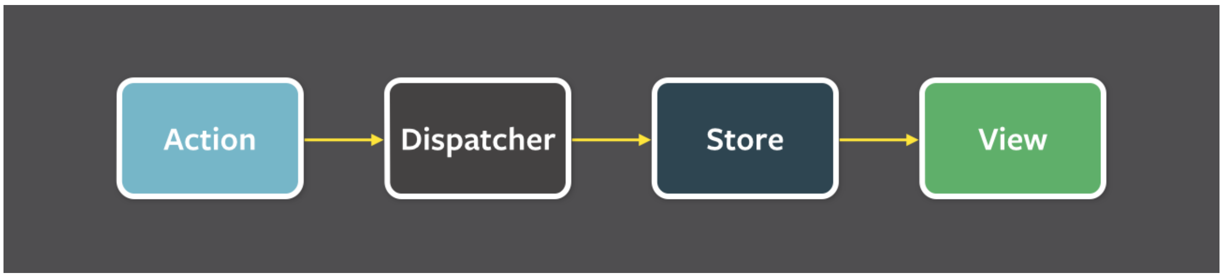
a Flux-architecture and Redux

So far, we have followed the state management conventions recommended by React. We have placed the state and the methods for handling it to the root component of the application. 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.

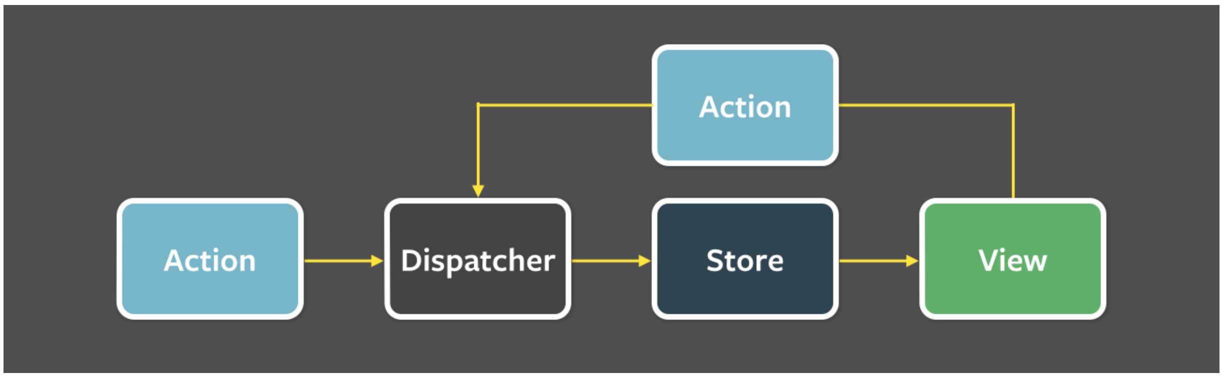
Flux-architecture

Facebook developed the Flux- architecture to make state management easier. In Flux, the state is separated completely from the React-components 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 rerendering the view again:

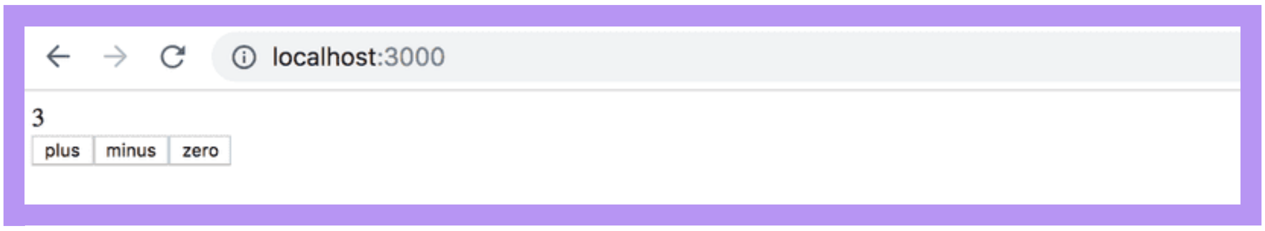


Flux offers a standard way for how and where the application's state is kept and how it is modified.

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 create-react-app-application and install redux with the command

```
npm install redux
```

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

The whole state of the application is stored into *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'
}
```

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 which 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. Reducer returns a *new state* based on the actions type.

Let's change the code a bit. It is customary to use the switch -command instead of ifs in 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
  }
}
```

Reducer is never supposed to be called directly from the applications code. Reducer 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'})
```

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())
```

would print the following to the console

```
0
3
-1
```

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

The third important method the store has is `subscribe`, which is used to create callback functions the store calls when its state is changed.

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)
})
```

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, so *store* is straight 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'
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 renderApp = () => {
  ReactDOM.render(<App />, document.getElementById('root'))
}
```

```
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 actionhandlers of the buttons *dispatch* the right actions to the store.

When the state in the store is changed, React is not able to automatically rerender 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.

Redux-notes

Our aim is 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.data)
    return state
  }

  return state
}

const store = createStore(noteReducer)

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

store.dispatch({
  type: 'NEW_NOTE',
  data: {
    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 *data*, which contains the note to be added:

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

Pure functions, immutable

The initial version of reducer is very simple:

```
const noteReducer = (state = [], action) => {
  if (action.type === 'NEW_NOTE') {
    state.push(action.data)
    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 of Redux reducer 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.data)` 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.data)
  }

  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.

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

```
{
  type: 'TOGGLE_IMPORTANCE',
  data: {
    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*.

To make testing easier, we'll first move the reducer's code to its own module to 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:

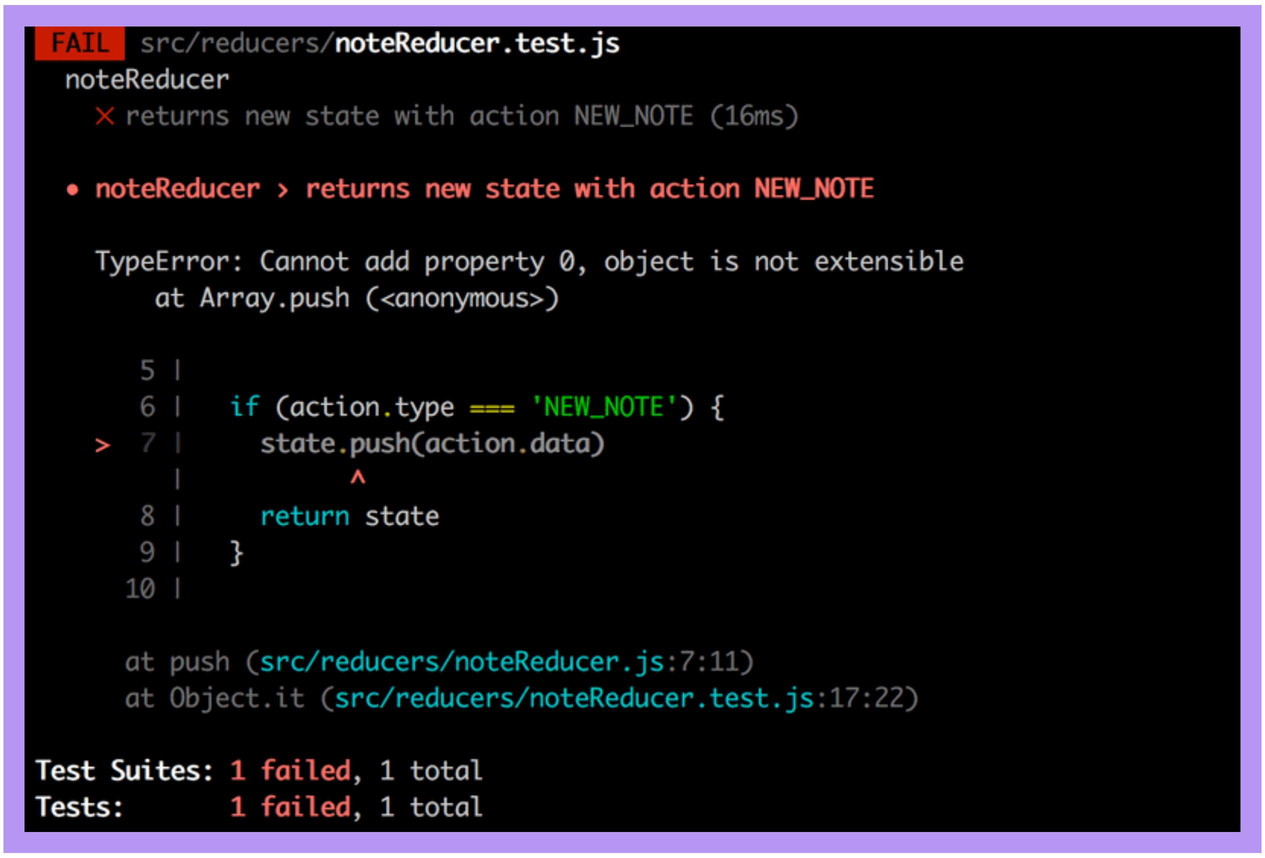
```
import noteReducer from './noteReducer'
import deepFreeze from 'deep-freeze'

describe('noteReducer', () => {
  test('returns new state with action NEW_NOTE', () => {
    const state = []
    const action = {
      type: 'NEW_NOTE',
      data: {
        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.data)
  })
})
```

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',
    data: {
      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',
  data: {
    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.data)
    case 'TOGGLE_IMPORTANCE': {
      const id = action.data.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 which 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)
```

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
}
```

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
)
```

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 it returns with Arrays `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.data]
    case 'TOGGLE_IMPORTANCE':
      // ...
    default:
    return state
  }
}
```

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 project from this repository https://github.com/fullstack-hy/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, step1

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
  }
  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 a sensible original state when it is called so that the first parameter *state*, which represents the previous state, is *undefined*.

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

Note that since all the code is in the file *index.js* and you have to manually reload the page after each change since the automatic reloading of the browser content does not work for that page!

Uncontrolled form

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

```
const generateId = () =>
  Math.floor(Math.random() * 1000000)

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

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

It is actually not necessary for React-components to know the Redux action types and forms. Let's separate creating actions into their own functions:

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

const toggleImportanceOf = (id) => {
  return {
    type: 'TOGGLE_IMPORTANCE',
    data: { 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 own 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 components. First we will look into the newest, and possibly the easiest way using the hooks-api of the react-redux library.

First we install react-redux

```
npm install react-redux
```

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

`Index.js` becomes:

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

const store = createStore(noteReducer)

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

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. File looks like this:

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

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

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

export const toggleImportanceOf = (id) => {
  return {
    type: 'TOGGLE_IMPORTANCE',
    data: { 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'
```

Code for the *App* component

```
import React from 'react'
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
```

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',
  data: { 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 *index.js*. 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))
```

More components

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

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

const NewNote = (props) => {
  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.js* file):

```
import React from 'react'
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 kind 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.

We will return to the presentational/container division later in this part.

The code of the Redux application can be found on Github, 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-hy/redux-anecdotes to base your solution on.

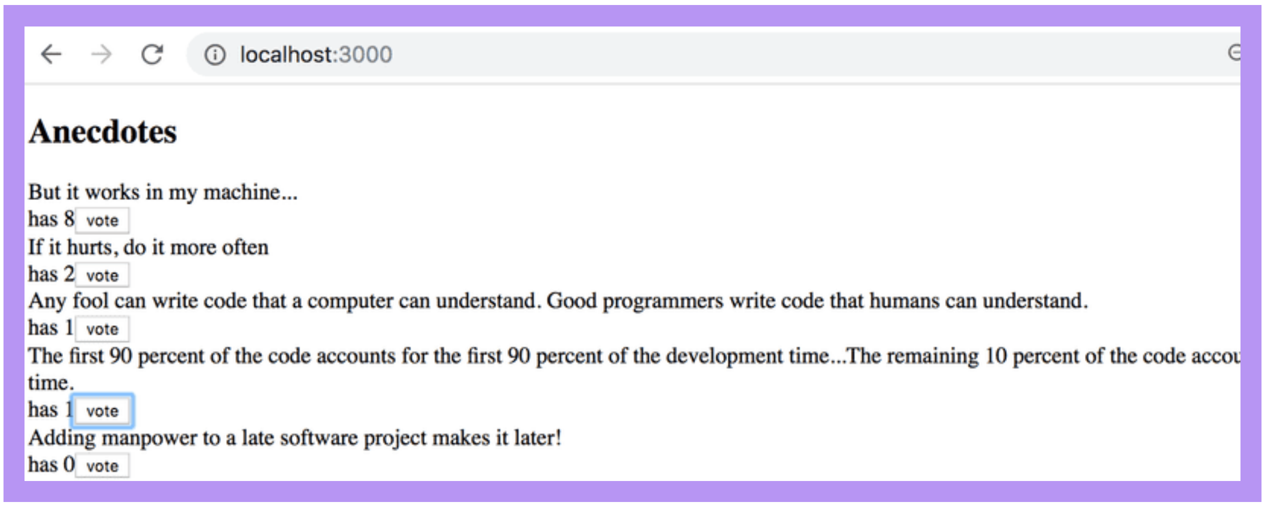
If you clone the project into an existing git-repository, *remove the git-configuration of the cloned* *application:*

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

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

```
npm install
npm start
```

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



Anecdotes

But it works in my machine...
has 8 vote
If it hurts, do it more often
has 2 vote
Any fool can write code that a computer can understand. Good programmers write code that humans can understand.
has 1 vote
The first 90 percent of the code accounts for the first 90 percent of the development time...The remaining 10 percent of the code accou
time.
has 1 vote
Adding manpower to a late software project makes it later!
has 0 vote

6.3: anecdotes, step1

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

6.4: anecdotes, step2

Implement the functionality for adding new anecdotes.

You can keep the form uncontrolled, like we did earlier.

6.5*: anecdotes, step3

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

6.6: anecdotes, step4

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 like we have been doing since the chapter action creators.

6.7: anecdotes, step5

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

6.8: anecdotes, step6

Separate the rendering of the anecdote list into its own 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 React from 'react'
import AnecdoteForm from './components/AnecdoteForm'
import AnecdoteList from './components/AnecdoteList'

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

export default App
```

About course
Course contents
FAQ
Partners
Challenge

UNIVERSITY OF HELSINKI

pose changes to material

Part 5
Previous part

Part 6b
Next part