a React-router

The exercises in this seventh part of the course differ a bit from the ones before. In this and the next chapter, as usual, there are exercises related to the theory in the chapter.

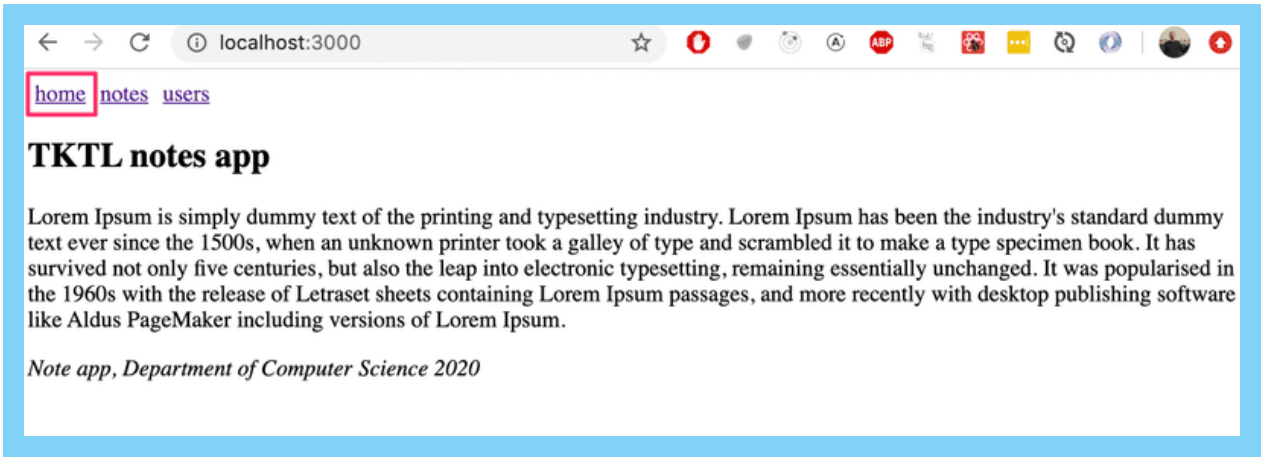
In addition to the exercises in this and the next chapter, there are a series of exercises which we'll be revising what we've learned during the whole course by expanding the Bloglist application, which we worked on during parts 4 and 5.

Application navigation structure

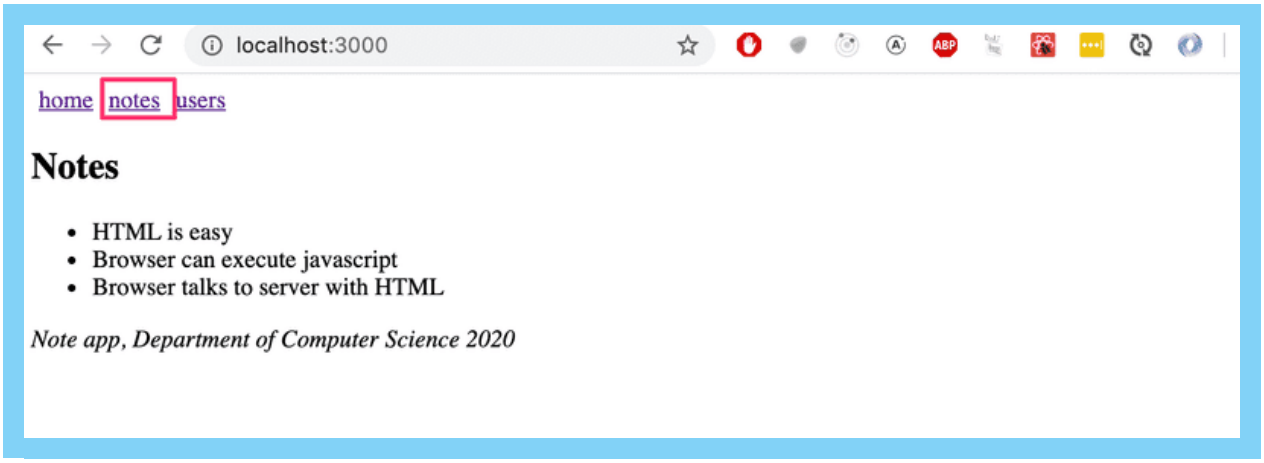
Following part 6, we return to React without Redux.

It is very common for web-applications to have a navigation bar, which enables switching the view of the application.

Our app could have a main page



and separate pages for showing information on notes and users:



In an old school web app, changing the page shown by the application would be accomplished by the browser making an HTTP GET request to the server and rendering the HTML representing the view that was returned.

In single page apps, we are, in reality, always on the same page. The Javascript code run by the browser creates an illusion of different "pages". If HTTP requests are made when switching view, they are only for fetching JSON formatted data, which the new view might require for it to be shown.

The navigation bar and an application containing multiple views is very easy to implement using React.

Here is one way:

```
import React, { useState } from 'react'
import ReactDOM from 'react-dom'

const Home = () => (
  <div> <h2>TKTL notes app</h2> </div>
)

const Notes = () => (
  <div> <h2>Notes</h2> </div>
)

const Users = () => (
  <div> <h2>Users</h2> </div>
)

const App = () => {
  const [page, setPage] = useState('home')

 const toPage = (page) => (event) => {
    event.preventDefault()
    setPage(page)
  }

  const content = () => {
    if (page === 'home') {
```

```
      return <Home />
    } else if (page === 'notes') {
      return <Notes />
    } else if (page === 'users') {
      return <Users />
    }
  }

  const padding = {
    padding: 5
  }

  return (
    <div>
      <div>
        <a href="" onClick={toPage('home')} style={padding}>
          home
        </a>
        <a href="" onClick={toPage('notes')} style={padding}>
          notes
        </a>
        <a href="" onClick={toPage('users')} style={padding}>
          users
        </a>
      </div>

      {content()}
    </div>
  )
}

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

Each view is implemented as its own component. We store the view component information in the application state called *page*. This information tells us which component, representing a view, should be shown below the menu bar.

However, the method is not very optimal. As we can see from the pictures, the address stays the same even though at times we are in different views. Each view should preferably have its own address, e.g. to make bookmarking possible. The *back*-button doesn't work as expected for our application either, meaning that *back* doesn't move you to the previously displayed view of the application, but somewhere completely different. If the application were to grow even bigger and we wanted to, for example, add separate views for each user and note, then this self made *routing*, which means the navigation management of the application, would get overly complicated.

Luckily, React has the React router -library, which provides an excellent solution for managing navigation in a React-application.

Let's change the above application to use React router. First, we install React router with the command

```
npm install react-router-dom
```

The routing provided by React Router is enabled by changing the application as follows:

```
import {
  BrowserRouter as Router,
  Switch, Route, Link
} from "react-router-dom"

const App = () => {

  const padding = {
    padding: 5
  }

  return (
    <Router>
      <div>
        <Link style={padding} to="/">home</Link>
        <Link style={padding} to="/notes">notes</Link>
        <Link style={padding} to="/users">users</Link>
      </div>

      <Switch>
        <Route path="/notes">
          <Notes />
        </Route>
        <Route path="/users">
          <Users />
        </Route>
        <Route path="/">
          <Home />
        </Route>
      </Switch>

      <div>
        <i>Note app, Department of Computer Science 2021</i>
      </div>
    </Router>
  )
}
```

Routing, or the conditional rendering of components *based on the url* in the browser, is used by placing components as children of the *Router* component, meaning inside *Router*-tags.

Notice that, even though the component is referred to by the name *Router*, we are in fact talking about BrowserRouter, because here the import happens by renaming the imported object:

```
import {
  BrowserRouter as Router,
  Switch, Route, Link
} from "react-router-dom"
```

According to the manual:

> *BrowserRouter* is a *Router* that uses the HTML5 history API (pushState, replaceState and the popState event) to keep your UI in sync with the URL.

Normally the browser loads a new page when the URL in the address bar changes. However, with the help of the HTML5 history API *BrowserRouter* enables us to use the URL in the address bar of the browser for internal "routing" in a React-application. So, even if the URL in the address bar changes, the content of the page is only manipulated using Javascript, and the browser will not load new content from the server. Using the back and forward actions, as well as making bookmarks, is still logical like on a traditional web page.

Inside the router we define *links* that modify the address bar with the help of the Link component. For example,

```
<Link to="/notes">notes</Link>
```

creates a link in the application with the text *notes*, which when clicked changes the URL in the address bar to */notes*.

Components rendered based on the URL of the browser are defined with the help of the component Route. For example,

```
<Route path="/notes">
  <Notes />
</Route>
```

defines, that if the browser address is */notes*, we render the *Notes* component.

We wrap the components to be rendered based on the url with a Switch-component

```
<Switch>
  <Route path="/notes">
    <Notes />
  </Route>
  <Route path="/users">
    <Users />
  </Route>
```

```
    <Route path="/">
      <Home />
    </Route>
  </Switch>
```

The switch works by rendering the first component whose *path* matches the url in the browser's address bar.

Note that the order of the components is important. If we would put the *Home*-component, whose path is *path="/"*, first, nothing else would ever get rendered because the "nonexistent" path "/" is the start of every path:

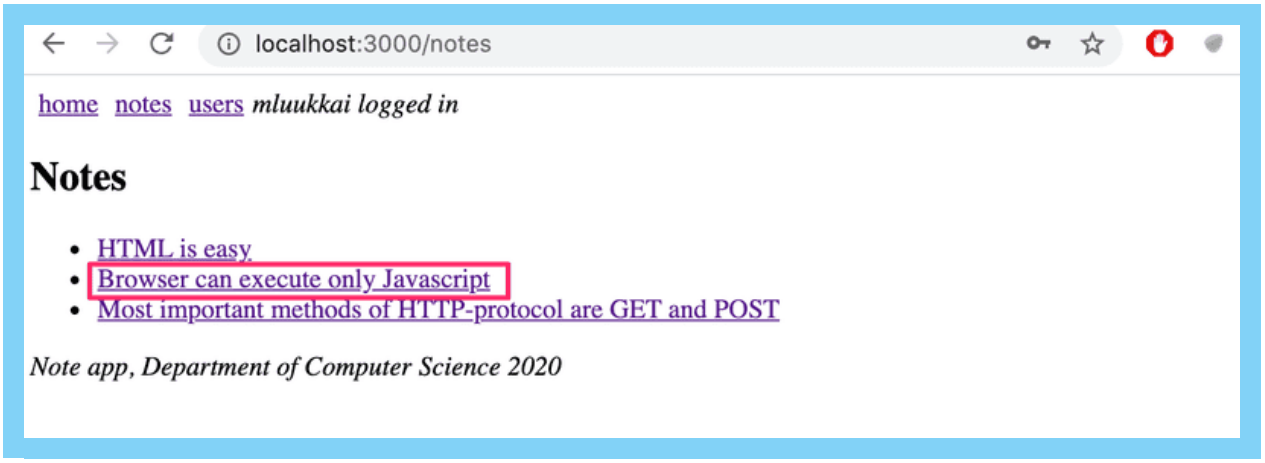
```
<Switch>
  <Route path="/">
    <Home />
  </Route>

  <Route path="/notes">
    <Notes />
  </Route>
  // ...
</Switch>
```

Parameterized route

Let's examine the slightly modified version from the previous example. The complete code for the example can be found here.

The application now contains five different views whose display is controlled by the router. In addition to the components from the previous example (*Home*, *Notes* and *Users*), we have *Login* representing the login view and *Note* representing the view of a single note.

Home and *Users* are unchanged from the previous exercise. *Notes* is a bit more complicated. It renders the list of notes passed to it as props in such a way that the name of each note is clickable.



The ability to click a name is implemented with the component *Link*, and clicking the name of a note whose id is 3 would trigger an event that changes the address of the browser into *notes/3*:

```
const Notes = ({notes}) => (
  <div>
    <h2>Notes</h2>
    <ul>
      {notes.map(note =>
        <li key={note.id}>
          <Link to={`/notes/${note.id}`}>{note.content}</Link>
        </li>
      )}
    </ul>
  </div>
)
```

We define parametrized urls in the routing in *App*-component as follows:

```
<Router>
  <div>
    <div>
      <Link style={padding} to="/">home</Link>
      <Link style={padding} to="/notes">notes</Link>
      <Link style={padding} to="/users">users</Link>
    </div>

    <Switch>
      <Route path="/notes/:id">
        <Note notes={notes} />
      </Route>
      <Route path="/notes">
        <Notes notes={notes} />
      </Route>
      <Route path="/">
        <Home />
      </Route>
```

```
    </Switch>
</Router>
```

We define the route rendering a specific note "express style" by marking the parameter with a colon *:id*

```
<Route path="/notes/:id">
```

When a browser navigates to the url for a specific note, for example */notes/3*, we render the *Note* component:

```
import {
  // ...
  useParams
} from "react-router-dom"

const Note = ({ notes }) => {
  const id = useParams().id
  const note = notes.find(n => n.id === Number(id))
  return (
    <div>
      <h2>{note.content}</h2>
      <div>{note.user}</div>
      <div><strong>{note.important ? 'important' : ''}</strong></div>
    </div>
  )
}
```

The `Note` component receives all of the notes as props *notes*, and it can access the url parameter (the id of the note to be displayed) with the `useParams` function of the react-router.

useHistory

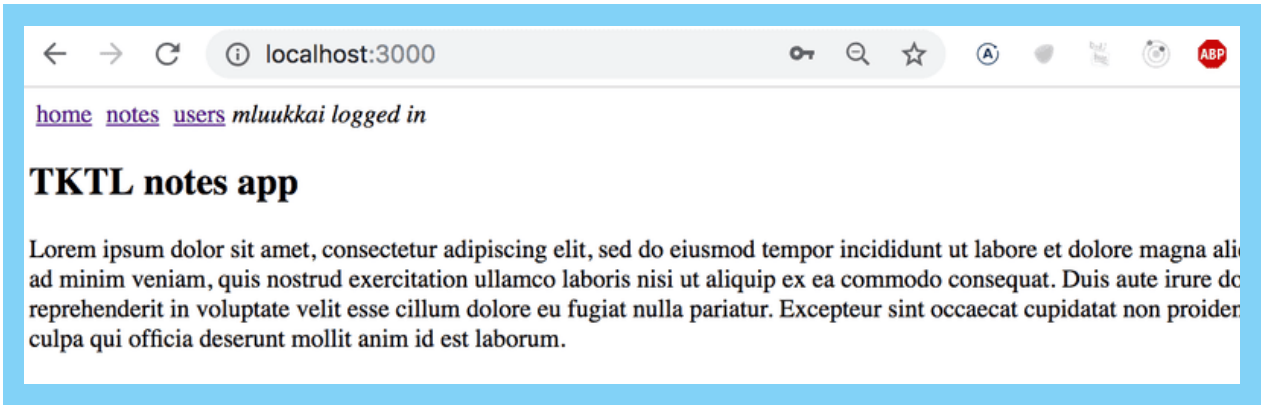
We have also implemented a simple log in function in our application. If a user is logged in, information about a logged in user is saved to the *user* field of the state of the *App* component.

The option to navigate to the *Login*-view is rendered conditionally in the menu.

```
<Router>
  <div>
    <Link style={padding} to="/">home</Link>
    <Link style={padding} to="/notes">notes</Link>
```

```
      <Link style={padding} to="/users">users</Link>
      {user
        ? <em>{user} logged in</em>
        : <Link style={padding} to="/login">login</Link>
      }
    </div>

    // ...
  </Router>
```

So if the user is already logged in, instead of displaying the link *Login*, we show the username of the user:



The code of the component handling the login functionality is as follows:

```
import {
  // ...
  useHistory
} from 'react-router-dom'

const Login = (props) => {
  const history = useHistory()

  const onSubmit = (event) => {
    event.preventDefault()
    props.onLogin('mluukkai')
    history.push('/')
  }

  return (
    <div>
      <h2>login</h2>
      <form onSubmit={onSubmit}>
        <div>
          username: <input />
        </div>
        <div>
          password: <input type='password' />
        </div>
```

```
        <button type="submit">login</button>
      </form>
    </div>
  )
}
```

What is interesting about this component is the use of the useHistory function of the react-router. With this function, the component can access a history object. The history object can be used to modify the browser's url programmatically.

With user log in, we call the push method of the history object. The `history.push('/')` call causes the browser's url to change to `/` and the application renders the corresponding component *Home*.

Both useParams and useHistory are hook-functions, just like useState and useEffect which we have used many times now. As you remember from part 1, there are some rules to using hook-functions. Create-react-app has been configured to warn you if you break these rules, for example, by calling a hook-function from a conditional statement.

redirect

There is one more interesting detail about the *Users* route:

```
<Route path="/users">
  {user ? <Users /> : <Redirect to="/login" />}
</Route>
```

If a user isn't logged in, the *Users* component is not rendered. Instead, the user is *redirected* using the *Redirect*-component to the login view

```
<Redirect to="/login" />
```

In reality, it would perhaps be better to not even show links in the navigation bar requiring login if the user is not logged into the application.

Here is the *App* component in its entirety:

```
const App = () => {
  const [notes, setNotes] = useState([
    // ...
  ])
```

```
  const [user, setUser] = useState(null)

  const login = (user) => {
    setUser(user)
  }

  const padding = { padding: 5 }

  return (
    <div>
      <Router>
        <div>
          <Link style={padding} to="/">home</Link>
          <Link style={padding} to="/notes">notes</Link>
          <Link style={padding} to="/users">users</Link>
          {user
            ? <em>{user} logged in</em>
            : <Link style={padding} to="/login">login</Link>
          }
        </div>

        <Switch>
          <Route path="/notes/:id">
            <Note notes={notes} />
          </Route>
          <Route path="/notes">
            <Notes notes={notes} />
          </Route>
          <Route path="/users">
            {user ? <Users /> : <Redirect to="/login" />}
          </Route>
          <Route path="/login">
            <Login onLogin={login} />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </Router>
      <div>
        <br />
        <em>Note app, Department of Computer Science 2021</em>
      </div>
    </div>
  )
}
```

We define an element common for modern web apps called *footer*, which defines the part at the bottom of the screen, outside of the *Router*, so that it is shown regardless of the component shown in the routed part of the application.

Parameterized route revisited

Our application has a flaw. The `Note` component receives all of the notes, even though it only displays the one whose id matches the url parameter:

```
const Note = ({ notes }) => {
  const id = useParams().id
  const note = notes.find(n => n.id === Number(id))
  // ...
}
```

Would it be possible to modify the application so that `Note` receives only the component it should display?

```
const Note = ({ note }) => {
  return (
    <div>
      <h2>{note.content}</h2>
      <div>{note.user}</div>
      <div><strong>{note.important ? 'important' : ''}</strong></div>
    </div>
  )
}
```

One way to do this would be to use react-router's useRouteMatch hook to figure out the id of the note to be displayed in the `App` component.

It is not possible to use *useRouteMatch*-hook in the component which defines the routed part of the application. Let's move the use of the `Router` components from `App` :

```
ReactDOM.render(
  <Router>
    <App />
  </Router>,
  document.getElementById('root')
)
```

The `App` component becomes:

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

```
  useRouteMatch
} from "react-router-dom"

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

  const match = useRouteMatch('/notes/:id')
  const note = match
    ? notes.find(note => note.id === Number(match.params.id))
    : null

  return (
    <div>
      <div>
        <Link style={padding} to="/">home</Link>
        // ...
      </div>

      <Switch>
        <Route path="/notes/:id">
          <Note note={note} />
        </Route>
        <Route path="/notes">
          <Notes notes={notes} />
        </Route>
         // ...
      </Switch>

      <div>
        <em>Note app, Department of Computer Science 2021</em>
      </div>
    </div>
  )
}
```

Every time the component is rendered, so practically every time the browser's url changes, the following command is executed:

```
const match = useRouteMatch('/notes/:id')
```

If the url matches `/notes/:id`, the match variable will contain an object from which we can access the parametrized part of the path, the id of the note to be displayed, and we can then fetch the correct note to display.

```
const note = match
  ? notes.find(note => note.id === Number(match.params.id))
  : null
```

The completed code can be found here.

Exercises 7.1.-7.3.

Let's return to working with anecdotes. Use the redux-free anecdote app found in the repository https://github.com/fullstack-hy/routed-anecdotes as the starting point for the exercises.

If you clone the project into an existing git repository remember to *delete the git configuration of the cloned application:*

```
cd routed-anecdotes   // go first to directory of the cloned repository
rm -rf .git
```

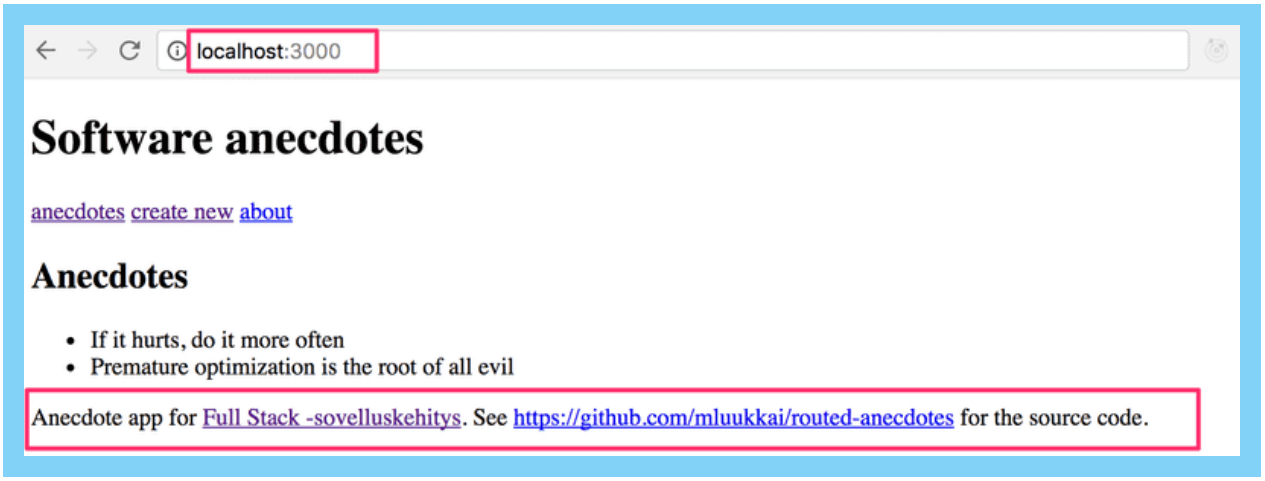
The application starts the usual way, but first you need to install the dependencies of the application:

```
npm install
npm start
```

7.1: routed anecdotes, step1

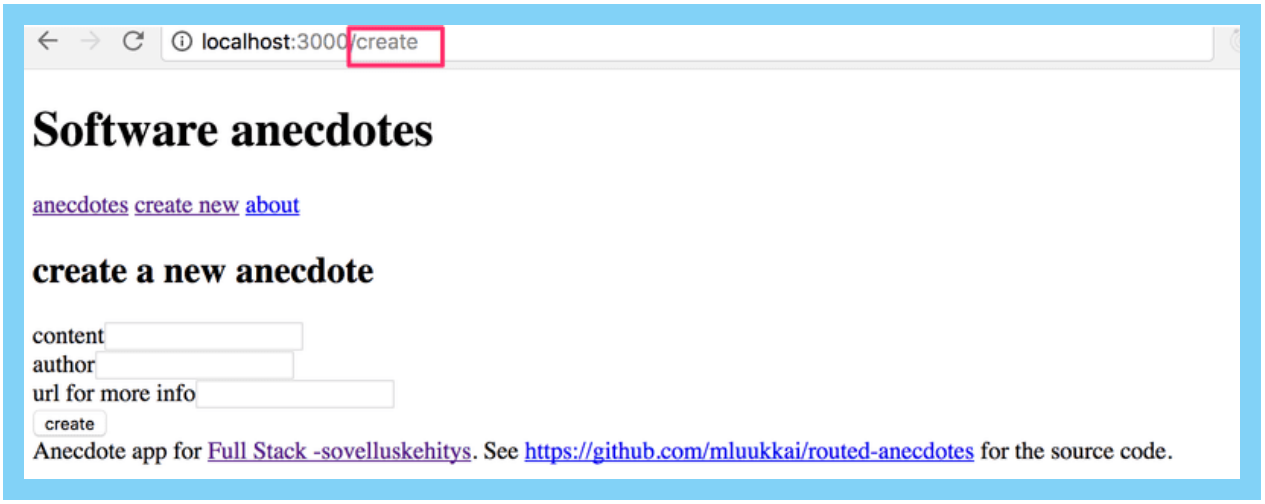
Add React Router to the application so that by clicking links in the *Menu*-component the view can be changed.

At the root of the application, meaning the path `/`, show the list of anecdotes:



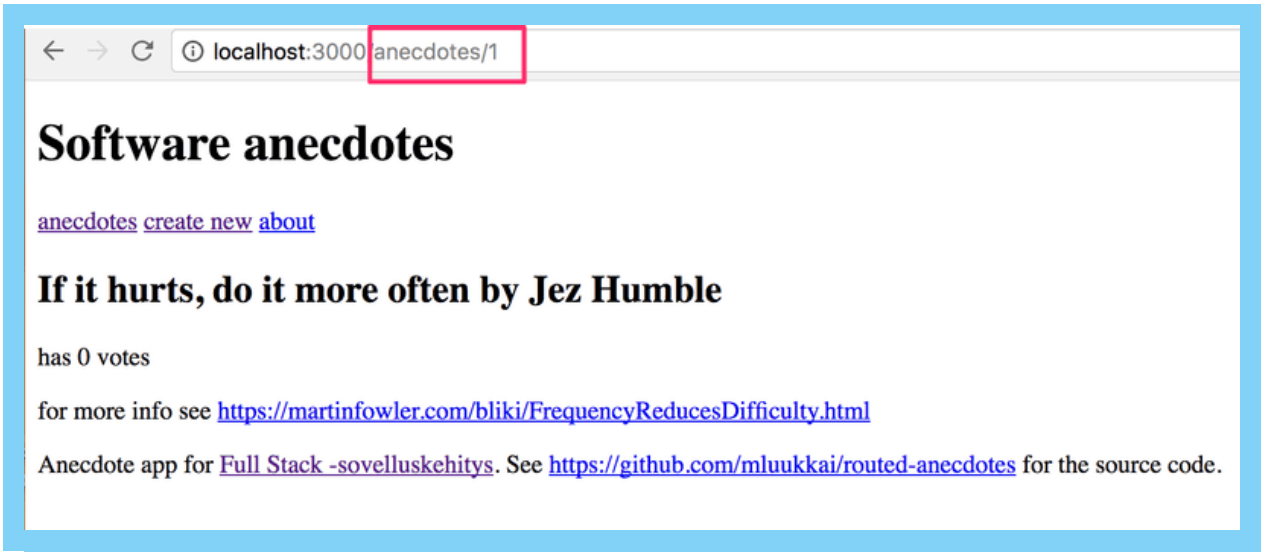
The *Footer*-component should always be visible at the bottom.

The creation of a new anecdote should happen e.g. in the path *create*:



7.2: routed anecdotes, step2

Implement a view for showing a single anecdote:



Navigating to the page showing the single anecdote is done by clicking the name of that anecdote

About course

Course contents

FAQ

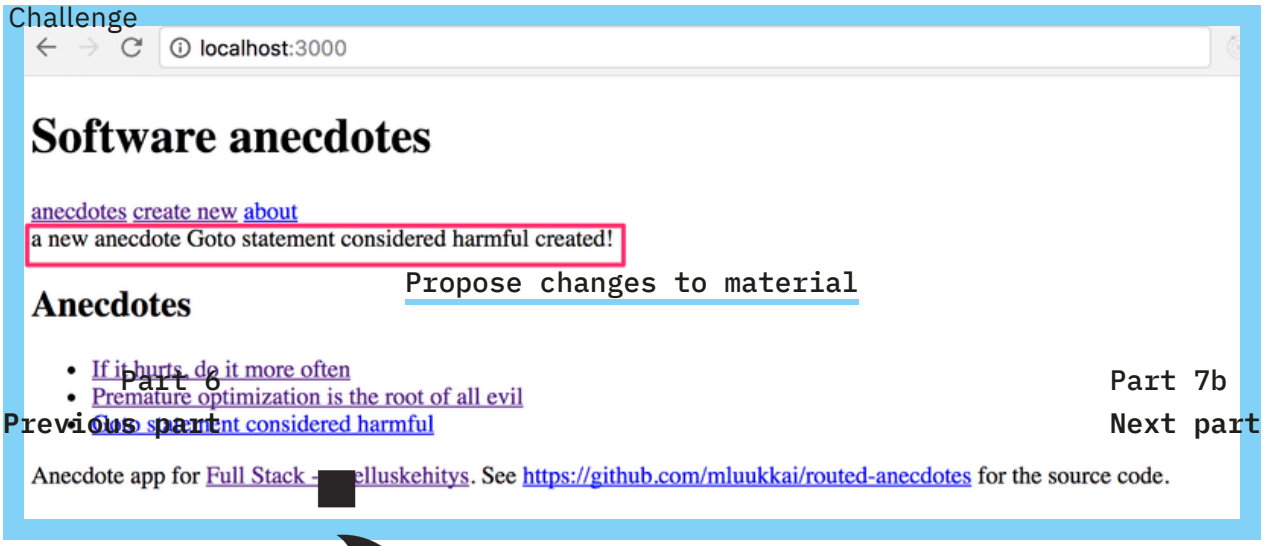
Partners

Challenge

7.3: routed anecdotes, step3

The default functionality of the creation form is quite confusing, because nothing seems to be happening after creating a new anecdote using the form.

Improve the functionality such that after creating a new anecdote the application transitions automatically to showing the view for all anecdotes *and* the user is shown a notification informing them of this successful creation for the next 10 seconds:



Propose changes to material

Previous part Part 6 Part 7b Next part

UNIVERSITY OF HELSINKI


HOUSTON