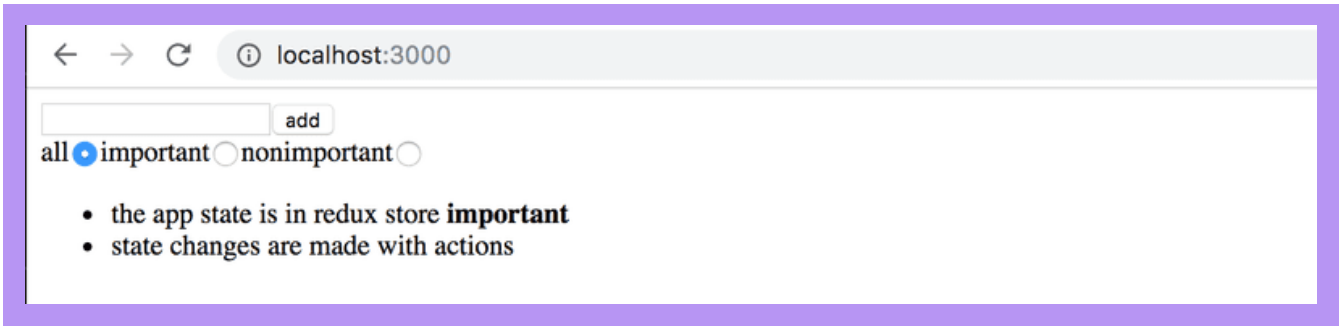
Let's continue our work with the simplified Redux version of our notes application.

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.

In the following section, we will implement filtering by storing both the notes as well as *the value of the filter* in the redux store. When we are finished, we would like the state of the store to look like this:

```
{
  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 was stored in the state of the previous 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 its code 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
```

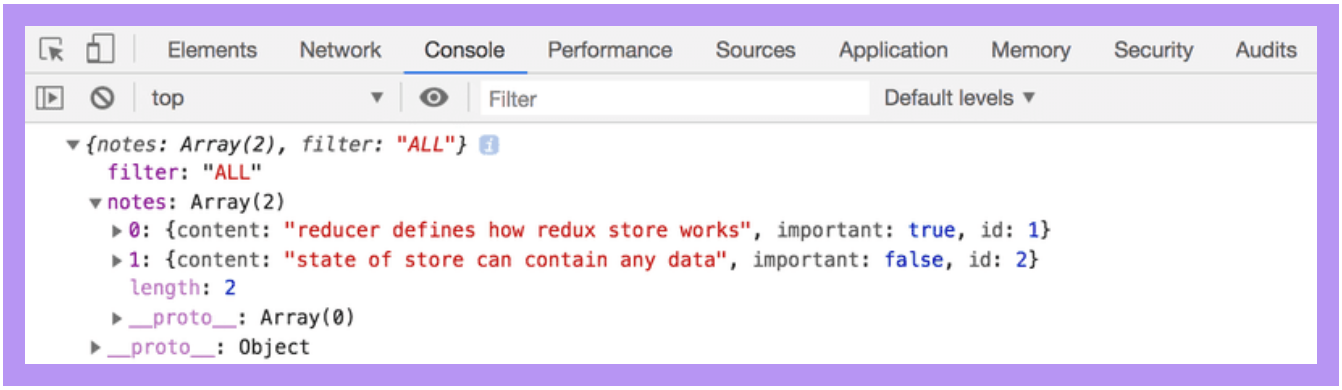
We can create the actual reducer for our application by combining the two existing reducers with the `combineReducers` function.

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

```
import 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:

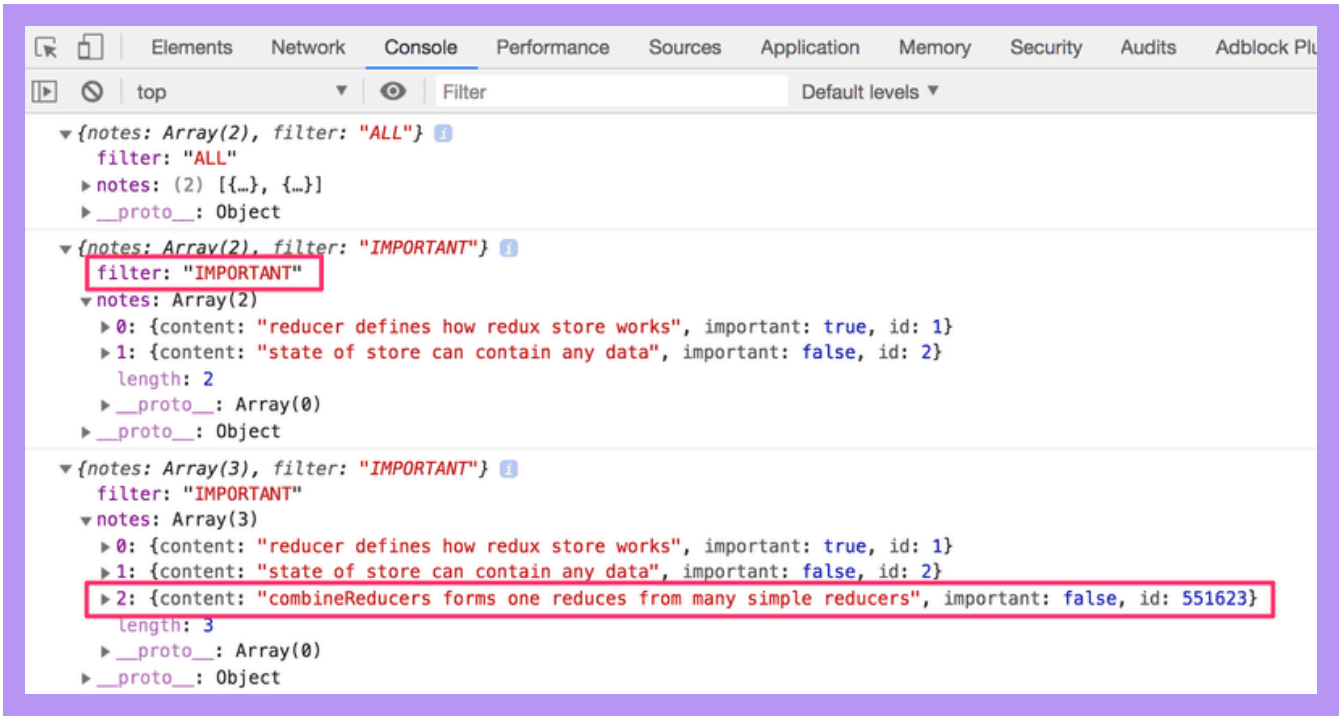
```
const reducer = combineReducers({
  notes: noteReducer,
  filter: filterReducer,
})
```

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'))
```

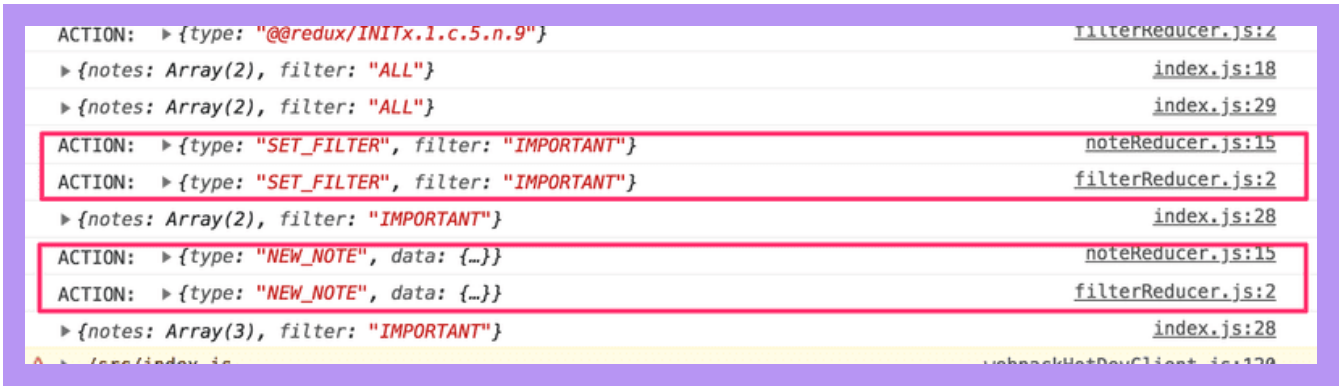
By simulating the creation of a note and changing the state of the filter in this fashion, the state of the store gets logged to the console after every change that is made to the store:



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, or in other words, every reducer "listens" to all of the dispatched actions and does something with them if it has been instructed to do so. 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:

```
Uncaught TypeError: notes.map is not a function
    at Notes (Notes.jsx:19:14)
    at renderWithHooks (react-dom.development.js:16305:18)
    at mountIndeterminateComponent (react-dom.development.js:20074:13)
    at beginWork (react-dom.development.js:21587:16)
    at HTMLUnknownElement.callCallback2 (react-dom.development.js:4164:14)
    at Object.invokeGuardedCallbackDev (react-dom.development.js:4213:16)
    at invokeGuardedCallback (react-dom.development.js:4277:31)
    at beginWork$1 (react-dom.development.js:27451:7)
    at performUnitOfWork (react-dom.development.js:26557:12)
    at workLoopSync (react-dom.development.js:26466:5)
```

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)
```

copy

And now it returns only its field *notes*

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

copy

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
```

copy

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'
```

copy

```
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)
})
```

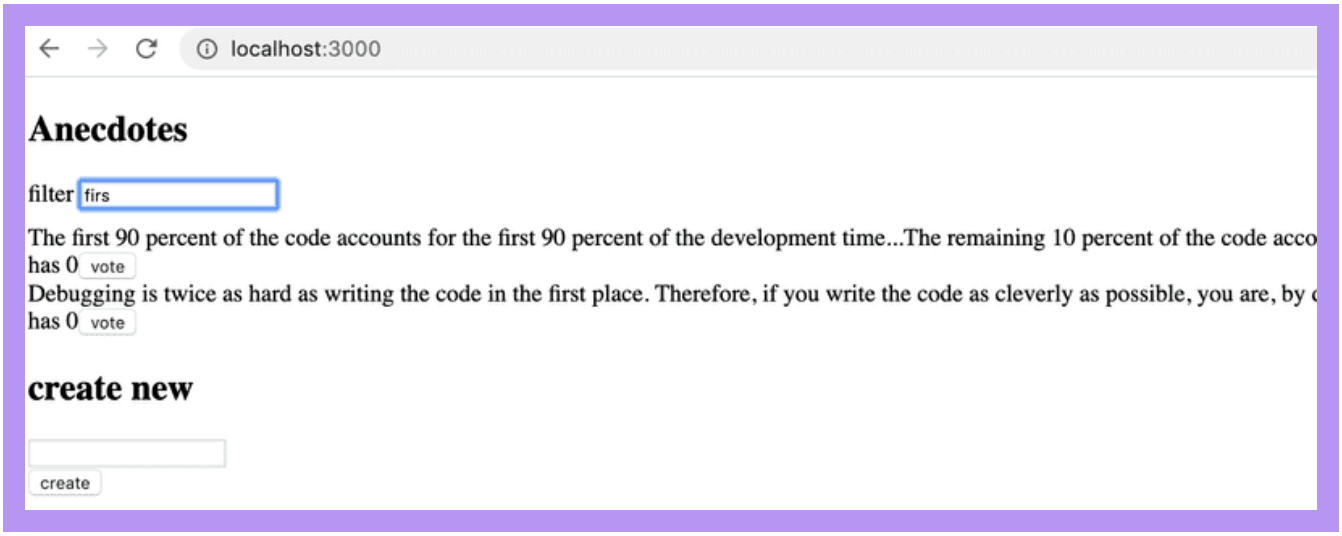
There is a slight cosmetic flaw in our application. Even though the filter is set to *ALL* by default, the associated radio button is not selected. Naturally, this issue can be fixed, but since this is an unpleasant but ultimately harmless bug we will save the fix for later.

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

Exercise 6.9

6.9 Better Anecdotes, step 7

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 = () => {
  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
```

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 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 we don't need the `combineReducers` function to create the store's reducer. 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!'))
```

copy

This dispatch call is equivalent to dispatching the following object:

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

copy

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(),
  })
}
```

copy

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 directly *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
```

copy

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

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

copy

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,
```

copy

```
        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

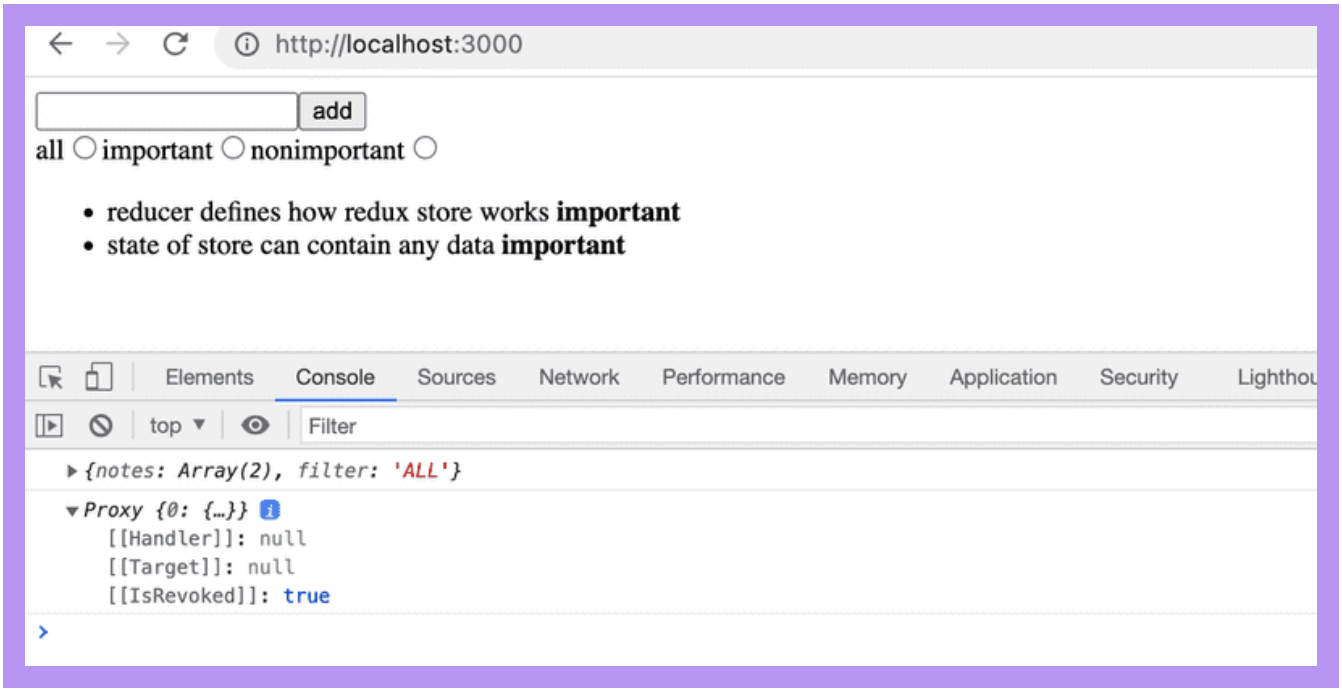
As we have learned, console.log is an extremely powerful tool; it often saves us from trouble.

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 internally to save the state of the Store.

The status can be converted to a human-readable format by using the `current` function from the immer library.

Let's update the imports to include the "current" function from the immer library:

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

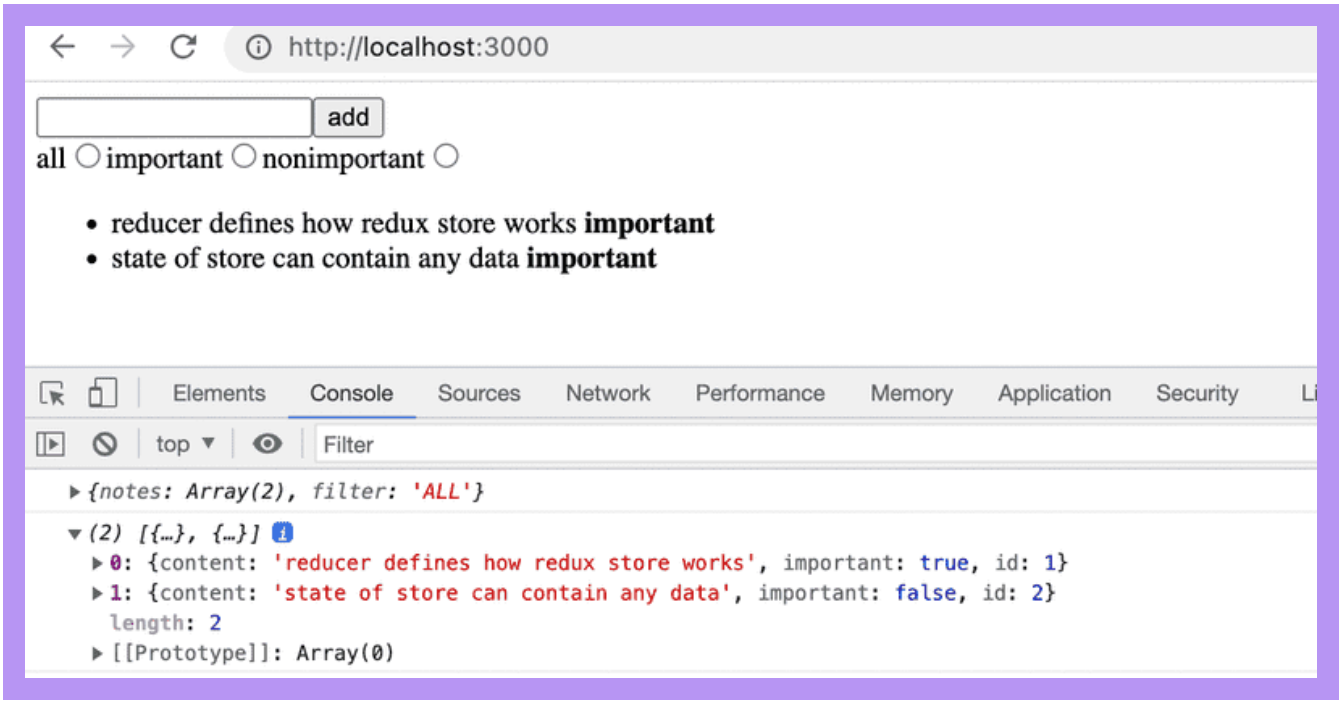
copy

Then we update the console.log function call:

```
console.log(current(state))
```

copy

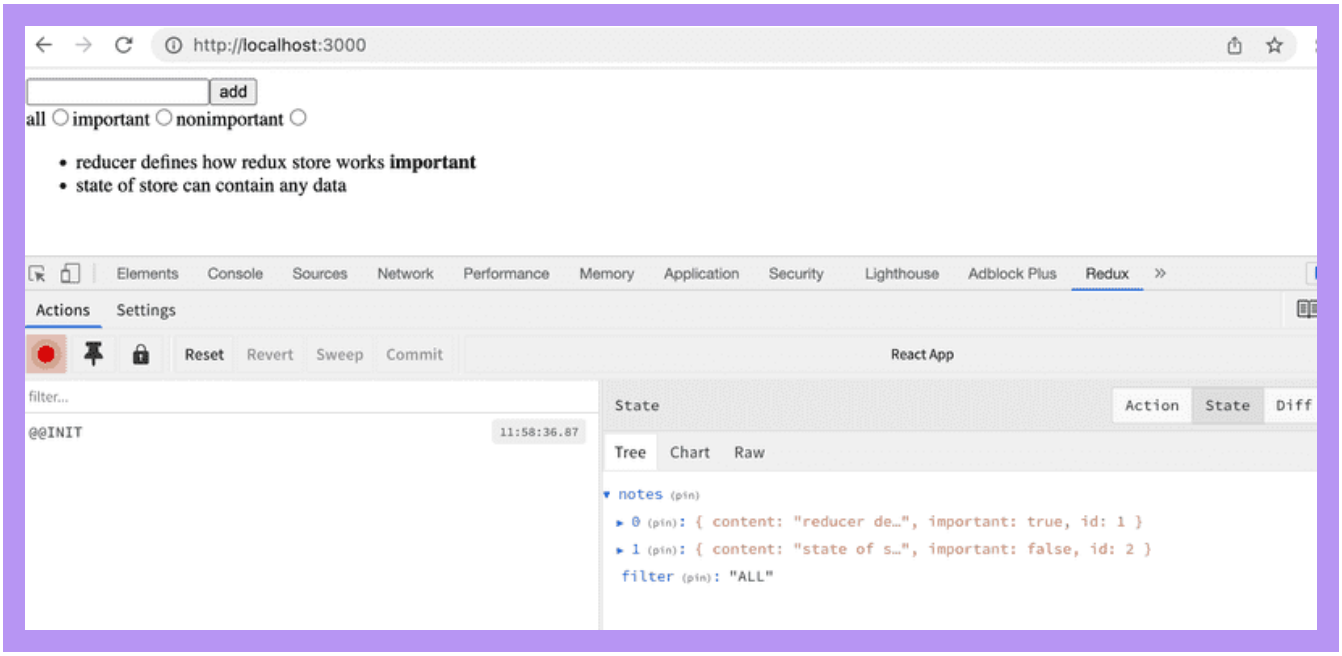
Console output is now human readable



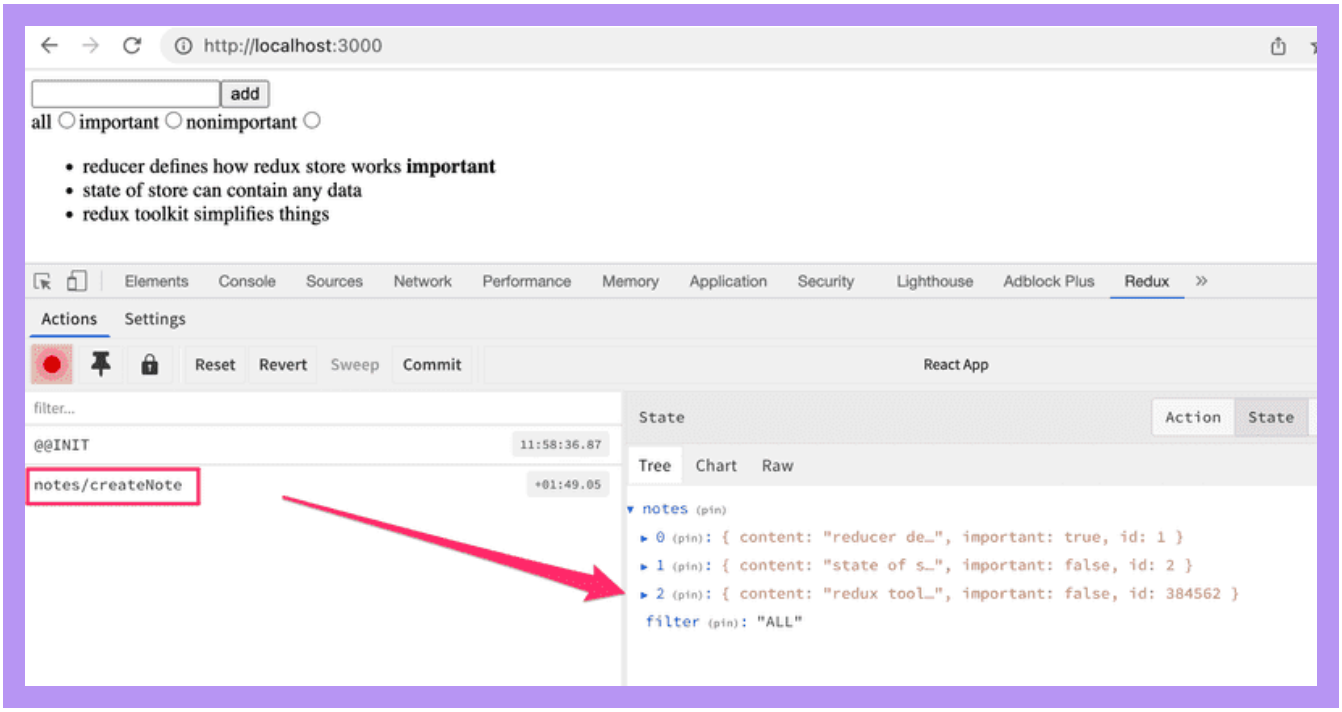
Redux DevTools

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.

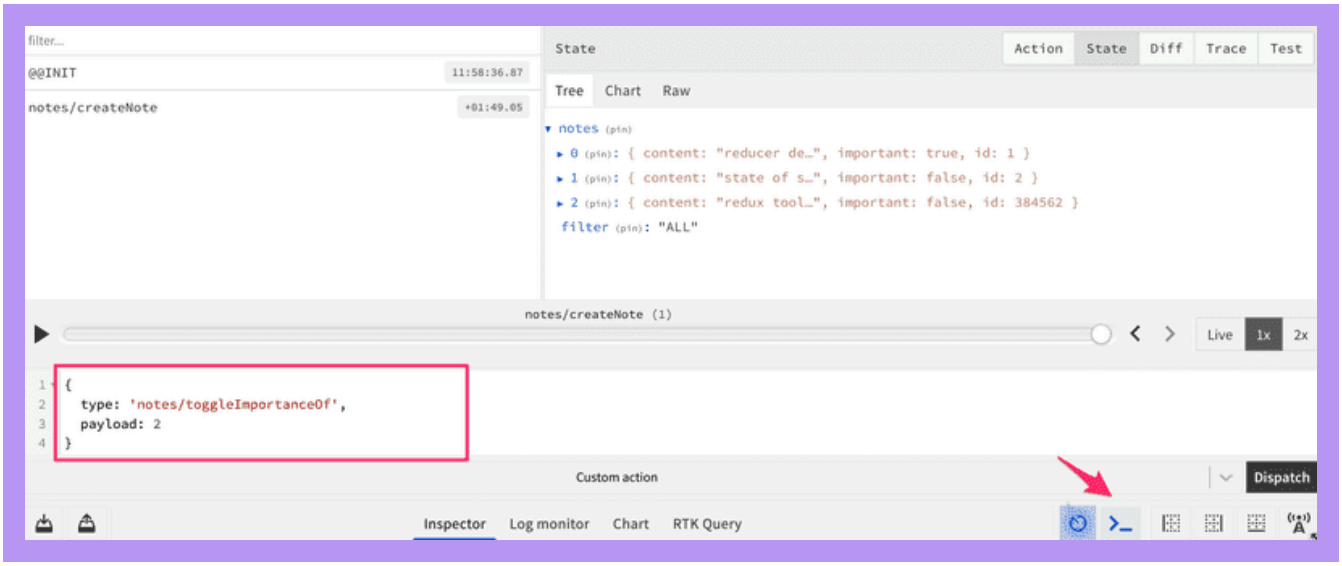
Once the addon is installed, clicking the *Redux* tab in the browser's developer tools, the Redux DevTools should open:



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, step 8

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, step 9

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

Implementation note: when you use the Redux Toolkit to return the initial state of anecdotes, it will be immutable, so you will need to make a copy of it to sort the anecdotes, or you will encounter the error "TypeError: Cannot assign to read only property". You can use the spread syntax to make a copy of the array. Instead of:

```
anecdotes.sort()
```

copy

Write:

```
[...anecdotes].sort()
```

copy

6.12 Better Anecdotes, step 10

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
```

copy

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

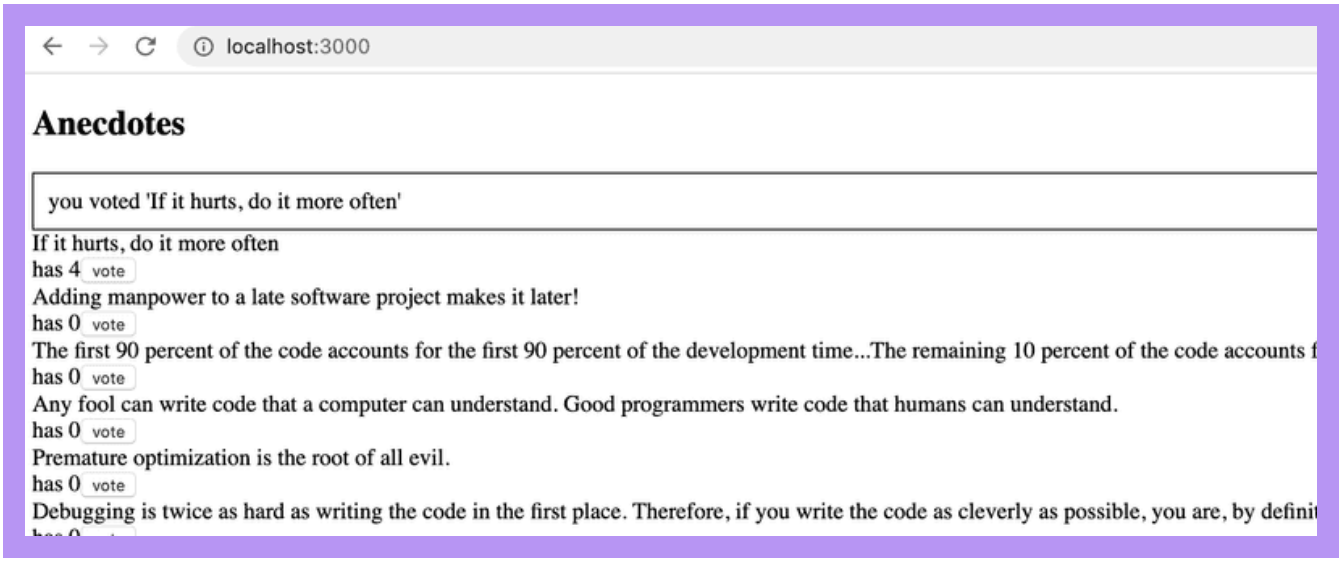
```
import { useSelector } from 'react-redux'

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, step 11

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

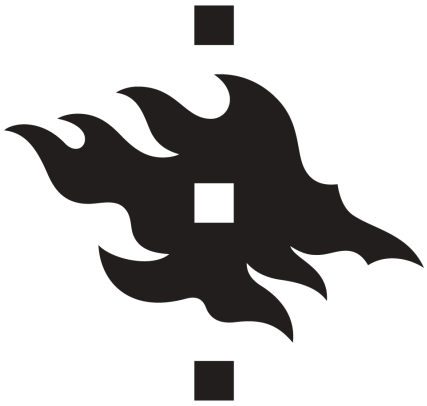
About course

Course contents

FAQ

Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 6 > Communicating with server in a Redux application

c Communicating with server in a Redux application

Let's expand the application so that the notes are stored in the backend. We'll use json-server, familiar from part 2.

The initial state of the database is stored in the file *db.json*, which is placed in the root of the project:

```
{
  "notes": [
    {
      "content": "the app state is in redux store",
      "important": true,
      "id": 1
    },
    {
      "content": "state changes are made with actions",
      "important": false,
      "id": 2
    }
  ]
}
```

copy

We'll install json-server for the project:

```
npm install json-server --save-dev
```

copy

and add the following line to the *scripts* part of the file *package.json*

```
"scripts": {
  "server": "json-server -p3001 --watch db.json",
  // ...
}
```

Now let's launch json-server with the command `npm run server`.

Getting data from the backend

Next, we'll create a method into the file *services/notes.js*, which uses *axios* to fetch data from the backend

```
import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

const getAll = async () => {
  const response = await axios.get(baseUrl)
  return response.data
}

export default { getAll }
```

We'll add axios to the project

```
npm install axios
```

We'll change the initialization of the state in *noteReducer*, so that by default there are no notes:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  // ...
})
```

Let's also add a new action `appendNote` for adding a note object:

```
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
      )
    },
    appendNote(state, action) {
      state.push(action.payload)
    }
  },
})

export const { createNote, toggleImportanceOf, appendNote } = noteSlice.actions

export default noteSlice.reducer
```

A quick way to initialize the notes state based on the data received from the server is to fetch the notes in the *main.jsx* file and dispatch an action using the `appendNote` action creator for each individual note object:

```
// ...
import noteService from './services/notes'
import noteReducer, { appendNote } from './reducers/noteReducer'

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

```
})

noteService.getAll().then(notes =>
  notes.forEach(note => {
    store.dispatch(appendNote(note))
  })
)

// ...
```

Dispatching multiple actions seems a bit impractical. Let's add an action creator `setNotes` which can be used to directly replace the notes array. We'll get the action creator from the `createSlice` function by implementing the `setNotes` action:

```
// ...

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
      )
    },
    appendNote(state, action) {
      state.push(action.payload)
    },
    setNotes(state, action) {
      return action.payload
    }
  },
})

export const { createNote, toggleImportanceOf, appendNote, setNotes } = noteSlice.actions
```

```
export default noteSlice.reducer
```

Now, the code in the *main.jsx* file looks a lot better:

```
// ...
import noteService from './services/notes'
import noteReducer, { setNotes } from './reducers/noteReducer'

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

noteService.getAll().then(notes =>
  store.dispatch(setNotes(notes))
)
```

> **NB:** Why didn't we use await in place of promises and event handlers (registered to `then` -methods)?
>
> Await only works inside *async* functions, and the code in *main.jsx* is not inside a function, so due to the simple nature of the operation, we'll abstain from using *async* this time.

We do, however, decide to move the initialization of the notes into the *App* component, and, as usual, when fetching data from a server, we'll use the *effect hook*.

```
import { useEffect } from 'react'
import NewNote from './components/NewNote'
import Notes from './components/Notes'
import VisibilityFilter from './components/VisibilityFilter'
import noteService from './services/notes'
import { setNotes } from './reducers/noteReducer'
import { useDispatch } from 'react-redux'

const App = () => {
  const dispatch = useDispatch()
  useEffect(() => {
    noteService
      .getAll().then(notes => dispatch(setNotes(notes)))
  }, [])

  return (
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
```

```
      </div>
    )
}

export default App
```

Sending data to the backend

We can do the same thing when it comes to creating a new note. Let's expand the code communicating with the server as follows:

```
const baseUrl = 'http://localhost:3001/notes'

const getAll = async () => {
  const response = await axios.get(baseUrl)
  return response.data
}

const createNew = async (content) => {
  const object = { content, important: false }
  const response = await axios.post(baseUrl, object)
  return response.data
}

export default {
  getAll,
  createNew,
}
```

The method `addNote` of the component *NewNote* changes slightly:

```
import { useDispatch } from 'react-redux'
import { createNote } from '../reducers/noteReducer'
import noteService from '../services/notes'

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

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    const newNote = await noteService.createNew(content)
    dispatch(createNote(newNote))
  }

  return (
    <form onSubmit={addNote}>
```

```
      <input name="note" />
      <button type="submit">add</button>
    </form>
  )
}

export default NewNote
```

Because the backend generates ids for the notes, we'll change the action creator `createNote` in the file *noteReducer.js* accordingly:

```
const noteSlice = createSlice({
  name: 'notes',
  initialState: [],
  reducers: {
    createNote(state, action) {
      state.push(action.payload)
    },
    // ..
  },
})
```

Changing the importance of notes could be implemented using the same principle, by making an asynchronous method call to the server and then dispatching an appropriate action.

The current state of the code for the application can be found on GitHub in the branch *part6-3*.

Exercises 6.14.-6.15.

6.14 Anecdotes and the Backend, step 1

When the application launches, fetch the anecdotes from the backend implemented using json-server.

As the initial backend data, you can use, e.g. this.

6.15 Anecdotes and the Backend, step 2

Modify the creation of new anecdotes, so that the anecdotes are stored in the backend.

Asynchronous actions and Redux Thunk

Our approach is quite good, but it is not great that the communication with the server happens inside the functions of the components. It would be better if the communication could be abstracted away

from the components so that they don't have to do anything else but call the appropriate *action creator*. As an example, *App* would initialize the state of the application as follows:

```
const App = () => {
  const dispatch = useDispatch()

  useEffect(() => {
    dispatch(initializeNotes())
  }, [])

  // ...
}
```

and *NewNote* would create a new note as follows:

```
const NewNote = () => {
  const dispatch = useDispatch()

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  // ...
}
```

In this implementation, both components would dispatch an action without the need to know about the communication with the server that happens behind the scenes. These kinds of *async actions* can be implemented using the Redux Thunk library. The use of the library doesn't need any additional configuration or even installation when the Redux store is created using the Redux Toolkit's `configureStore` function.

With Redux Thunk it is possible to implement *action creators* which return a function instead of an object. The function receives Redux store's `dispatch` and `getState` methods as parameters. This allows for example implementations of asynchronous action creators, which first wait for the completion of a certain asynchronous operation and after that dispatch some action, which changes the store's state.

We can define an action creator `initializeNotes` which initializes the notes based on the data received from the server:

```
// ...
import noteService from '../services/notes'
```

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

export const { createNote, toggleImportanceOf, setNotes, appendNote } = noteSlice.actions

export const initializeNotes = () => {
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch(setNotes(notes))
  }
}

export default noteSlice.reducer
```

In the inner function, meaning the *asynchronous action*, the operation first fetches all the notes from the server and then *dispatches* the `setNotes` action, which adds them to the store.

The component *App* can now be defined as follows:

```
// ...
import { initializeNotes } from './reducers/noteReducer'

const App = () => {
  const dispatch = useDispatch()

  useEffect(() => {
    dispatch(initializeNotes())
  }, [])

  return (
    <div>
      <NewNote />
      <VisibilityFilter />
      <Notes />
    </div>
  )
}
```

The solution is elegant. The initialization logic for the notes has been completely separated from the React component.

Next, let's replace the `createNote` action creator created by the `createSlice` function with an asynchronous action creator:

```
// ...
import noteService from '../services/notes'

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
      }

      return state.map(note =>
        note.id !== id ? note : changedNote
      )
    },
    appendNote(state, action) {
      state.push(action.payload)
    },
    setNotes(state, action) {
      return action.payload
    }
    // createNote definition removed from here!
  },
})

export const { toggleImportanceOf, appendNote, setNotes } = noteSlice.actions

export const initializeNotes = () => {
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch(setNotes(notes))
  }
}

export const createNote = content => {
  return async dispatch => {
    const newNote = await noteService.createNew(content)
    dispatch(appendNote(newNote))
  }
}

export default noteSlice.reducer
```

The principle here is the same: first, an asynchronous operation is executed, after which the action changing the state of the store is *dispatched*.

The component *NewNote* changes as follows:

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

const NewNote = () => {
  const dispatch = useDispatch()
```

```
  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    dispatch(createNote(content))
  }

  return (
    <form onSubmit={addNote}>
      <input name="note" />
      <button type="submit">add</button>
    </form>
  )
}
```

Finally, let's clean up the *main.jsx* file a bit by moving the code related to the creation of the Redux store into its own, *store.js* file:

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

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

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

export default store
```

After the changes, the content of the *main.jsx* is the following:

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

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

The current state of the code for the application can be found on GitHub in the branch *part6-5*.

Redux Toolkit offers a multitude of tools to simplify asynchronous state management. Suitable tools for this use case are for example the `createAsyncThunk` function and the `RTK Query` API.

Exercises 6.16.-6.19.

6.16 Anecdotes and the Backend, step 3

Modify the initialization of the Redux store to happen using asynchronous action creators, which are made possible by the Redux Thunk library.

6.17 Anecdotes and the Backend, step 4

Also modify the creation of a new anecdote to happen using asynchronous action creators, made possible by the Redux Thunk library.

6.18 Anecdotes and the Backend, step 5

Voting does not yet save changes to the backend. Fix the situation with the help of the Redux Thunk library.

6.19 Anecdotes and the Backend, step 6

The creation of notifications is still a bit tedious since one has to do two actions and use the `setTimeout` function:

```
dispatch(setNotification(`new anecdote '${content}'`))
setTimeout(() => {
  dispatch(clearNotification())
}, 5000)
```

copy

Make an action creator, which enables one to provide the notification as follows:

```
dispatch(setNotification(`you voted '${anecdote.content}'`, 10))
```

copy

The first parameter is the text to be rendered and the second parameter is the time to display the notification given in seconds.

Implement the use of this improved notification in your application.

Propose changes to material

Part 6b
Previous part

Part 6d
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 6 > React Query, useReducer and the context

d React Query, useReducer and the context

At the end of this part, we will look at a few more different ways to manage the state of an application.

Let's continue with the note application. We will focus on communication with the server. Let's start the application from scratch. The first version is as follows:

```
const App = () => {
  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    console.log(content)
  }

  const toggleImportance = (note) => {
    console.log('toggle importance of', note.id)
  }

  const notes = []

  return(
    <div>
      <h2>Notes app</h2>
      <form onSubmit={addNote}>
        <input name="note" />
        <button type="submit">add</button>
      </form>
      {notes.map(note =>
        <li key={note.id} onClick={() => toggleImportance(note)}>
```

```
            {note.content}
            <strong> {note.important ? 'important' : ''}</strong>
          </li>
        )}
      </div>
    )
}

export default App
```

The initial code is on GitHub in this repository, in the branch *part6-0*.

Note: By default, cloning the repo will only give you the main branch. To get the initial code from the part6-0 branch, use the following command:

```
git clone --branch part6-0 https://github.com/fullstack-hy2020/query-notes.git
```

copy

Managing data on the server with the React Query library

We shall now use the React Query library to store and manage data retrieved from the server. The latest version of the library is also called TanStack Query, but we stick to the familiar name.

Install the library with the command

```
npm install @tanstack/react-query
```

copy

A few additions to the file *main.jsx* are needed to pass the library functions to the entire application:

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import { QueryClient, QueryClientProvider } from '@tanstack/react-query'

import App from './App'

const queryClient = new QueryClient()

ReactDOM.createRoot(document.getElementById('root')).render(
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>
)
```

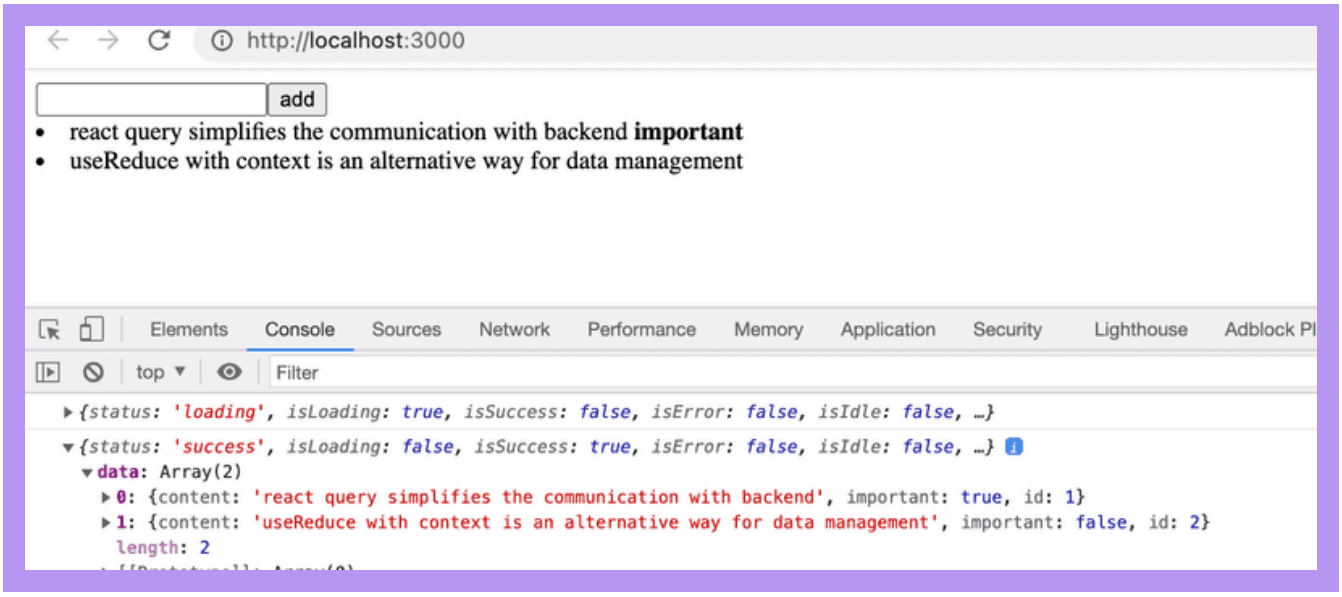
copy

We can now retrieve the notes in the *App* component. The code expands as follows:

```
import { useQuery } from '@tanstack/react-query'
import axios from 'axios'

const App = () => {
  // ...

  const result = useQuery({
    queryKey: ['notes'],
    queryFn: () => axios.get('http://localhost:3001/notes').then(res => res.data)
  })
  console.log(JSON.parse(JSON.stringify(result)))

  if ( result.isLoading ) {
    return <div>loading data...</div>
  }

  const notes = result.data

  return (
    // ...
  )
}
```

Retrieving data from the server is still done in a familiar way with the Axios *get* method. However, the Axios method call is now wrapped in a query formed with the useQuery function. The first parameter of the function call is a string *notes* which acts as a key to the query defined, i.e. the list of notes.

The return value of the *useQuery* function is an object that indicates the status of the query. The output to the console illustrates the situation:



That is, the first time the component is rendered, the query is still in *loading* state, i.e. the associated HTTP request is pending. At this stage, only the following is rendered:

```
<div>loading data...</div>
```

copy

However, the HTTP request is completed so quickly that not even Max Verstappen would be able to see the text. When the request is completed, the component is rendered again. The query is in the state *success* on the second rendering, and the field *data* of the query object contains the data returned by the request, i.e. the list of notes that is rendered on the screen.

So the application retrieves data from the server and renders it on the screen without using the React hooks *useState* and *useEffect* used in chapters 2-5 at all. The data on the server is now entirely under the administration of the React Query library, and the application does not need the state defined with React's *useState* hook at all!

Let's move the function making the actual HTTP request to its own file *requests.js*

```
import axios from 'axios'

export const getNotes = () =>
  axios.get('http://localhost:3001/notes').then(res => res.data)
```

copy

The *App* component is now slightly simplified

```
import { useQuery } from '@tanstack/react-query'
import { getNotes } from './requests'

const App = () => {
  // ...

  const result = useQuery({
    queryKey: ['notes'],
    queryFn: getNotes
  })

  // ...
}
```

copy

The current code for the application is in GitHub in the branch *part6-1*.

Synchronizing data to the server using React Query

Data is already successfully retrieved from the server. Next, we will make sure that the added and modified data is stored on the server. Let's start by adding new notes.

Let's make a function *createNote* to the file *requests.js* for saving new notes:

```
import axios from 'axios'

const baseUrl = 'http://localhost:3001/notes'

export const getNotes = () =>
  axios.get(baseUrl).then(res => res.data)

export const createNote = newNote =>
  axios.post(baseUrl, newNote).then(res => res.data)
```

The *App* component will change as follows

```
import { useQuery, useMutation } from '@tanstack/react-query'
import { getNotes, createNote } from './requests'

const App = () => {
 const newNoteMutation = useMutation({ mutationFn: createNote })

  const addNote = async (event) => {
    event.preventDefault()
    const content = event.target.note.value
    event.target.note.value = ''
    newNoteMutation.mutate({ content, important: true })
  }

  //

}
```

To create a new note, a mutation is defined using the function useMutation:

```
const newNoteMutation = useMutation({ mutationFn: createNote })
```

The parameter is the function we added to the file *requests.js*, which uses Axios to send a new note to the server.

The event handler *addNote* performs the mutation by calling the mutation object's function *mutate* and passing the new note as a parameter:

```
newNoteMutation.mutate({ content, important: true })
```

Our solution is good. Except it doesn't work. The new note is saved on the server, but it is not updated on the screen.

In order to render a new note as well, we need to tell React Query that the old result of the query whose key is the string *notes* should be invalidated.

Fortunately, invalidation is easy, it can be done by defining the appropriate *onSuccess* callback function to the mutation:

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { getNotes, createNote } from './requests'

const App = () => {
  const queryClient = useQueryClient()

  const newNoteMutation = useMutation({
    mutationFn: createNote,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['notes'] })
    },
  })

  // ...
}
```

Now that the mutation has been successfully executed, a function call is made to

```
queryClient.invalidateQueries('notes')
```

This in turn causes React Query to automatically update a query with the key *notes*, i.e. fetch the notes from the server. As a result, the application renders the up-to-date state on the server, i.e. the added note is also rendered.

Let us also implement the change in the importance of notes. A function for updating notes is added to the file *requests.js*:

```
export const updateNote = updatedNote =>
  axios.put(`${baseUrl}/${updatedNote.id}`, updatedNote).then(res => res.data)
```

Updating the note is also done by mutation. The *App* component expands as follows:

```
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query'
import { getNotes, createNote, updateNote } from './requests'

const App = () => {
  // ...
```

```
  const updateNoteMutation = useMutation({
    mutationFn: updateNote,
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['notes'] })
    },
  })

  const toggleImportance = (note) => {
    updateNoteMutation.mutate({...note, important: !note.important })
  }

  // ...
}
```

So again, a mutation was created that invalidated the query *notes* so that the updated note is rendered correctly. Using mutations is easy, the method *mutate* receives a note as a parameter, the importance of which is been changed to the negation of the old value.

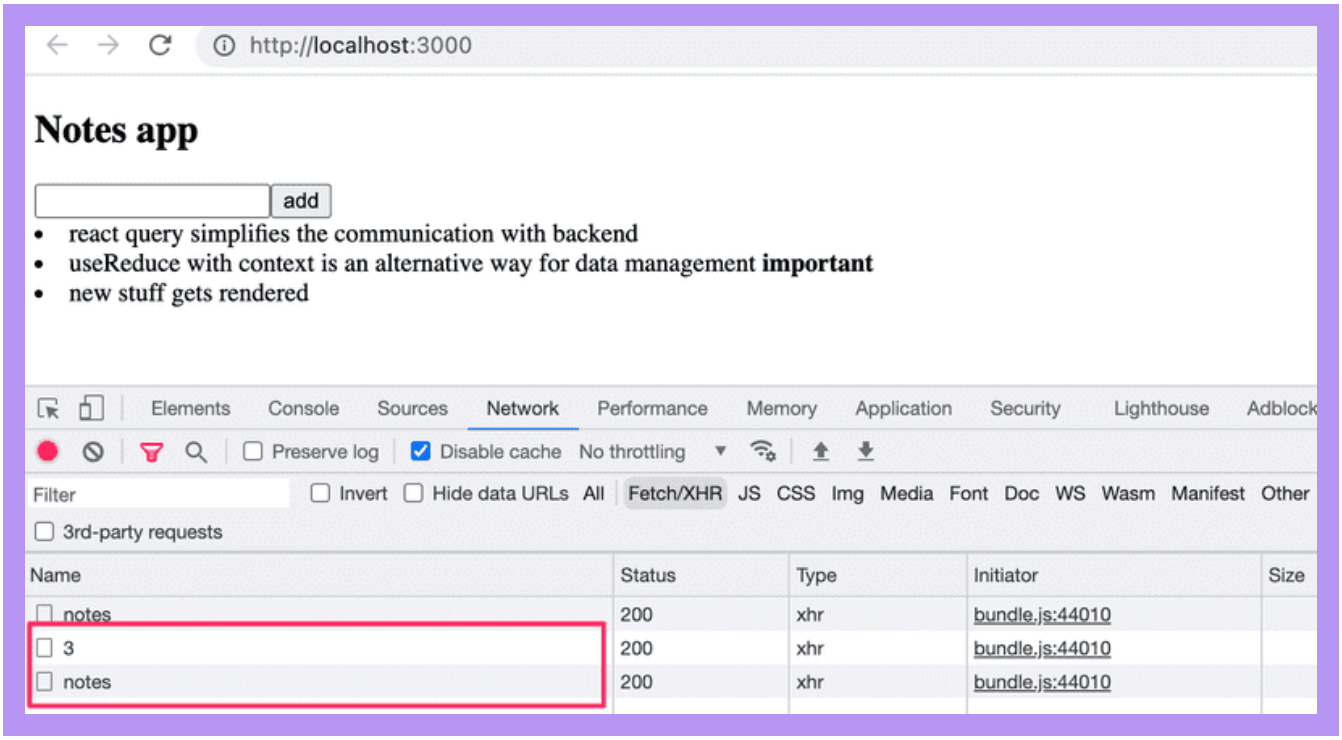
The current code for the application is on GitHub in the branch *part6-2*.

Optimizing the performance

The application works well, and the code is relatively simple. The ease of making changes to the list of notes is particularly surprising. For example, when we change the importance of a note, invalidating the query *notes* is enough for the application data to be updated:

```
  const updateNoteMutation = useMutation({
    mutationFn: updateNote,
    onSuccess: () => {
      queryClient.invalidateQueries('notes')
    },
  })
```

The consequence of this, of course, is that after the PUT request that causes the note change, the application makes a new GET request to retrieve the query data from the server:



If the amount of data retrieved by the application is not large, it doesn't really matter. After all, from a browser-side functionality point of view, making an extra HTTP GET request doesn't really matter, but in some situations it might put a strain on the server.

If necessary, it is also possible to optimize performance by manually updating the query state maintained by React Query.

The change for the mutation adding a new note is as follows:

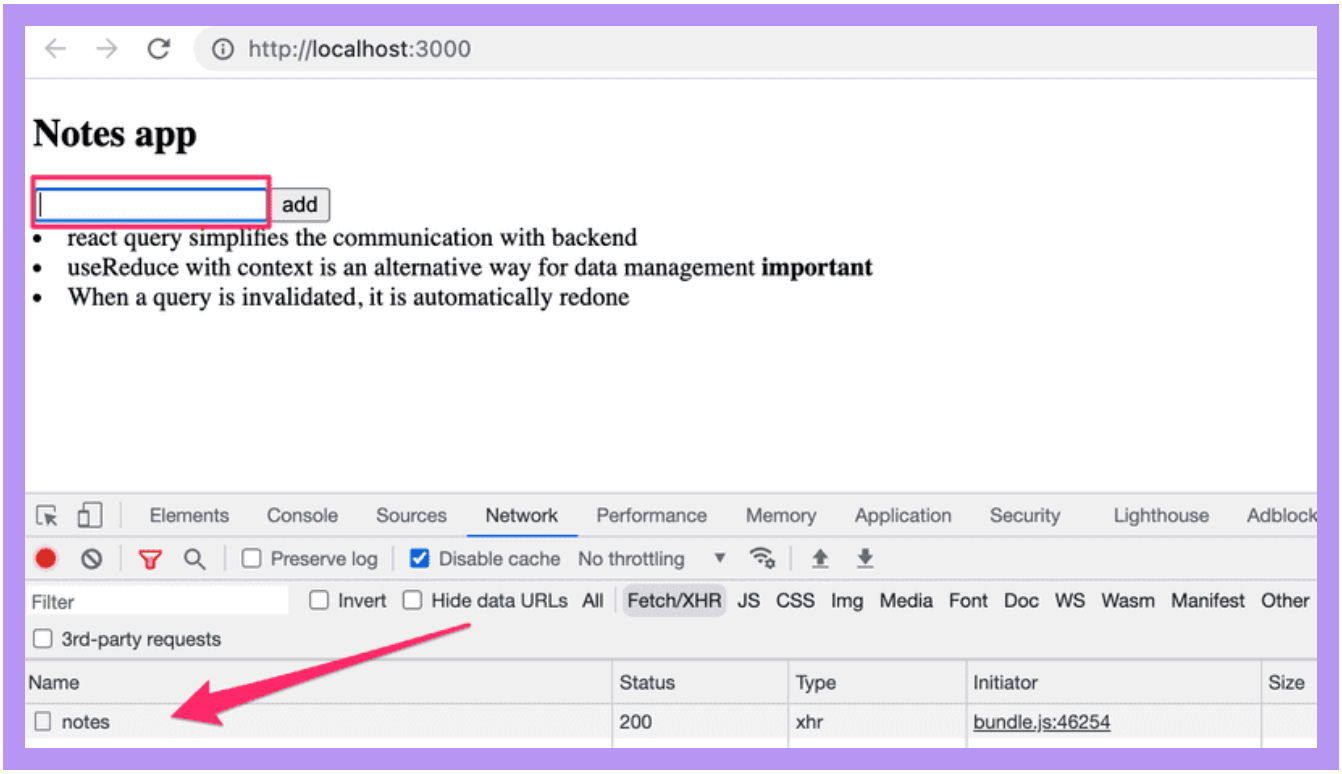
```
const App = () => {
  const queryClient =  useQueryClient()

  const newNoteMutation = useMutation({
    mutationFn: createNote,
    onSuccess: (newNote) => {
      const notes = queryClient.getQueryData(['notes'])
      queryClient.setQueryData(['notes'], notes.concat(newNote))
    }
  })
  // ...
}
```

That is, in the *onSuccess* callback, the *queryClient* object first reads the existing *notes* state of the query and updates it by adding a new note, which is obtained as a parameter of the callback function. The value of the parameter is the value returned by the function *createNote*, defined in the file *requests.js* as follows:

```
export const createNote = newNote =>
  axios.post(baseUrl, newNote).then(res => res.data)
```

It would be relatively easy to make a similar change to a mutation that changes the importance of the note, but we leave it as an optional exercise.

If we closely follow the browser's network tab, we notice that React Query retrieves all notes as soon as we move the cursor to the input field:



What is going on? By reading the documentation, we notice that the default functionality of React Query's queries is that the queries (whose status is *stale*) are updated when *window focus*, i.e. the active element of the application's user interface, changes. If we want, we can turn off the functionality by creating a query as follows:

```
const App = () => {
  // ...
  const result = useQuery({
    queryKey: ['notes'],
    queryFn: getNotes,
    refetchOnWindowFocus: false
  })

  // ...
}
```

If you put a console.log statement to the code, you can see from browser console how often React Query causes the application to be re-rendered. The rule of thumb is that rerendering happens at least whenever there is a need for it, i.e. when the state of the query changes. You can read more about it e.g. here.

The code for the application is in GitHub in the branch *part6-3*.

React Query is a versatile library that, based on what we have already seen, simplifies the application. Does React Query make more complex state management solutions such as Redux unnecessary? No. React Query can partially replace the state of the application in some cases, but as the documentation states

- React Query is a *server-state library*, responsible for managing asynchronous operations between your server and client
- Redux, etc. are *client-state libraries* that can be used to store asynchronous data, albeit inefficiently when compared to a tool like React Query

So React Query is a library that maintains the *server state* in the frontend, i.e. acts as a cache for what is stored on the server. React Query simplifies the processing of data on the server, and can in some cases eliminate the need for data on the server to be saved in the frontend state.

Most React applications need not only a way to temporarily store the served data, but also some solution for how the rest of the frontend state (e.g. the state of forms or notifications) is handled.

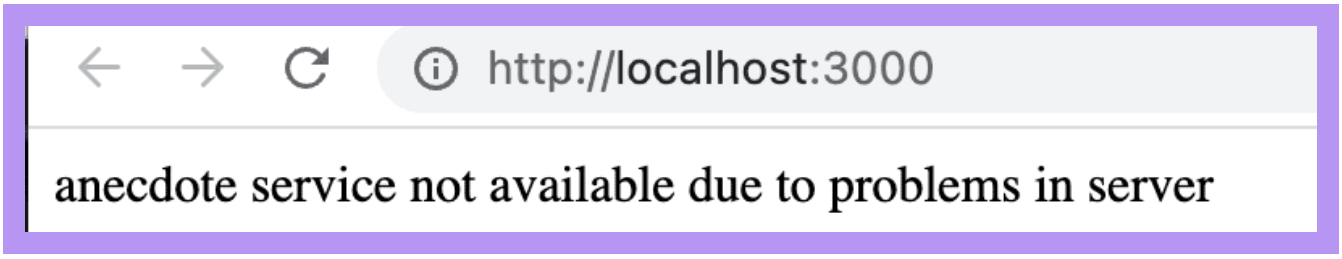
Exercises 6.20.-6.22.

Now let's make a new version of the anecdote application that uses the React Query library. Take this project as your starting point. The project has a ready-installed JSON Server, the operation of which has been slightly modified (Review the `server.js` file for more details. Make sure you're connecting to the correct `PORT`). Start the server with *npm run server*.

Exercise 6.20

Implement retrieving anecdotes from the server using React Query.

The application should work in such a way that if there are problems communicating with the server, only an error page will be displayed:



You can find here info how to detect the possible errors.

You can simulate a problem with the server by e.g. turning off the JSON Server. Please note that in a problem situation, the query is first in the state *isLoading* for a while, because if a request fails, React Query tries the request a few times before it states that the request is not successful. You can optionally specify that no retries are made:

```
const result = useQuery(
  {
    queryKey: ['anecdotes'],
    queryFn: getAnecdotes,
    retry: false
  }
)
```

or that the request is retried e.g. only once:

```
const result = useQuery(
  {
    queryKey: ['anecdotes'],
    queryFn: getAnecdotes,
    retry: 1
  }
)
```

Exercise 6.21

Implement adding new anecdotes to the server using React Query. The application should render a new anecdote by default. Note that the content of the anecdote must be at least 5 characters long, otherwise the server will reject the POST request. You don't have to worry about error handling now.

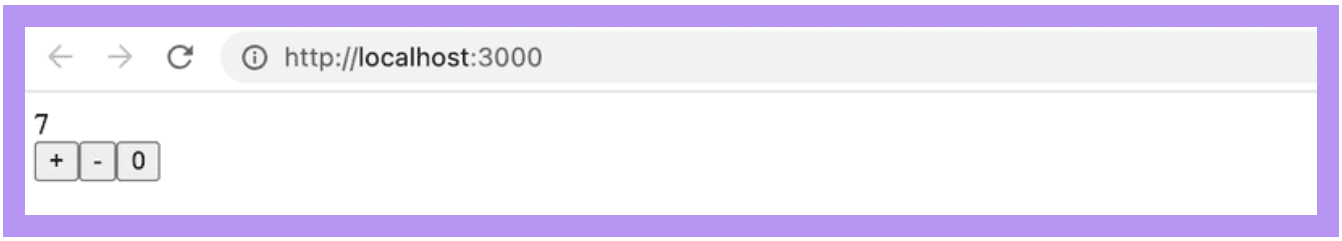
Exercise 6.22

Implement voting for anecdotes using again the React Query. The application should automatically render the increased number of votes for the voted anecdote.

useReducer

So even if the application uses React Query, some kind of solution is usually needed to manage the rest of the frontend state (for example, the state of forms). Quite often, the state created with *useState* is a sufficient solution. Using Redux is of course possible, but there are other alternatives.

Let's look at a simple counter application. The application displays the counter value, and offers three buttons to update the counter status:



We shall now implement the counter state management using a Redux-like state management mechanism provided by React's built-in useReducer hook. Code looks like the following:

```
import { useReducer } from 'react'

const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
        return state + 1
    case "DEC":
        return state - 1
    case "ZERO":
        return 0
    default:
        return state
  }
}

const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <div>
      <div>{counter}</div>
      <div>
        <button onClick={() => counterDispatch({ type: "INC"})}>+</button>
        <button onClick={() => counterDispatch({ type: "DEC"})}>-</button>
        <button onClick={() => counterDispatch({ type: "ZERO"})}>0</button>
      </div>
    </div>
  )
}

export default App
```

The hook useReducer provides a mechanism to create a state for an application. The parameter for creating a state is the reducer function that handles state changes, and the initial value of the state:

```
const [counter, counterDispatch] = useReducer(counterReducer, 0)
```

The reducer function that handles state changes is similar to Redux's reducers, i.e. the function gets as parameters the current state and the action that changes the state. The function returns the new state updated based on the type and possible contents of the action:

```
const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
        return state + 1
    case "DEC":
        return state - 1
    case "ZERO":
        return 0
    default:
        return state
  }
}
```

copy

In our example, actions have nothing but a type. If the action's type is *INC*, it increases the value of the counter by one, etc. Like Redux's reducers, actions can also contain arbitrary data, which is usually put in the action's *payload* field.

The function *useReducer* returns an array that contains an element to access the current value of the state (first element of the array), and a *dispatch* function (second element of the array) to change the state:

```
const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <div>
      <div>{counter}</div>
      <div>
        <button onClick={() => counterDispatch({ type: "INC" })}>+</button>
        <button onClick={() => counterDispatch({ type: "DEC" })}>-</button>
        <button onClick={() => counterDispatch({ type: "ZERO" })}>0</button>
      </div>
    </div>
  )
}
```

copy

As can be seen the state change is done exactly as in Redux, the dispatch function is given the appropriate state-changing action as a parameter:

```
counterDispatch({ type: "INC" })
```

copy

The current code for the application is in the repository https://github.com/fullstack-hy2020/hook-counter in the branch *part6-1*.

Using context for passing the state to components

If we want to split the application into several components, the value of the counter and the dispatch function used to manage it must also be passed to the other components. One solution would be to pass these as props in the usual way:

```
const Display = ({ counter }) => {
  return <div>{counter}</div>
}

const Button = ({ dispatch, type, label }) => {
  return (
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  )
}

const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <div>
      <Display counter={counter}/>
      <div>
        <Button dispatch={counterDispatch} type='INC' label='+' />
        <Button dispatch={counterDispatch} type='DEC' label='-' />
        <Button dispatch={counterDispatch} type='ZERO' label='0' />
      </div>
    </div>
  )
}
```

The solution works, but is not optimal. If the component structure gets complicated, e.g. the dispatcher should be forwarded using props through many components to the components that need it, even though the components in between in the component tree do not need the dispatcher. This phenomenon is called *prop drilling*.

React's built-in Context API provides a solution for us. React's context is a kind of global state of the application, to which it is possible to give direct access to any component app.

Let us now create a context in the application that stores the state management of the counter.

The context is created with React's hook createContext. Let's create a context in the file *CounterContext.jsx*:

```
import { createContext } from 'react'

const CounterContext = createContext()

export default CounterContext
```

The *App* component can now *provide* a context to its child components as follows:

```
import CounterContext from './CounterContext'

const App = () => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <CounterContext.Provider value={[counter, counterDispatch]}>
      <Display />
      <div>
        <Button type='INC' label='+' />
        <Button type='DEC' label='-' />
        <Button type='ZERO' label='0' />
      </div>
    </CounterContext.Provider>
  )
}
```

As can be seen, providing the context is done by wrapping the child components inside the *CounterContext.Provider* component and setting a suitable value for the context.

The context value is now set to be an array containing the value of the counter, and the *dispatch* function.

Other components now access the context using the `useContext` hook:

```
import { useContext } from 'react'
import CounterContext from './CounterContext'

const Display = () => {
  const [counter] = useContext(CounterContext)
  return <div>
    {counter}
  </div>
}

const Button = ({ type, label }) => {
  const [counter, dispatch] = useContext(CounterContext)
  return (
    <button onClick={() => dispatch({ type })}>
      {label}
```

```
      </button>
    )
}
```

The current code for the application is in GitHub in the branch *part6-2*.

Defining the counter context in a separate file

Our application has an annoying feature, that the functionality of the counter state management is partly defined in the *App* component. Now let's move everything related to the counter to *CounterContext.jsx*:

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

const counterReducer = (state, action) => {
  switch (action.type) {
    case "INC":
        return state + 1
    case "DEC":
        return state - 1
    case "ZERO":
        return 0
    default:
        return state
  }
}

const CounterContext = createContext()

export const CounterContextProvider = (props) => {
  const [counter, counterDispatch] = useReducer(counterReducer, 0)

  return (
    <CounterContext.Provider value={[counter, counterDispatch] }>
      {props.children}
    </CounterContext.Provider>
  )
}

export default CounterContext
```

The file now exports, in addition to the *CounterContext* object corresponding to the context, the *CounterContextProvider* component, which is practically a context provider whose value is a counter and a dispatcher used for its state management.

Let's enable the context provider by making a change in *main.jsx*:

```
import ReactDOM from 'react-dom/client'
import App from './App'
import { CounterContextProvider } from './CounterContext'
```

```
ReactDOM.createRoot(document.getElementById('root')).render(
  <CounterContextProvider>
    <App />
  </CounterContextProvider>
)
```

Now the context defining the value and functionality of the counter is available to *all* components of the application.

The *App* component is simplified to the following form:

```
import Display from './components/Display'
import Button from './components/Button'

const App = () => {
  return (
    <div>
      <Display />
      <div>
        <Button type='INC' label='+' />
        <Button type='DEC' label='-' />
        <Button type='ZERO' label='0' />
      </div>
    </div>
  )
}

export default App
```

The context is still used in the same way, e.g. the component *Button* is defined as follows:

```
import { useContext } from 'react'
import CounterContext from '../CounterContext'

const Button = ({ type, label }) => {
  const [counter, dispatch] = useContext(CounterContext)
  return (
    <button onClick={() => dispatch({ type })}>
      {label}
    </button>
  )
}

export default Button
```

The *Button* component only needs the *dispatch* function of the counter, but it also gets the value of the counter from the context using the function *useContext*:

```
  const [counter, dispatch] = useContext(CounterContext)
```

copy

This is not a big problem, but it is possible to make the code a bit more pleasant and expressive by defining a couple of helper functions in the *CounterContext* file:

```
import { createContext, useReducer, useContext } from 'react'

const CounterContext = createContext()

// ...

export const useCounterValue = () => {
  const counterAndDispatch = useContext(CounterContext)
  return counterAndDispatch[0]
}

export const useCounterDispatch = () => {
  const counterAndDispatch = useContext(CounterContext)
  return counterAndDispatch[1]
}

// ...
```

copy

With the help of these helper functions, it is possible for the components that use the context to get hold of the part of the context that they need. The *Display* component changes as follows:

```
import { useCounterValue } from '../CounterContext'

const Display = () => {
  const counter = useCounterValue()
  return <div>
    {counter}
  </div>
}


export default Display
```

copy

Component *Button* becomes:

```
import { useCounterDispatch } from '../CounterContext'

const Button = ({ type, label }) => {
  const dispatch = useCounterDispatch()
  return (
```

copy

```
      <button onClick={() => dispatch({ type })}>
        {label}
      </button>
    )
}

export default Button
```

The solution is quite elegant. The entire state of the application, i.e. the value of the counter and the code for managing it, is now isolated in the file *CounterContext*, which provides components with well-named and easy-to-use auxiliary functions for managing the state.

The final code for the application is in GitHub in the branch *part6-3*.

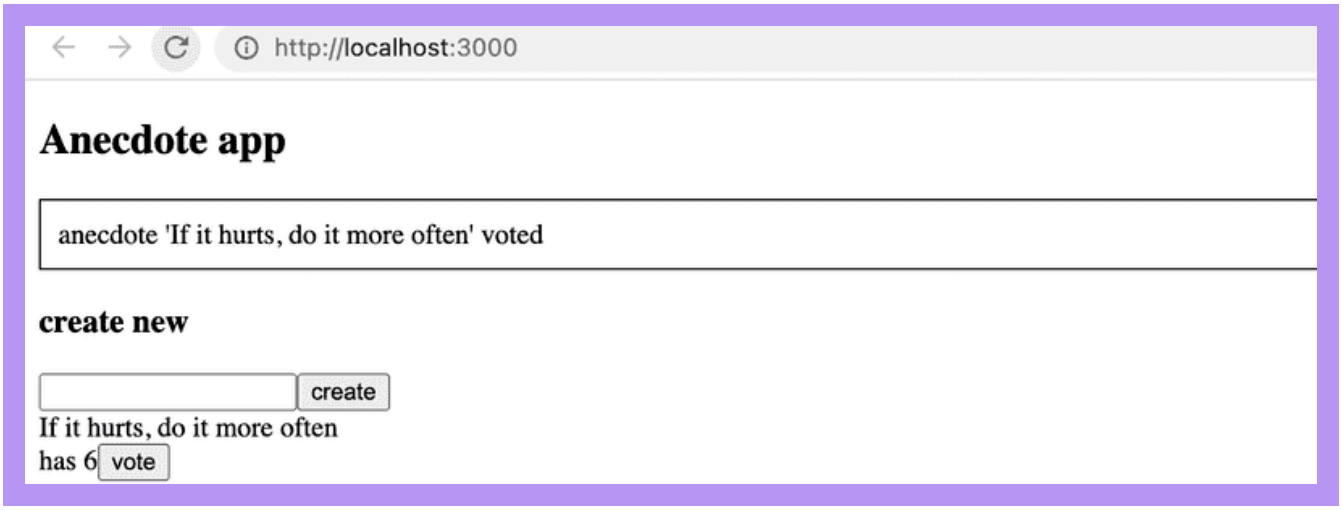
As a technical detail, it should be noted that the helper functions *useCounterValue* and *useCounterDispatch* are defined as custom hooks, because calling the hook function *useContext* is possible only from React components or custom hooks. Custom hooks are JavaScript functions whose name must start with the word `use`. We will return to custom hooks in a little more detail in part 7 of the course.

Exercises 6.23.-6.24.

Exercise 6.23.

The application has a *Notification* component for displaying notifications to the user.

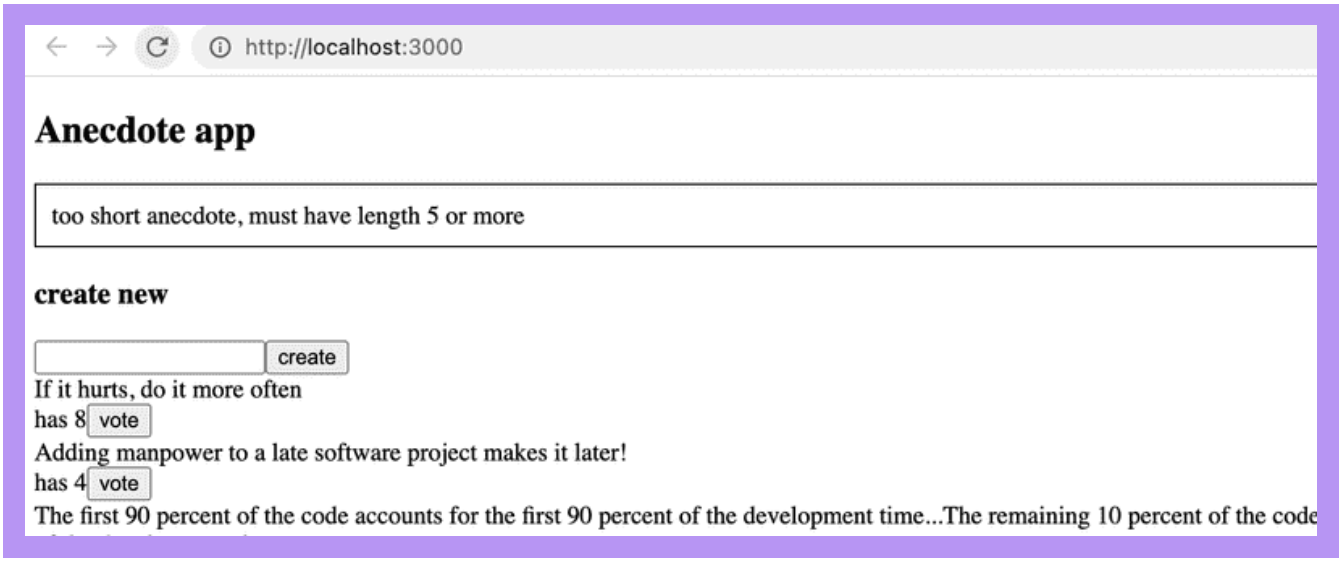
Implement the application's notification state management using the useReducer hook and context. The notification should tell the user when a new anecdote is created or an anecdote is voted on:



The notification is displayed for five seconds.

Exercise 6.24.

As stated in exercise 6.21, the server requires that the content of the anecdote to be added is at least 5 characters long. Now implement error handling for the insertion. In practice, it is sufficient to display a notification to the user in case of a failed POST request:



The error condition should be handled in the callback function registered for it, see here how to register a function.

This was the last exercise for this part of the course and it's time to push your code to GitHub and mark all of your completed exercises to the exercise submission system.

Which state management solution to choose?

In chapters 1-5, all state management of the application was done using React's hook *useState*. Asynchronous calls to the backend required the use of the *useEffect* hook in some situations. In principle, nothing else is needed.

A subtle problem with a solution based on a state created with the *useState* hook is that if some part of the application's state is needed by multiple components of the application, the state and the functions for manipulating it must be passed via props to all components that handle the state. Sometimes props need to be passed through multiple components, and the components along the way may not even be interested in the state in any way. This somewhat unpleasant phenomenon is called *prop drilling*.

Over the years, several alternative solutions have been developed for state management of React applications, which can be used to ease problematic situations (e.g. prop drilling). However, no solution has been "final", all have their own pros and cons, and new solutions are being developed all the time.

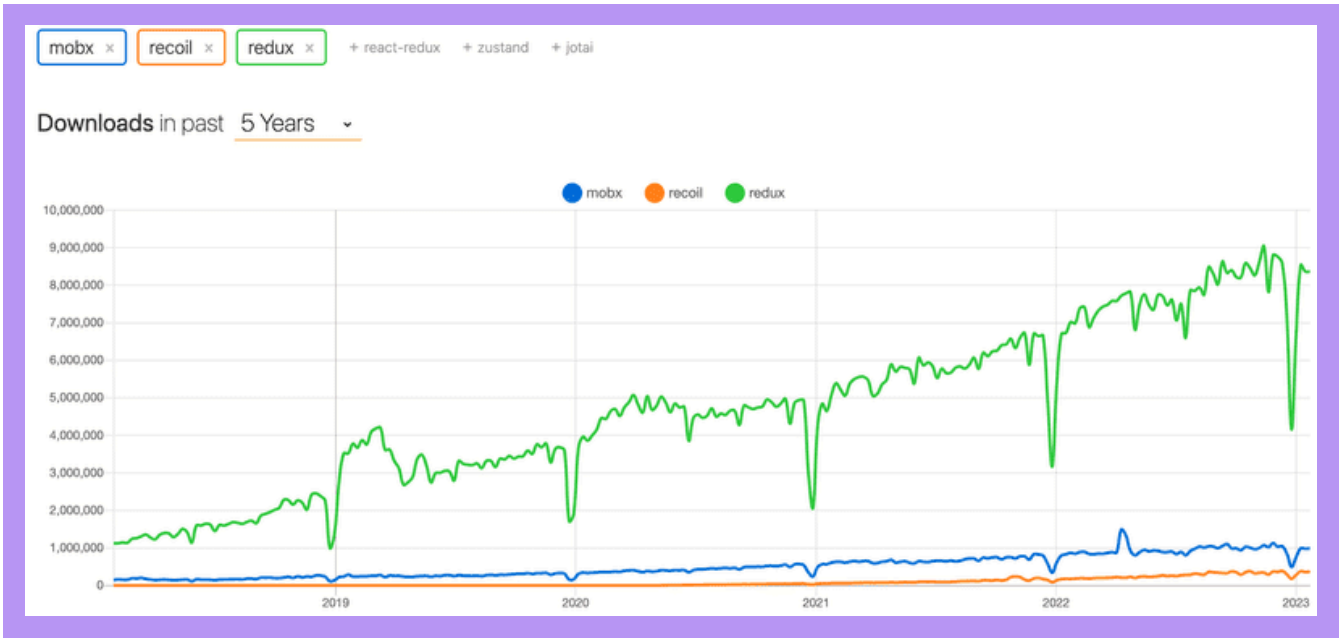
The situation may confuse a beginner and even an experienced web developer. Which solution should be used?

For a simple application, *useState* is certainly a good starting point. If the application is communicating with the server, the communication can be handled in the same way as in chapters 1-5, using the state

of the application itself. Recently, however, it has become more common to move the communication and associated state management at least partially under the control of React Query (or some other similar library). If you are concerned about useState and the prop drilling it entails, using context may be a good option. There are also situations where it may make sense to handle some of the state with useState and some with contexts.

The most comprehensive and robust state management solution is Redux, which is a way to implement the so-called Flux architecture. Redux is slightly older than the solutions presented in this section. The rigidity of Redux has been the motivation for many new state management solutions, such as React's *useReducer*. Some of the criticisms of Redux's rigidity have already become obsolete thanks to the Redux Toolkit.

Over the years, there have also been other state management libraries developed that are similar to Redux, such as the newer entrant Recoil and the slightly older MobX. However, according to Npm trends, Redux still clearly dominates, and in fact seems to be increasing its lead:



Also, Redux does not have to be used in its entirety in an application. It may make sense, for example, to manage the form state outside of Redux, especially in situations where the state of a form does not affect the rest of the application. It is also perfectly possible to use Redux and React Query together in the same application.

The question of which state management solution should be used is not at all straightforward. It is impossible to give a single correct answer. It is also likely that the selected state management solution may turn out to be suboptimal as the application grows to such an extent that the solution has to be changed even if the application has already been put into production use.

Propose changes to material

Part 6c
Previous part

Part 7
Next part


HOUSTON