NOTE: this is the old ending section of the part 6 that has 30th January 2023 been replaced with material about React Query, useReducer and context. This section remains here for a couple of weeks.

If you have started with the exercises (6.19-6.21) that use the Redux connect you may continue with those. If you have not yet started, I recommend to proceed with the new section.

So far we have used our Redux store with the help of the hook API from react-redux. Practically this has meant using the useSelector and useDispatch functions.

To finish this part we will look into another older and more complicated way to use Redux, the connect function provided by react-redux.

*In new applications, use the hook API*. Knowing how to use connect though is useful when maintaining older projects using Redux.

## Using the connect function to share the Redux store to

## components

Let's modify the *Notes* component so that instead of using the hook API (the `useDispatch` and `useSelector` functions ) it uses the `connect` function. We have to modify the following parts of the component:

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

const Notes = () => {
  const dispatch = useDispatch()
  const notes = useSelector(({filter, notes}) => {
    if ( filter === 'ALL' ) {
      return notes
    }
    return filter === 'IMPORTANT'
      ? notes.filter(note => note.important)
      : notes.filter(note => !note.important)
  })

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

export default Notes
```

The `connect` function can be used for transforming "regular" React components so that the state of the Redux store can be "mapped" into the component's props.

Let's first use the connect function to transform our *Notes* component into a *connected component*:

```
import { connect } from 'react-redux'
import { toggleImportanceOf } from '../reducers/noteReducer'

const Notes = () => {
  // ...
}
```

```
const ConnectedNotes = connect()(Notes)
export default ConnectedNotes
```

The module exports the *connected component* that works exactly like the previous regular component for now.

The component needs the list of notes and the value of the filter from the Redux store. The `connect` function accepts a so-called mapStateToProps function as its first parameter. The function can be used for defining the props of the *connected component* that are based on the state of the Redux store.

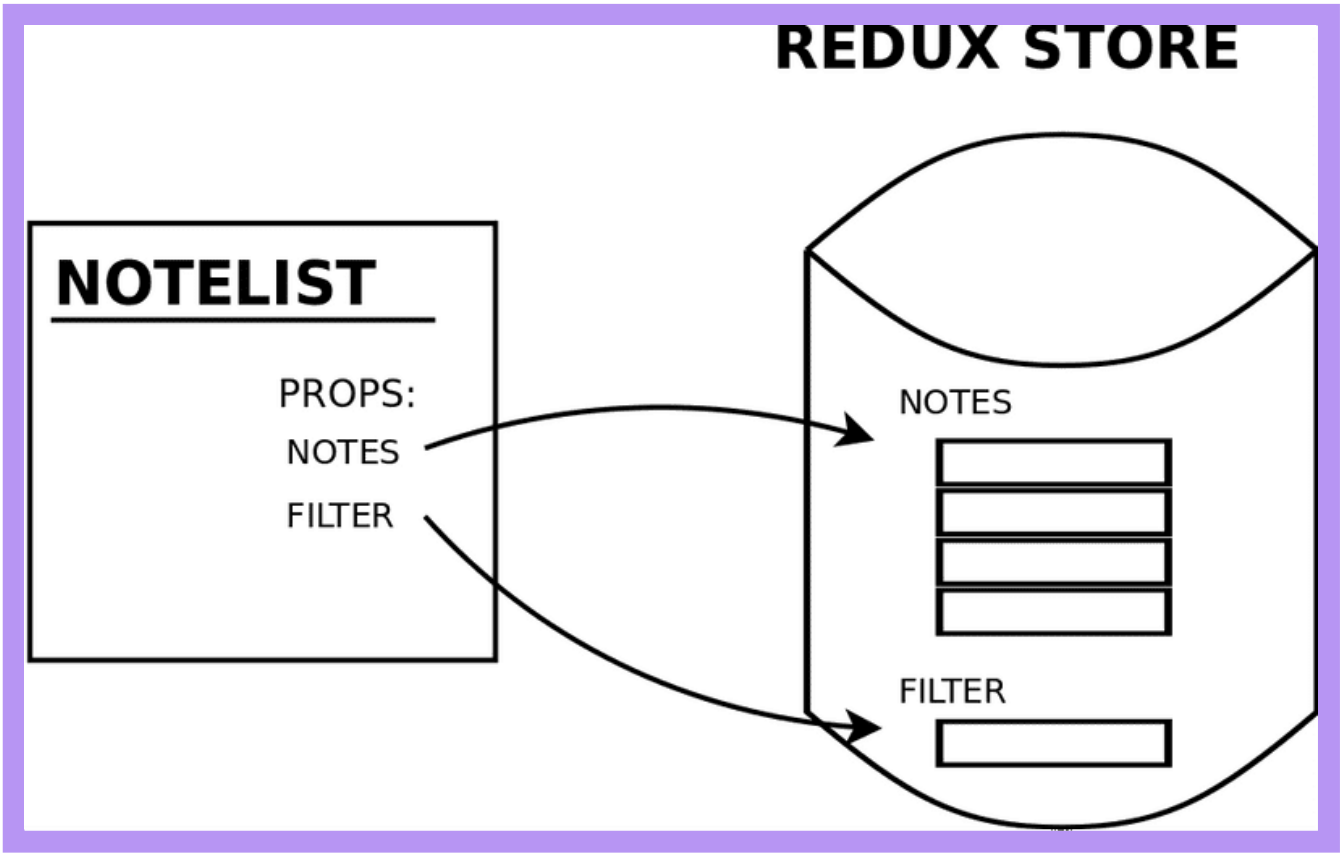
If we define:

copy

```
const Notes = (props) => {
  const dispatch = useDispatch()

  const notesToShow = () => {
    if ( props.filter === 'ALL' ) {
      return props.notes
    }

    return props.filter  === 'IMPORTANT'
      ? props.notes.filter(note => note.important)
      : props.notes.filter(note => !note.important)
  }

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

const mapStateToProps = (state) => {
  return {
    notes: state.notes,
    filter: state.filter,
  }
}

const ConnectedNotes = connect(mapStateToProps)(Notes)

export default ConnectedNotes
```

The *Notes* component can access the state of the store directly, e.g. through *props.notes* contains the list of notes. Similarly, *props.filter* references the value of the filter.

The situation that results from using *connect* with the *mapStateToProps* function we defined can be visualized like this:




The *Notes* component has "direct access" via *props.notes* and *props.filter* for inspecting the state of the Redux store.

The `NoteList` component does not need the information about which filter is selected, so we can move the filtering logic elsewhere. We just have to give it correctly filtered notes in the `notes` prop:

```
const Notes = (props) => {
  const dispatch = useDispatch()

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

```
    )
  }

  const mapStateToProps = (state) => {
    if ( state.filter === 'ALL' ) {
      return {
        notes: state.notes
      }
    }
    return {
      notes: (state.filter  === 'IMPORTANT'
        ? state.notes.filter(note => note.important)
        : state.notes.filter(note => !note.important)
      )
    }
  `

    {() => fs}
```

## mapDispatchToProps

Now we have gotten rid of `useSelector`, but *Notes* still uses the `useDispatch` hook and the `dispatch` function returning it:

```
const Notes = (props) => {
  const dispatch = useDispatch()

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

The second parameter of the `connect` function can be used for defining mapDispatchToProps which is a group of *action creator* functions passed to the connected component as props. Let's make the following changes to our existing connect operation:

```
const mapStateToProps = (state) => {
  return {
    notes: state.notes,
    filter: state.filter,
  }
}

const mapDispatchToProps = {
  toggleImportanceOf,
}

const ConnectedNotes = connect(
  mapStateToProps,
  mapDispatchToProps
)(Notes)

export default ConnectedNotes
```

Now the component can directly dispatch the action defined by the `toggleImportanceOf` action creator by calling the function through its props:

```
const Notes = (props) => {
  return(
    <ul>
      {props.notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() => props.toggleImportanceOf(note.id)}
        />
      )}
    </ul>
  )
}
```

This means that instead of dispatching the action like this:

```
dispatch(toggleImportanceOf(note.id))
```

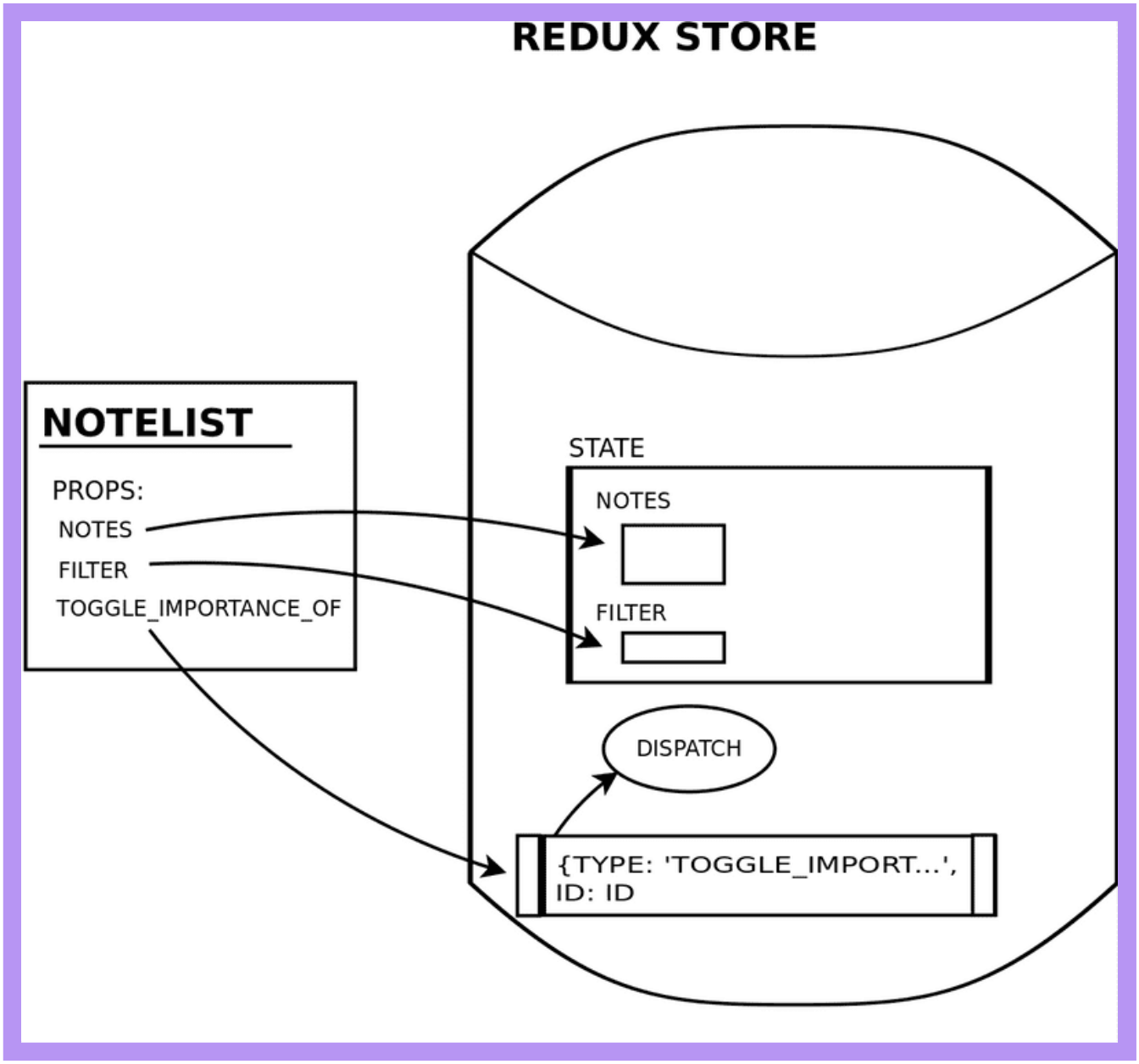
When using `connect` we can simply do this:

```
props.toggleImportanceOf(note.id)
```

There is no need to call the `dispatch` function separately since `connect` has already modified the `toggleImportanceOf` action creator into a form that contains the dispatch.

It can take some time to wrap your head around how `mapDispatchToProps` works, especially once we take a look at an alternative way of using it.

The resulting situation from using `connect` can be visualized like this:




In addition to accessing the store's state via *props.notes* and *props.filter*, the component also references a function that can be used for dispatching *notes/toggleImportanceOf*-type actions via its *toggleImportanceOf* prop.

The code for the newly refactored *Notes* component looks like this:

```
import { connect } from 'react-redux'
import { toggleImportanceOf } from '../reducers/noteReducer'
```

```
const Notes = (props) => {
  return(
    <ul>
      {props.notes.map(note =>
        <Note
          key={note.id}
          note={note}
          handleClick={() => props.toggleImportanceOf(note.id)}
        />
      )}
    </ul>
  )
}

const mapStateToProps = (state) => {
  if ( state.filter === 'ALL' ) {
    return {
      notes: state.notes
    }
  }

  return {
    notes: (state.filter  === 'IMPORTANT'
    ? state.notes.filter(note => note.important)
    : state.notes.filter(note => !note.important)
    )
  }
}

const mapDispatchToProps = {
  toggleImportanceOf
}

export default connect(
  mapStateToProps,
  mapDispatchToProps
)(Notes)
```

Let's also use `connect` to create new notes:

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

const NewNote = (props) => {

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

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

export default connect(
  null,
  { createNote }
)(NewNote)
```

Since the component does not need to access the store's state, we can simply pass *null* as the first parameter to `connect`.

You can find the code for our current application in its entirety in the *part6-5* branch of this GitHub repository.

## Referencing action creators passed as props

Let's direct our attention to one interesting detail in the *NewNote* component:

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

const NewNote = (props) => {

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

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

export default connect(
  null,
  { createNote }
)(NewNote)
```

Developers who are new to connect may find it puzzling that there are two versions of the *createNote* action creator in the component.

The function must be referenced as *props.createNote* through the component's props, as this is the version that *contains the automatic dispatch* added by `connect`.

Due to the way that the action creator is imported:

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

copy

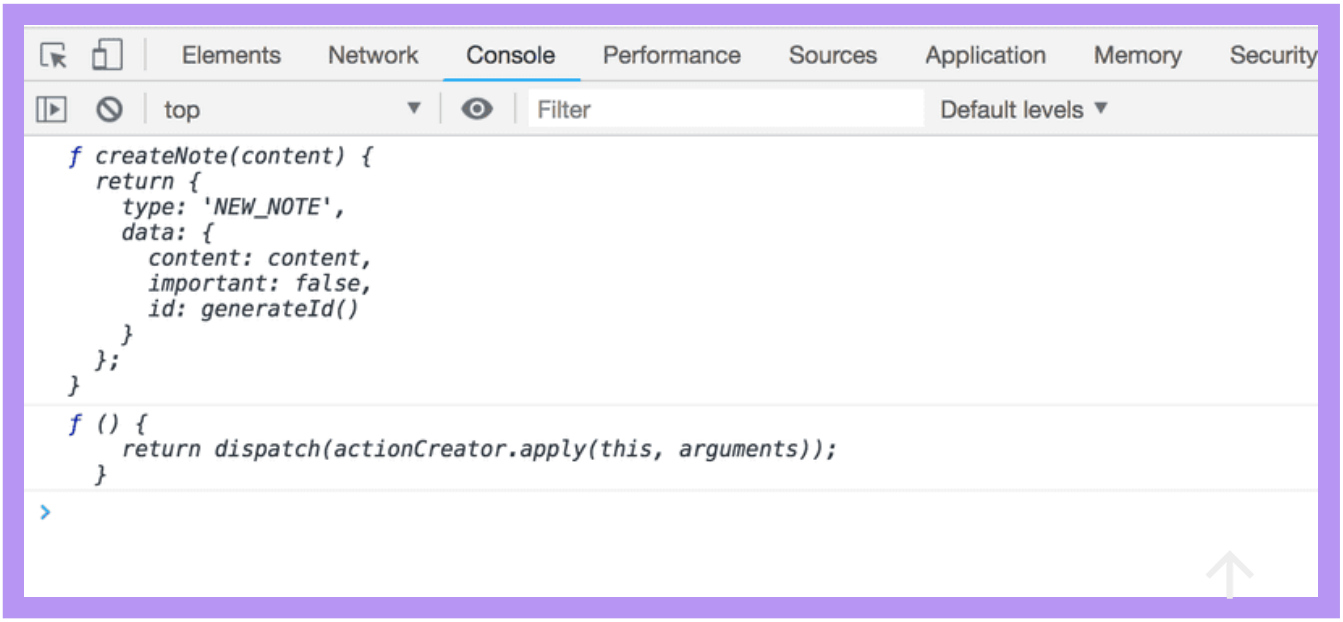
The action creator can also be referenced directly by calling `createNote`. You should not do this, since this is the unmodified version of the action creator that does not contain the added automatic dispatch.

If we print the functions to the console from the code (we have not yet looked at this useful debugging trick):

```
const NewNote = (props) => {
  console.log(createNote)
  console.log(props.createNote)

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

  // ...
}
```

copy

We can see the difference between the two functions:



The first function is a regular *action creator* whereas the second function contains the additional dispatch to the store that was added by connect.

Connect is an incredibly useful tool although it may seem difficult at first due to its level of abstraction.

## An alternative way of using mapDispatchToProps

We defined the function for dispatching actions from the connected *NewNote* component in the following way:

```
const NewNote = () => {
  // ...
}

export default connect(
  null,
  { createNote }
)(NewNote)
```

The connect expression above enables the component to dispatch actions for creating new notes with the `props.createNote('a new note')` command.

The functions passed in *mapDispatchToProps* must be *action creators*, that is, functions that return Redux actions.

It is worth noting that the *mapDispatchToProps* parameter is a *JavaScript object*, as the definition:

```
{
  createNote
}
```

Is just shorthand for defining the object literal:

```
{
  createNote: createNote
}
```

Which is an object that has a single *createNote* property with the *createNote* function as its value.

Alternatively, we could pass the following *function* definition as the second parameter to `connect`:

```
const NewNote = (props) => {
  // ...
```

```
}

const mapDispatchToProps = dispatch => {
  return {
    createNote: value => {
      dispatch(createNote(value))
    },
  }
}

export default connect(
  null,
  mapDispatchToProps
)(NewNote)
```

In this alternative definition, *mapDispatchToProps* is a function that `connect` will invoke by passing to it the `dispatch` function as its parameter. The return value of the function is an object that defines a group of functions that get passed to the connected component as props. Our example defines the function passed as the *createNote* prop:

```
value => {
  dispatch(createNote(value))
}
```

Which simply dispatches the action created with the *createNote* action creator.

The component then references the function through its props by calling *props.createNote*:

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

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

The concept is quite complex and describing it through text is challenging. In most cases, it is sufficient to use the simpler form of *mapDispatchToProps*. However, there are situations where a more complicated definition is necessary, like if the *dispatched actions* need to reference the props of the component.

The creator of Redux Dan Abramov has created a wonderful tutorial called Getting started with Redux that you can find on Egghead.io. I highly recommend the tutorial to everyone. The last four videos discuss the `connect` method, particularly the more "complicated" way of using it.

## Presentational/Container revisited

The refactored *Notes* component is almost entirely focused on rendering notes and is quite close to being a so-called presentational component. According to the description provided by Dan Abramov, presentational components:

- Are concerned with how things look.
- May contain both presentational and container components inside, and usually have some DOM markup and styles of their own.
- Often allow containment via props.children.
- Have no dependencies on the rest of the app, such as Redux actions or stores.
- Don't specify how the data is loaded or mutated.
- Receive data and callbacks exclusively via props.
- Rarely have their own state (when they do, it's UI state rather than data).
- Are written as functional components unless they need state, lifecycle hooks, or performance optimizations.

The `connected component` that is created with the `connect` function:

```
const mapStateToProps = (state) => {
  if ( state.filter === 'ALL' ) {
    return {
      notes: state.notes
    }
  }

  return {
    notes: (state.filter  === 'IMPORTANT'
    ? state.notes.filter(note => note.important)
    : state.notes.filter(note => !note.important)
    )
  }
}

const mapDispatchToProps = {
  toggleImportanceOf,
}

export default connect(
  mapStateToProps,
```

```
    mapDispatchToProps
)(Notes)
```

Fits the description of a *container* component. According to the description provided by Dan Abramov, container components:

- Are concerned with how things work.
- May contain both presentational and container components inside but usually don't have any DOM markup of their own except for some wrapping divs, and never have any styles.
- Provide the data and behavior to presentational or other container components.
- Call Redux actions and provide these as callbacks to the presentational components.
- Are often stateful, as they tend to serve as data sources.
- Are usually generated using higher-order components such as connect from React Redux, rather than written by hand.

Dividing the application into presentational and container components is one way of structuring React applications that has been deemed beneficial. The division may be a good design choice or it may not, it depends on the context.

Abramov attributes the following benefits to the division:

- Better separation of concerns. You understand your app and your UI better by writing components this way.
- Better reusability. You can use the same presentational component with completely different state sources, and turn those into separate container components that can be further reused.
- Presentational components are essentially your app's "palette". You can put them on a single page and let the designer tweak all their variations without touching the app's logic. You can run screenshot regression tests on that page.

Abramov mentions the term higher-order component. The *Notes* component is an example of a regular component, whereas the *connect* method provided by React-Redux is an example of a *high-order component*. Essentially, a higher-order component is a function that accepts a "regular" component as its parameter, which then returns a new "regular" component as its return value.

Higher-order components, or HOCs, are a way of defining generic functionality that can be applied to components. This is a concept from functional programming that very slightly resembles inheritance in object-oriented programming.

HOCs are a generalization of the Higher-Order Function (HOF) concept. HOFs are functions that either accept functions as parameters or return functions. We have been using HOFs throughout the course, e.g. all of the methods used for dealing with arrays like `map, filter and find` are HOFs.

After the React hook API was published, HOCs have become less and less popular. Almost all libraries which used to be based on HOCs have now been modified to use hooks. Most of the time hook-based APIs are a lot simpler than HOC-based ones, as is the case with Redux as well.

## Redux and the component state

We have come a long way in this course and, finally, we have come to the point at which we are using React "the right way", meaning React only focuses on generating the views, and the application state is wholly separated from the React components and passed on to Redux, its actions, and its reducers.

What about the `useState` hook, which provides components with their own state? Does it have any role if an application is using Redux or some other external state management solution? If the application has more complicated forms, it may be beneficial to implement their local state using the state provided by the `useState` function. One can, of course, have Redux manage the state of the forms, however, if the state of the form is only relevant when filling the form (e.g. for validation) it may be wise to leave the management of state to the component responsible for the form.

Should we always use Redux? Probably not. Dan Abramov, the developer of Redux, discusses this in his article You Might Not Need Redux.

Nowadays it is possible to implement Redux-like state management without Redux by using the React context api and the useReducer hook. More about this here and here. We will also practice this in part 9.

## Exercises 6.19.-6.21

NOTE: this is the old ending section of the part 6 that has 30th January 2023 been replaced with material about React Query, useReducer and context. This section remains here for a couple of weeks.

If you have started with the exercises that use the Redux connect you may continue with those. If you have not yet started, I recommend to proceed with the new section.

### 6.19 anecdotes and connect, step1

The *redux store* is currently being accessed by the components through the `useSelector` and `useDispatch` hooks.

Modify the *Notification* component so that it uses the `connect` function instead of the hooks.

### 6.20 anecdotes and connect, step2

Do the same for the *Filter* and *AnecdoteForm* components.

### 6.21 anecdotes, the grand finale

You (probably) have one nasty bug in your application. If the user clicks the vote button multiple times in a row, the notification is displayed funnily. For example, if a user votes twice in three seconds, the last notification is only displayed for two seconds (assuming the notification is normally shown for 5 seconds). This happens because removing the first notification accidentally removes the second notification.

Fix the bug so that after multiple votes in a row, the notification for the last vote is displayed for five seconds.

The fix can be done by canceling the previous notification when a new notification is displayed, whenever necessary. The documentation for the setTimeout function might also be useful for this.

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.

Propose changes to material

< Part 6d
Previous part

Part 7 >
Next part

About course

Course contents

FAQ

Partners

Challenge