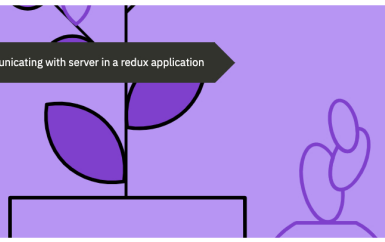

c Communicating with server in a redux application

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
    }
  ]
}
```

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`.

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

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, 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.

6.15 Anecdotes and the backend, step2

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 between 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:

```
// ...                                                                    copy
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:

```
// ...                                                                    copy
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:

```
// ...                                                                    copy
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:

```
// ...                                                                    copy
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, step3

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, step4

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, step5

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, 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 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