c Communicating with server in a redux application

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

The initial state of the database is stored into 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
    }
  ]
}
```

We'll install json-server for the project...

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

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` .

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*, such that by default there are no notes:

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

A quick way to initialize the state based on the data on the server is to fetch the notes in the file *index.js* and dispatch the action *NEW_NOTE* for each of them:

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

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

const store = createStore(reducer, composeWithDevTools())

noteService.getAll().then(notes =>
  notes.forEach(note => {
    store.dispatch({ type: 'NEW_NOTE', data: note })
  })
)

// ...
```

Let's add support in the reducer for the action *INIT_NOTES*, using which the initialization can be done by dispatching a single action. Let's also create an action creator function `initializeNotes`.

```
// ...
const noteReducer = (state = [], action) => {
  console.log('ACTION:', action)
  switch (action.type) {
    case 'NEW_NOTE':
      return [...state, action.data]
    case 'INIT_NOTES':
      return action.data
    // ...
  }
}

export const initializeNotes = (notes) => {
  return {
    type: 'INIT_NOTES',
    data: notes,
  }
}

// ...
```

index.js simplifies:

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

noteService.getAll().then(notes =>
  store.dispatch(initializeNotes(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 *index.js* 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*.

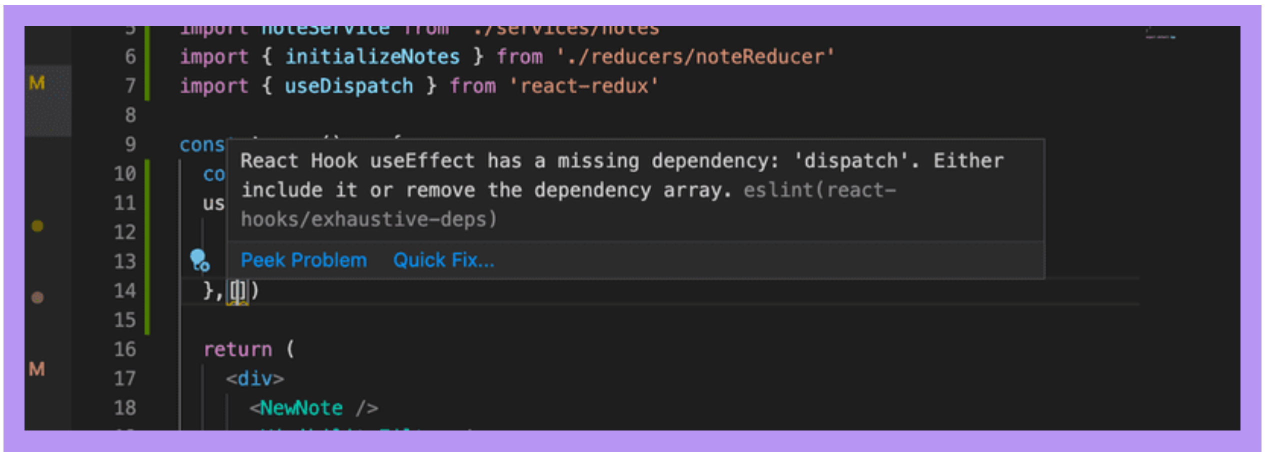
```
import React, {useEffect} from 'react'
import NewNote from './components/NewNote'
import Notes from './components/Notes'
import VisibilityFilter from './components/VisibilityFilter'
import noteService from './services/notes'
import { initializeNotes } from './reducers/noteReducer'
import { useDispatch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

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

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

export default App
```

Using the useEffect hook causes an eslint-warning:



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

cons
  co
  us

  },[])

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

React Hook useEffect has a missing dependency: 'dispatch'. Either include it or remove the dependency array. eslint(react-hooks/exhaustive-deps)

We can get rid of it by doing the following:

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

  // ...
}
```

Now the variable *dispatch* we define in the `App` component, which practically is the dispatch function of the redux-store, has been added to the array useEffect receives as a parameter. If the value of the dispatch-variable would change during runtime, the effect would be executed again. This however cannot happen in our application, so the warning is unnecessary.

Another way to get rid of the warning would be to disable eslint on that line:

```
const App = () => {
  const dispatch = useDispatch()
  useEffect(() => {
    noteService
      .getAll().then(notes => dispatch(initializeNotes(notes)))
  },[]) // eslint-disable-line react-hooks/exhaustive-deps

  // ...
}
```

Generally disabling eslint when it throws a warning is not a good idea. Even though the eslint rule in question has caused some arguments, we will use the first solution.

More about the need to define the hooks dependencies in the react documentation.

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 React from 'react'
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`

```
export const createNote = (data) => {
  return {
    type: 'NEW_NOTE',
    data,
  }
}
```

Changing the importance of notes could be implemented using the same principle, meaning 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 6.13.-6.14.

6.13 Anecdotes and the backend, step1

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.

6.14 Anecdotes and the backend, step2

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

Asynchronous actions and redux thunk

Our approach is OK, 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, such 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()))
  },[dispatch])

  // ...
}
```

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))
  }

  // ...
}
```

Both components would only use the function provided to them as a prop without caring about the communication with the server that is happening in the background.

Now let's install the redux-thunk-library, which enables us to create *asynchronous actions*. Installation is done with the command:

```
npm install redux-thunk
```

The redux-thunk-library is a so-called *redux-middleware*, which must be initialized along with the initialization of the store. While we're here, let's extract the definition of the store into its own file *src/store.js*:

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
import thunk from 'redux-thunk'
import { composeWithDevTools } from 'redux-devtools-extension'

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

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

const store = createStore(
  reducer,
  composeWithDevTools(
    applyMiddleware(thunk)
  )
)

export default store
```

After the changes the file *src/index.js* looks like this

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

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

Thanks to redux-thunk, it is possible to define *action creators* so that they return a function having the *dispatch*-method of redux-store as its parameter. As a result of this, one can make asynchronous action creators, which first wait for some operation to finish, after which they then dispatch the real action.

Now we can define the action creator, *initializeNotes*, that initializes the state of the notes as follows:

```
export const initializeNotes = () => {
  return async dispatch => {
    const notes = await noteService.getAll()
    dispatch({
      type: 'INIT_NOTES',
      data: notes,
    })
  }
}
```

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

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

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

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

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

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

The action creator `createNote`, which adds a new note looks like this

```
export const createNote = content => {
  return async dispatch => {
    const newNote = await noteService.createNew(content)
    dispatch({
      type: 'NEW_NOTE',
      data: newNote,
    })
  }
}
```

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:

```
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">lisää</button>
    </form>
  )
}
```

The current state of the code for the application can be found on github in the branch *part6-4*.

Exercises 6.15.-6.18.

6.15 Anecdotes and the backend, step3

Modify the initialization of redux-store to happen using asynchronous action creators, which are made possible by the *redux-thunk*-library.

6.16 Anecdotes and the backend, step4

Also modify the creation of a new anecdote to happen using asynchronous action creators, made possible by the *redux-thunk*-library.

6.17 Anecdotes and the backend, step5

Voting does not yet save changes to the backend. Fix the situation with the help of the *redux-thunk*-library.

6.18 Anecdotes and the backend, step6

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)
```

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

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

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

Course contents

FAQ

Partners

Challenge