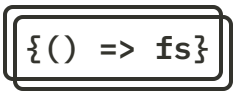


a Login in frontend

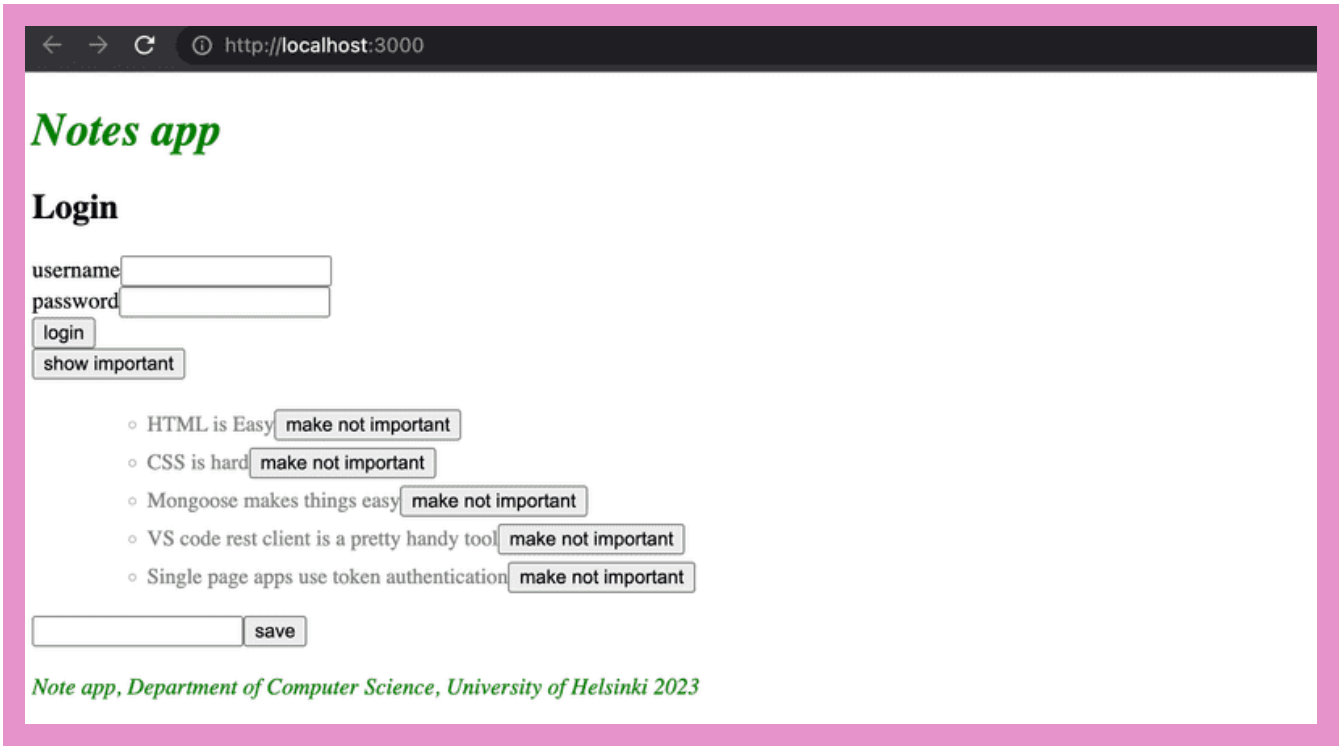
In the last two parts, we have mainly concentrated on the backend. The frontend that we developed in part 2 does not yet support the user management we implemented to the backend in part 4.

At the moment the frontend shows existing notes and lets users change the state of a note from important to not important and vice versa. New notes cannot be added anymore because of the changes made to the backend in part 4: the backend now expects that a token verifying a user's identity is sent with the new note.

We'll now implement a part of the required user management functionality in the frontend. Let's begin with the user login. Throughout this part, we will assume that new users will not be added from the frontend.

Handling login

A login form has now been added to the top of the page:



The code of the *App* component now looks as follows:

```
const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)
  const [errorMessage, setErrorMessage] = useState(null)
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')

  useEffect(() => {
    noteService
      .getAll().then(initialNotes => {
        setNotes(initialNotes)
      })
  }, [])

  // ...

  const handleLogin = (event) => {
    event.preventDefault()
    console.log('logging in with', username, password)
  }

  return (
    <div>
      <h1>Notes</h1>

      <Notification message={errorMessage} />

      <form onSubmit={handleLogin}>
```

```
          <div>
            username
              <input
              type="text"
              value={username}
              name="Username"
              onChange={({ target }) => setUsername(target.value)}
            />
          </div>
          <div>
            password
              <input
              type="password"
              value={password}
              name="Password"
              onChange={({ target }) => setPassword(target.value)}
            />
          </div>
          <button type="submit">login</button>
        </form>

        // ...
      </div>
    )
}

export default App
```

The current application code can be found on GitHub, in the branch *part5-1*. If you clone the repo, don't forget to run `npm install` before attempting to run the frontend.

The frontend will not display any notes if it's not connected to the backend. You can start the backend with `npm run dev` in its folder from Part 4. This will run the backend on port 3001. While that is active, in a separate terminal window you can start the frontend with `npm start`, and now you can see the notes that are saved in your MongoDB database from Part 4.

Keep this in mind from now on.

The login form is handled the same way we handled forms in part 2. The app state has fields for *username* and *password* to store the data from the form. The form fields have event handlers, which synchronize changes in the field to the state of the *App* component. The event handlers are simple: An object is given to them as a parameter, and they destructure the field *target* from the object and save its value to the state.

```
({ target }) => setUsername(target.value)
```

copy

The method `handleLogin`, which is responsible for handling the data in the form, is yet to be implemented.

Logging in is done by sending an HTTP POST request to the server address *api/login*. Let's separate the code responsible for this request into its own module, to file *services/login.js*.

We'll use *async/await* syntax instead of promises for the HTTP request:

```
import axios from 'axios'
const baseUrl = '/api/login'

const login = async credentials => {
  const response = await axios.post(baseUrl, credentials)
  return response.data
}

export default { login }
```

The method for handling the login can be implemented as follows:

```
import loginService from './services/login'

const App = () => {
  // ...
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')
  const [user, setUser] = useState(null)

  const handleLogin = async (event) => {
    event.preventDefault()

    try {
      const user = await loginService.login({
        username, password,
      })
      setUser(user)
      setUsername('')
      setPassword('')
    } catch (exception) {
      setErrorMessage('Wrong credentials')
      setTimeout(() => {
        setErrorMessage(null)
      }, 5000)
    }
  }

  // ...
}
```

If the login is successful, the form fields are emptied *and* the server response (including a *token* and the user details) is saved to the *user* field of the application's state.

If the login fails or running the function `loginService.login` results in an error, the user is notified.

The user is not notified about a successful login in any way. Let's modify the application to show the login form only *if the user is not logged-in*, so when `user === null` . The form for adding new notes is shown only if the *user is logged-in*, so when *user* contains the user's details.

Let's add two helper functions to the *App* component for generating the forms:

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

  const loginForm = () => (
    <form onSubmit={handleLogin}>
      <div>
        username
          <input
          type="text"
          value={username}
          name="Username"
          onChange={({ target }) => setUsername(target.value)}
        />
      </div>
      <div>
        password
          <input
          type="password"
          value={password}
          name="Password"
          onChange={({ target }) => setPassword(target.value)}
        />
      </div>
      <button type="submit">login</button>
    </form>
  )

  const noteForm = () => (
    <form onSubmit={addNote}>
      <input
        value={newNote}
        onChange={handleNoteChange}
      />
      <button type="submit">save</button>
    </form>
  )

  return (
    // ...
  )
}
```

and conditionally render them:

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

  const loginForm = () => (
    // ...
  )

  const noteForm = () => (
    // ...
  )

  return (
    <div>
      <h1>Notes</h1>

      <Notification message={errorMessage} />

      {user === null && loginForm()}
      {user !== null && noteForm()}

      <div>
        <button onClick={() => setShowAll(!showAll)}>
          show {showAll ? 'important' : 'all'}
        </button>
      </div>
      <ul>
        {notesToShow.map((note, i) =>
          <Note
            key={i}
            note={note}
            toggleImportance={() => toggleImportanceOf(note.id)}
          />
        )}
      </ul>

      <Footer />
    </div>
  )
}
```

A slightly odd looking, but commonly used React trick is used to render the forms conditionally:

```
{
  user === null && loginForm()
}
```

If the first statement evaluates to false or is falsy, the second statement (generating the form) is not executed at all.

We can make this even more straightforward by using the conditional operator:

```
return (
  <div>
    <h1>Notes</h1>

    <Notification message={errorMessage}/>

    {user === null ?
      loginForm() :
      noteForm()
    }

    <h2>Notes</h2>

    // ...

  </div>
)
```

If `user === null` is truthy, `loginForm()` is executed. If not, `noteForm()` is.

Let's do one more modification. If the user is logged in, their name is shown on the screen:

```
return (
  <div>
    <h1>Notes</h1>

    <Notification message={errorMessage} />

    {user === null ?
      loginForm() :
      <div>
        <p>{user.name} logged-in</p>
        {noteForm()}
      </div>
    }

    <h2>Notes</h2>

    // ...

  </div>
)
```

The solution isn't perfect, but we'll leave it like this for now.

Our main component *App* is at the moment way too large. The changes we did now are a clear sign that the forms should be refactored into their own components. However, we will leave that for an optional exercise.

The current application code can be found on GitHub, in the branch *part5-2*.

Creating new notes

The token returned with a successful login is saved to the application's state - the *user*'s field *token*:

```
const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login({
      username, password,
    })

    setUser(user)
    setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}
```

Let's fix creating new notes so it works with the backend. This means adding the token of the logged-in user to the Authorization header of the HTTP request.

The *noteService* module changes like so:

```
import axios from 'axios'
const baseUrl = '/api/notes'

let token = null

const setToken = newToken => {
  token = `Bearer ${newToken}`
}

const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

const create = async newObject => {
  const config = {
    headers: { Authorization: token },
  }
```

```
  const response = await axios.post(baseUrl, newObject, config)
  return response.data
}

const update = (id, newObject) => {
  const request = axios.put(`${ baseUrl }/${id}`, newObject)
  return request.then(response => response.data)
}

export default { getAll, create, update, setToken }
```

The noteService module contains a private variable called `token`. Its value can be changed with the `setToken` function, which is exported by the module. `create`, now with async/await syntax, sets the token to the *Authorization* header. The header is given to axios as the third parameter of the *post* method.

The event handler responsible for login must be changed to call the method `noteService.setToken(user.token)` with a successful login:

```
const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login({
      username, password,
    })

    noteService.setToken(user.token)
    setUser(user)
    setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}
```

And now adding new notes works again!

Saving the token to the browser's local storage

Our application has a small flaw: if the browser is refreshed (eg. pressing F5), the user's login information disappears.

This problem is easily solved by saving the login details to local storage. Local Storage is a key-value database in the browser.

It is very easy to use. A *value* corresponding to a certain *key* is saved to the database with the method setItem. For example:

```
window.localStorage.setItem('name', 'juha tauriainen')
```

copy

saves the string given as the second parameter as the value of the key *name*.

The value of a key can be found with the method getItem:

```
window.localStorage.getItem('name')
```

copy

while removeItem removes a key.

Values in the local storage are persisted even when the page is re-rendered. The storage is origin-specific so each web application has its own storage.

Let's extend our application so that it saves the details of a logged-in user to the local storage.

Values saved to the storage are DOMstrings, so we cannot save a JavaScript object as it is. The object has to be parsed to JSON first, with the method `JSON.stringify`. Correspondingly, when a JSON object is read from the local storage, it has to be parsed back to JavaScript with `JSON.parse`.

Changes to the login method are as follows:

```
const handleLogin = async (event) => {
  event.preventDefault()
  try {
    const user = await loginService.login({
      username, password,
    })

    window.localStorage.setItem(
      'loggedNoteappUser', JSON.stringify(user)
    )
    noteService.setToken(user.token)
    setUser(user)
    setUsername('')
    setPassword('')
  } catch (exception) {
    // ...
  }
}
```

copy

The details of a logged-in user are now saved to the local storage, and they can be viewed on the console (by typing `window.localStorage` in it):

You can also inspect the local storage using the developer tools. On Chrome, go to the *Application* tab and select *Local Storage* (more details here). On Firefox go to the *Storage* tab and select *Local Storage* (details here).

We still have to modify our application so that when we enter the page, the application checks if user details of a logged-in user can already be found on the local storage. If they are there, the details are saved to the state of the application and to *noteService*.

The right way to do this is with an effect hook: a mechanism we first encountered in part 2, and used to fetch notes from the server.

We can have multiple effect hooks, so let's create a second one to handle the first loading of the page:

```
const App = () => {
  const [notes, setNotes] = useState([])
  const [newNote, setNewNote] = useState('')
  const [showAll, setShowAll] = useState(true)
  const [errorMessage, setErrorMessage] = useState(null)
  const [username, setUsername] = useState('')
  const [password, setPassword] = useState('')
  const [user, setUser] = useState(null)

  useEffect(() => {
    noteService
      .getAll().then(initialNotes => {
        setNotes(initialNotes)
      })
  }, [])

  useEffect(() => {
    const loggedUserJSON = window.localStorage.getItem('loggedNoteappUser')
    if (loggedUserJSON) {
```

```
      const user = JSON.parse(loggedUserJSON)
      setUser(user)
      noteService.setToken(user.token)
    }
  }, [])

  // ...
}
```

The empty array as the parameter of the effect ensures that the effect is executed only when the component is rendered for the first time.

Now a user stays logged in to the application forever. We should probably add a *logout* functionality, which removes the login details from the local storage. We will however leave it as an exercise.

It's possible to log out a user using the console, and that is enough for now. You can log out with the command:

```
window.localStorage.removeItem('loggedNoteappUser')
```

copy

or with the command which empties *localstorage* completely:

```
window.localStorage.clear()
```

copy

The current application code can be found on GitHub, in the branch *part5-3*.

Exercises 5.1.-5.4.

We will now create a frontend for the blog list backend we created in the last part. You can use this application from GitHub as the base of your solution. You need to connect your backend with a proxy as shown in part 3.

It is enough to submit your finished solution. You can commit after each exercise, but that is not necessary.

The first few exercises revise everything we have learned about React so far. They can be challenging, especially if your backend is incomplete. It might be best to use the backend that we marked as the answer for part 4.

While doing the exercises, remember all of the debugging methods we have talked about, especially keeping an eye on the console.

Warning: If you notice you are mixing in the `async/await` and `then` commands, it's 99.9% certain you are doing something wrong. Use either or, never both.

5.1: Blog List Frontend, step 1

Clone the application from GitHub with the command:

```
git clone https://github.com/fullstack-hy2020/bloglist-frontend
```

copy

Remove the git configuration of the cloned application

```
cd bloglist-frontend   // go to cloned repository
rm -rf .git
```

copy

The application is started the usual way, but you have to install its dependencies first:

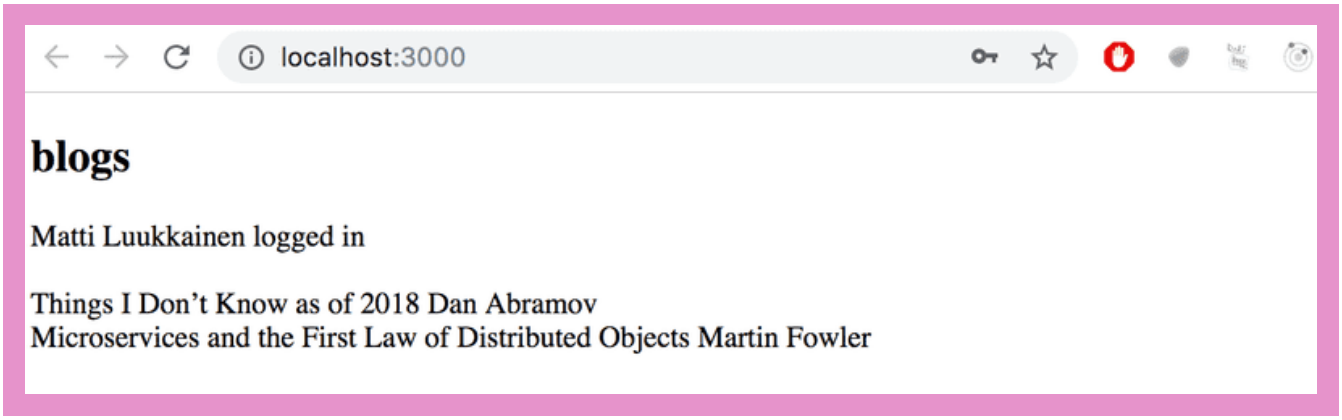
```
npm install
npm run dev
```

copy

Implement login functionality to the frontend. The token returned with a successful login is saved to the application's state *user*.

If a user is not logged in, *only* the login form is visible.

If the user is logged-in, the name of the user and a list of blogs is shown.



User details of the logged-in user do not have to be saved to the local storage yet.

NB You can implement the conditional rendering of the login form like this for example:

```
if (user === null) {
  return (
    <div>
      <h2>Log in to application</h2>
      <form>
        //...
      </form>
    </div>
  )
}

return (
  <div>
    <h2>blogs</h2>
    {blogs.map(blog =>
      <Blog key={blog.id} blog={blog} />
    )}
  </div>
)
}
```

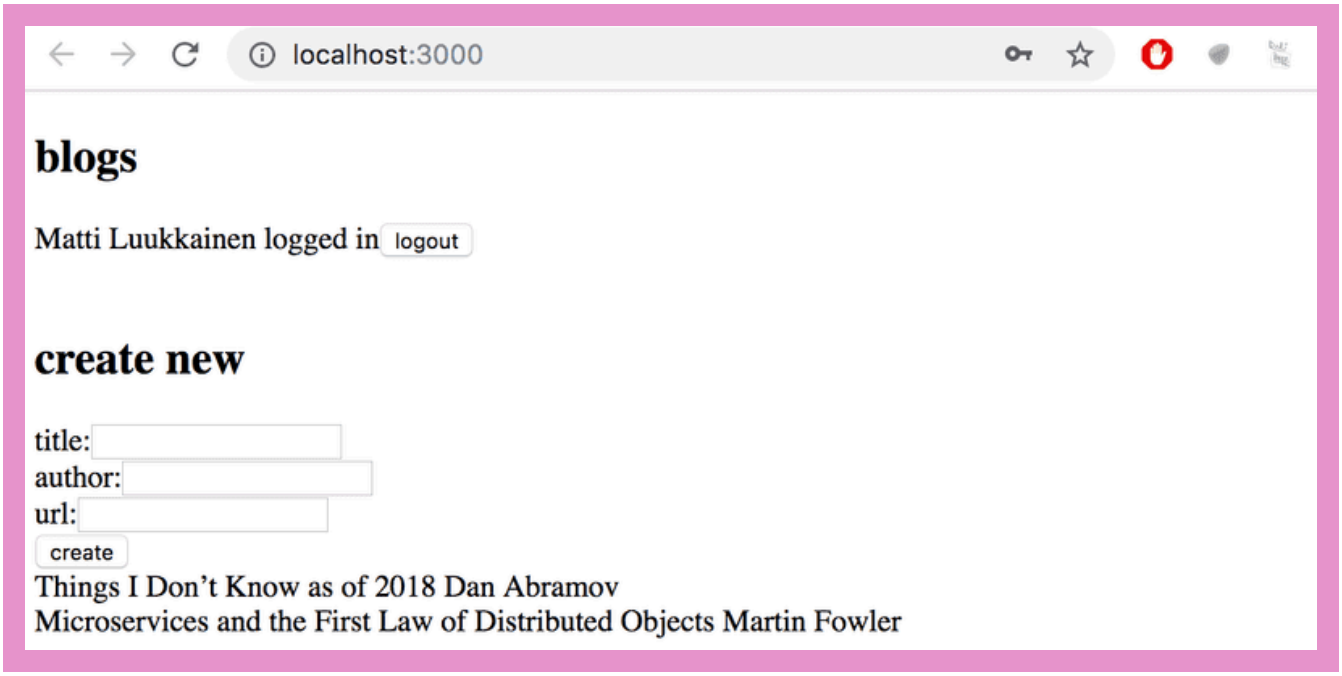
5.2: Blog List Frontend, step 2

Make the login 'permanent' by using the local storage. Also, implement a way to log out.

Ensure the browser does not remember the details of the user after logging out.

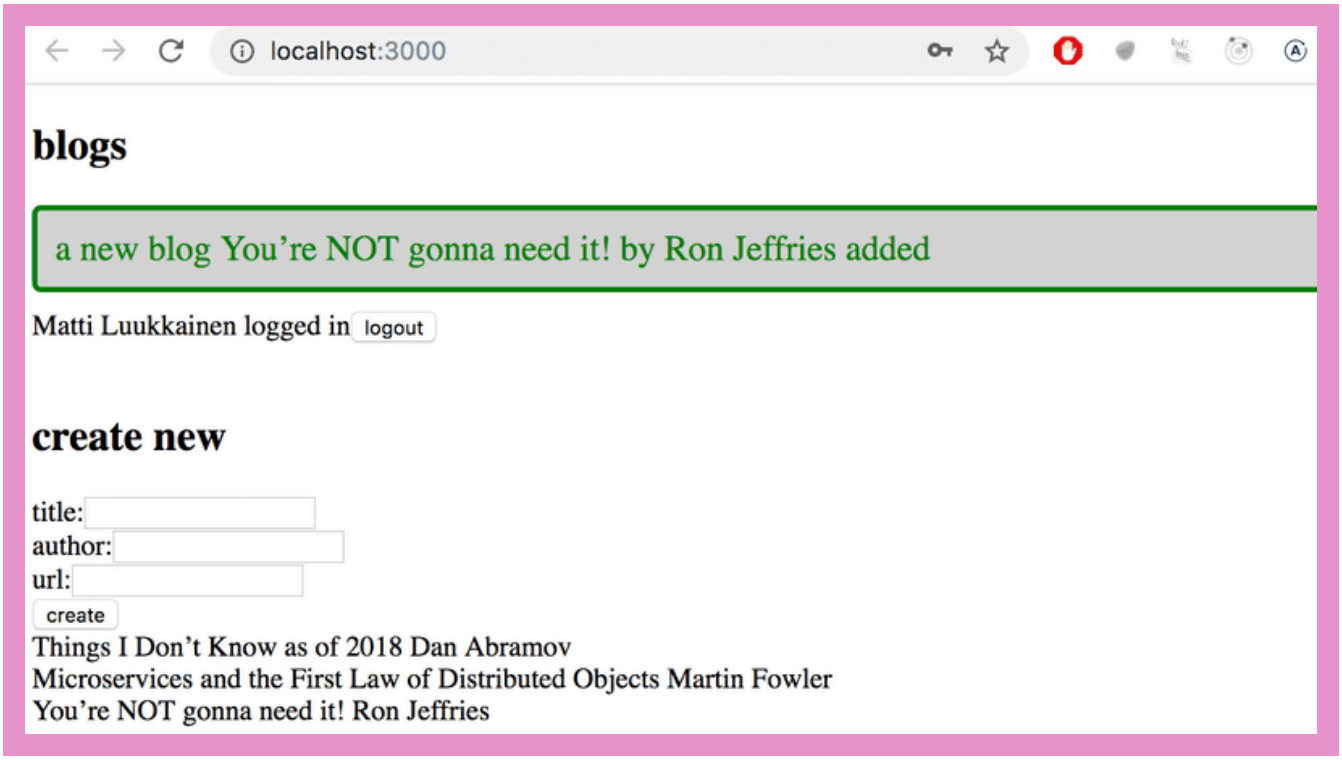
5.3: Blog List Frontend, step 3

Expand your application to allow a logged-in user to add new blogs:

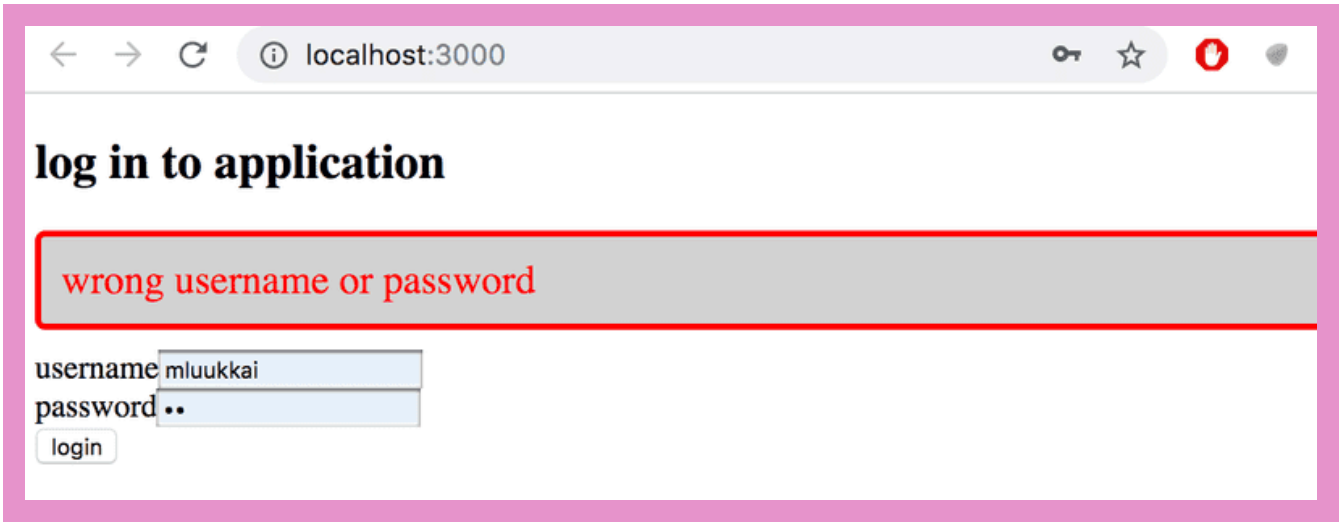


5.4: Blog List Frontend, step 4

Implement notifications that inform the user about successful and unsuccessful operations at the top of the page. For example, when a new blog is added, the following notification can be shown:



Failed login can show the following notification:



The notifications must be visible for a few seconds. It is not compulsory to add colors.

A note on using local storage

At the end of the last part, we mentioned that the challenge of token-based authentication is how to cope with the situation when the API access of the token holder to the API needs to be revoked.

There are two solutions to the problem. The first one is to limit the validity period of a token. This forces the user to re-login to the app once the token has expired. The other approach is to save the validity information of each token to the backend database. This solution is often called a *server-side session*.

No matter how the validity of tokens is checked and ensured, saving a token in the local storage might contain a security risk if the application has a security vulnerability that allows Cross Site Scripting (XSS) attacks. An XSS attack is possible if the application would allow a user to inject arbitrary JavaScript code (e.g. using a form) that the app would then execute. When using React sensibly it should not be possible since React sanitizes all text that it renders, meaning that it is not executing the rendered content as JavaScript.

If one wants to play safe, the best option is to not store a token in local storage. This might be an option in situations where leaking a token might have tragic consequences.

It has been suggested that the identity of a signed-in user should be saved as httpOnly cookies, so that JavaScript code could not have any access to the token. The drawback of this solution is that it would make implementing SPA applications a bit more complex. One would need at least to implement a separate page for logging in.

However, it is good to notice that even the use of httpOnly cookies does not guarantee anything. It has even been suggested that httpOnly cookies are not any safer than the use of local storage.

So no matter the used solution the most important thing is to minimize the risk of XSS attacks altogether.

Propose changes to material

Part 4
Previous part

Part 5b
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 5 > props.children and proptypes

b props.children and proptypes

Displaying the login form only when appropriate

Let's modify the application so that the login form is not displayed by default:

The login form appears when the user presses the *login* button:

The user can close the login form by clicking the *cancel* button.

Let's start by extracting the login form into its own component:

```
const LoginForm = ({
   handleSubmit,
   handleUsernameChange,
   handlePasswordChange,
   username,
   password
  }) => {
  return (
    <div>
      <h2>Login</h2>

      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            value={username}
            onChange={handleUsernameChange}
          />
        </div>
        <div>
          password
          <input
            type="password"
            value={password}
            onChange={handlePasswordChange}
          />
      </div>
        <button type="submit">login</button>
      </form>
    </div>
  )
}
```

```
export default LoginForm
```

The state and all the functions related to it are defined outside of the component and are passed to the component as props.

Notice that the props are assigned to variables through *destructuring*, which means that instead of writing:

```
const LoginForm = (props) => {
  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={props.handleSubmit}>
        <div>
          username
          <input
            value={props.username}
            onChange={props.handleChange}
            name="username"
          />
        </div>
        // ...
        <button type="submit">login</button>
      </form>
    </div>
  )
}
```

where the properties of the `props` object are accessed through e.g. `props.handleSubmit`, the properties are assigned directly to their own variables.

One fast way of implementing the functionality is to change the `loginForm` function of the *App* component like so:

```
const App = () => {
  const [loginVisible, setLoginVisible] = useState(false)

  // ...

  const loginForm = () => {
    const hideWhenVisible = { display: loginVisible ? 'none' : '' }
    const showWhenVisible = { display: loginVisible ? '' : 'none' }

    return (
      <div>
        <div style={hideWhenVisible}>
          <button onClick={() => setLoginVisible(true)}>log in</button>
        </div>
        <div style={showWhenVisible}>
```

```
          <LoginForm
            username={username}
            password={password}
            handleUsernameChange={({ target }) => setUsername(target.value)}
            handlePasswordChange={({ target }) => setPassword(target.value)}
            handleSubmit={handleLogin}
          />
          <button onClick={() => setLoginVisible(false)}>cancel</button>
        </div>
      </div>
    )
  }

  // ...
}
```

The *App* component state now contains the boolean *loginVisible*, which defines if the login form should be shown to the user or not.

The value of `loginVisible` is toggled with two buttons. Both buttons have their event handlers defined directly in the component:

```
<button onClick={() => setLoginVisible(true)}>log in</button>

<button onClick={() => setLoginVisible(false)}>cancel</button>
```

The visibility of the component is defined by giving the component an inline style rule, where the value of the display property is *none* if we do not want the component to be displayed:

```
const hideWhenVisible = { display: loginVisible ? 'none' : '' }
const showWhenVisible = { display: loginVisible ? '' : 'none' }

<div style={hideWhenVisible}>
  // button
</div>

<div style={showWhenVisible}>
  // button
</div>
```

We are once again using the "question mark" ternary operator. If `loginVisible` is *true*, then the CSS rule of the component will be:

```
display: 'none';
```

If `loginVisible` is *false*, then *display* will not receive any value related to the visibility of the component.

The components children, aka. props.children

The code related to managing the visibility of the login form could be considered to be its own logical entity, and for this reason, it would be good to extract it from the *App* component into a separate component.

Our goal is to implement a new *Togglable* component that can be used in the following way:

```
<Togglable buttonLabel='login'>
  <LoginForm
    username={username}
    password={password}
    handleUsernameChange={({ target }) => setUsername(target.value)}
    handlePasswordChange={({ target }) => setPassword(target.value)}
    handleSubmit={handleLogin}
  />
</Togglable>
```

copy

The way that the component is used is slightly different from our previous components. The component has both opening and closing tags that surround a *LoginForm* component. In React terminology *LoginForm* is a child component of *Togglable*.

We can add any React elements we want between the opening and closing tags of *Togglable*, like this for example:

```
<Togglable buttonLabel="reveal">
  <p>this line is at start hidden</p>
  <p>also this is hidden</p>
</Togglable>
```

copy

The code for the *Togglable* component is shown below:

```
import { useState } from 'react'

const Togglable = (props) => {
  const [visible, setVisible] = useState(false)

  const hideWhenVisible = { display: visible ? 'none' : '' }
  const showWhenVisible = { display: visible ? '' : 'none' }

  const toggleVisibility = () => {
    setVisible(!visible)
```

copy

```
  }

  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>{props.buttonLabel}</button>
      </div>
      <div style={showWhenVisible}>
        {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  )
}

export default Togglable
```

The new and interesting part of the code is `props.children`, which is used for referencing the child components of the component. The child components are the React elements that we define between the opening and closing tags of a component.

This time the children are rendered in the code that is used for rendering the component itself:

```
<div style={showWhenVisible}>
  {props.children}
  <button onClick={toggleVisibility}>cancel</button>
</div>
```

Unlike the "normal" props we've seen before, *children* is automatically added by React and always exists. If a component is defined with an automatically closing `/>` tag, like this:

```
<Note
  key={note.id}
  note={note}
  toggleImportance={() => toggleImportanceOf(note.id)}
/>
```

Then *props.children* is an empty array.

The *Togglable* component is reusable and we can use it to add similar visibility toggling functionality to the form that is used for creating new notes.

Before we do that, let's extract the form for creating notes into a component:

```
const NoteForm = ({ onSubmit, handleChange, value}) => {
  return (
    <div>
```

```
      <h2>Create a new note</h2>

      <form onSubmit={onSubmit}>
        <input
          value={value}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Next let's define the form component inside of a *Togglable* component:

```
<Togglable buttonLabel="new note">
  <NoteForm
    onSubmit={addNote}
    value={newNote}
    handleChange={handleNoteChange}
  />
</Togglable>
```

You can find the code for our current application in its entirety in the *part5-4* branch of this GitHub repository.

State of the forms

The state of the application currently is in the `App` component.

React documentation says the following about where to place the state:

Sometimes, you want the state of two components to always change together. To do it, remove state from both of them, move it to their closest common parent, and then pass it down to them via props. This is known as lifting state up, and it's one of the most common things you will do writing React code.

If we think about the state of the forms, so for example the contents of a new note before it has been created, the `App` component does not need it for anything. We could just as well move the state of the forms to the corresponding components.

The component for creating a new note changes like so:

```
import { useState } from 'react'

const NoteForm = ({ createNote }) => {
  const [newNote, setNewNote] = useState('')

  const addNote = (event) => {
```

```
    event.preventDefault()
    createNote({
      content: newNote,
      important: true
    })

    setNewNote('')
  }

  return (
    <div>
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={event => setNewNote(event.target.value)}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

NOTE At the same time, we changed the behavior of the application so that new notes are important by default, i.e. the field *important* gets the value *true*.

The *newNote* state variable and the event handler responsible for changing it have been moved from the `App` component to the component responsible for the note form.

There is only one prop left, the `createNote` function, which the form calls when a new note is created.

The `App` component becomes simpler now that we have got rid of the *newNote* state and its event handler. The `addNote` function for creating new notes receives a new note as a parameter, and the function is the only prop we send to the form:

```
const App = () => {
  // ...
  const addNote = (noteObject) => {
    noteService
      .create(noteObject)
      .then(returnedNote => {
        setNotes(notes.concat(returnedNote))
      })
  }
  // ...
  const noteForm = () => (
    <Togglable buttonLabel='new note'>
      <NoteForm createNote={addNote} />
```

```
      </Togglable>
    )

    // ...
}
```

We could do the same for the log in form, but we'll leave that for an optional exercise.

The application code can be found on GitHub, branch *part5-5*.

References to components with ref

Our current implementation is quite good; it has one aspect that could be improved.

After a new note is created, it would make sense to hide the new note form. Currently, the form stays visible. There is a slight problem with hiding it, the visibility is controlled with the *visible* state variable inside of the *Togglable* component.

One solution to this would be to move control of the Togglable component's state outside the component. However, we won't do that now, because we want the component to be responsible for its own state. So we have to find another solution, and find a mechanism to change the state of the component externally.

There are several different ways to implement access to a component's functions from outside the component, but let's use the ref mechanism of React, which offers a reference to the component.

Let's make the following changes to the *App* component:

```
import { useState, useEffect, useRef } from 'react'

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

  const noteForm = () => (
    <Togglable buttonLabel='new note' ref={noteFormRef}>
      <NoteForm createNote={addNote} />
    </Togglable>
  )

  // ...
}
```

The useRef hook is used to create a *noteFormRef* reference, that is assigned to the *Togglable* component containing the creation note form. The *noteFormRef* variable acts as a reference to the component. This hook ensures the same reference (ref) that is kept throughout re-renders of the component.

We also make the following changes to the *Togglable* component:

```
import { useState, forwardRef, useImperativeHandle } from 'react'

const Togglable = forwardRef((props, refs) => {
  const [visible, setVisible] = useState(false)

  const hideWhenVisible = { display: visible ? 'none' : '' }
  const showWhenVisible = { display: visible ? '' : 'none' }

  const toggleVisibility = () => {
    setVisible(!visible)
  }

  useImperativeHandle(refs, () => {
    return {
      toggleVisibility
    }
  })

  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>{props.buttonLabel}</button>
      </div>
      <div style={showWhenVisible}>
        {props.children}
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  )
})

export default Togglable
```

The function that creates the component is wrapped inside of a `forwardRef` function call. This way the component can access the ref that is assigned to it.

The component uses the `useImperativeHandle` hook to make its *toggleVisibility* function available outside of the component.

We can now hide the form by calling *noteFormRef.current.toggleVisibility()* after a new note has been created:

```
const App = () => {
  // ...
  const addNote = (noteObject) => {
    noteFormRef.current.toggleVisibility()
    noteService
      .create(noteObject)
      .then(returnedNote => {
        setNotes(notes.concat(returnedNote))
```

```
      })
    }
    // ...
}
```

To recap, the `useImperativeHandle` function is a React hook, that is used for defining functions in a component, which can be invoked from outside of the component.

This trick works for changing the state of a component, but it looks a bit unpleasant. We could have accomplished the same functionality with slightly cleaner code using "old React" class-based components. We will take a look at these class components during part 7 of the course material. So far this is the only situation where using React hooks leads to code that is not cleaner than with class components.

There are also other use cases for refs than accessing React components.

You can find the code for our current application in its entirety in the *part5-6* branch of this GitHub repository.

One point about components

When we define a component in React:

```
const Togglable = () => ...
  // ...
}
```

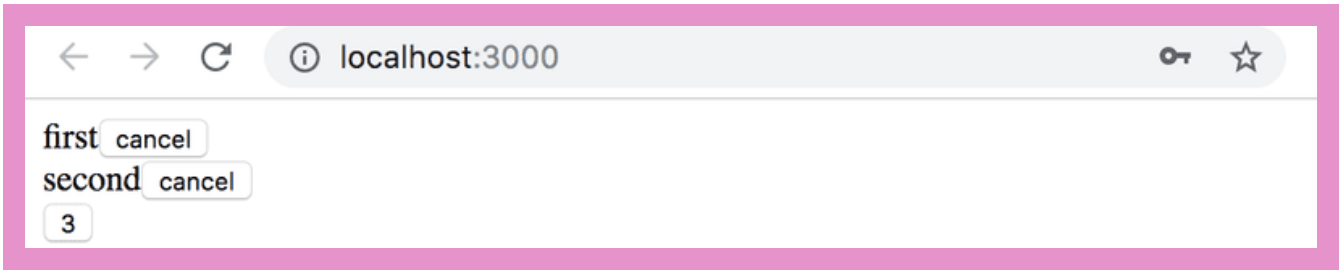
copy

And use it like this:

```
<div>
  <Togglable buttonLabel="1" ref={togglable1}>
    first
  </Togglable>

  <Togglable buttonLabel="2" ref={togglable2}>
    second
  </Togglable>

  <Togglable buttonLabel="3" ref={togglable3}>
    third
  </Togglable>
</div>
```

copy

We create *three separate instances of the component* that all have their separate state:



The *ref* attribute is used for assigning a reference to each of the components in the variables *togglable1*, *togglable2* and *togglable3*.

The updated full stack developer's oath

The number of moving parts increases. At the same time, the likelihood of ending up in a situation where we are looking for a bug in the wrong place increases. So we need to be even more systematic.

So we should once more extend our oath:

Full stack development is *extremely hard*, that is why I will use all the possible means to make it easier

- I will have my browser developer console open all the time
- I will use the network tab of the browser dev tools to ensure that frontend and backend are communicating as I expect
- I will constantly keep an eye on the state of the server to make sure that the data sent there by the frontend is saved there as I expect
- I will keep an eye on the database: does the backend save data there in the right format
- I progress with small steps
- *when I suspect that there is a bug in the frontend, I'll make sure that the backend works as expected*
- *when I suspect that there is a bug in the backend, I'll make sure that the frontend works as expected*
- I will write lots of `console.log` statements to make sure I understand how the code and the tests behave and to help pinpoint problems
- If my code does not work, I will not write more code. Instead, I'll start deleting it until it works or will just return to a state where everything was still working
- If a test does not pass, I'll make sure that the tested functionality works properly in the application
- When I ask for help in the course Discord or Telegram channel or elsewhere I formulate my questions properly, see here how to ask for help

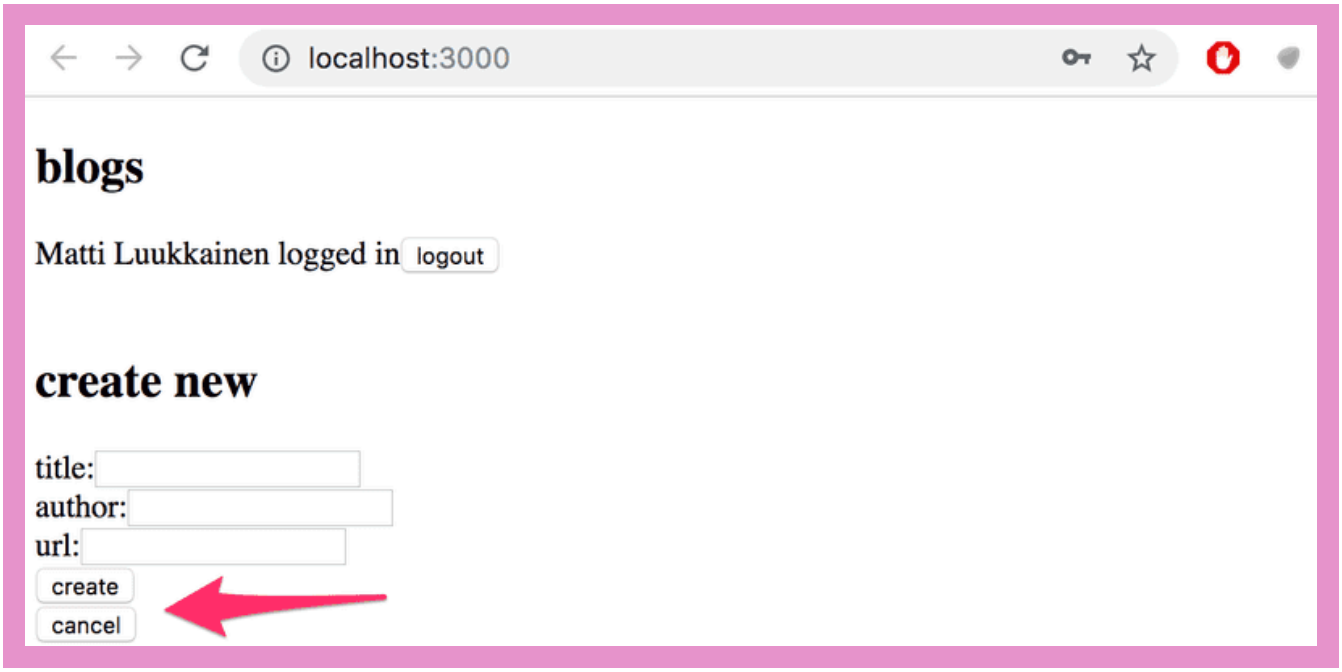
Exercises 5.5.-5.11.

5.5 Blog List Frontend, step 5

Change the form for creating blog posts so that it is only displayed when appropriate. Use functionality similar to what was shown earlier in this part of the course material. If you wish to do so, you can use the *Togglable* component defined in part 5.

By default the form is not visible

It expands when button *create new blog* is clicked



The form hides again after a new blog is created.

5.6 Blog List Frontend, step 6

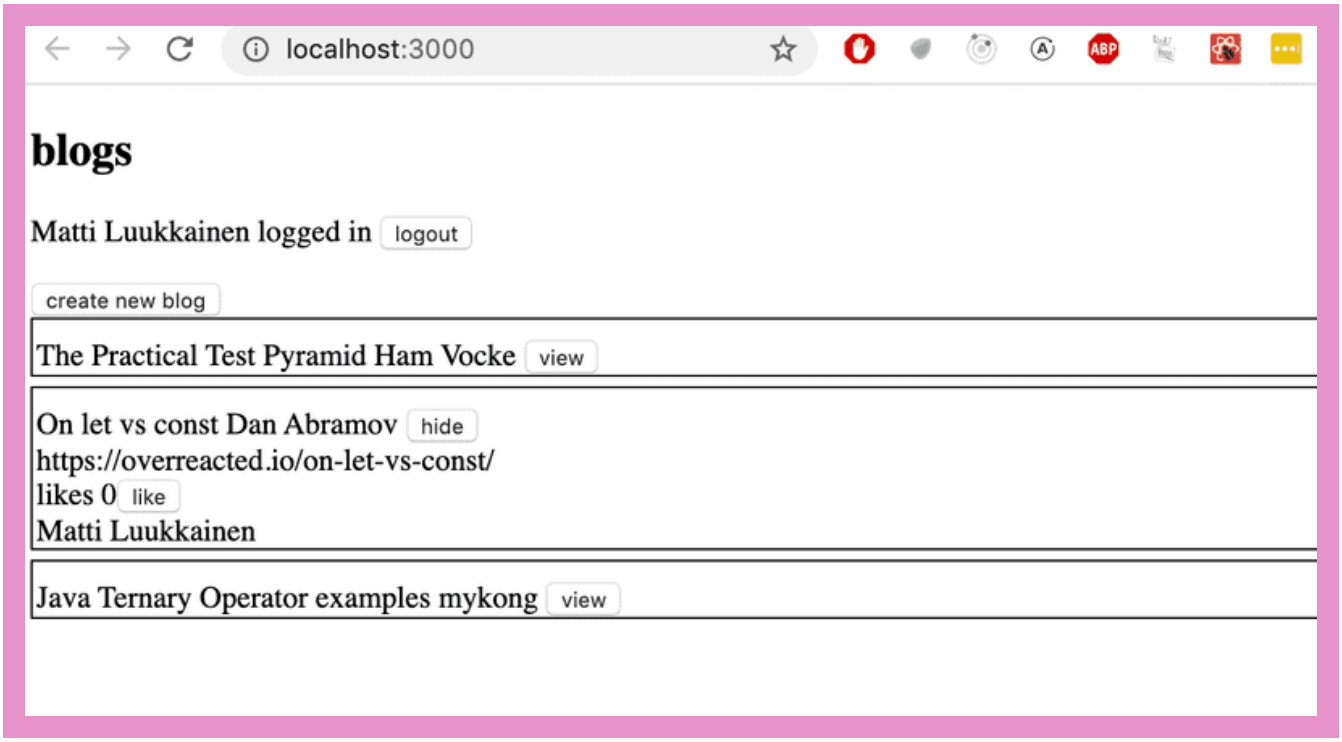
Separate the form for creating a new blog into its own component (if you have not already done so), and move all the states required for creating a new blog to this component.

The component must work like the *NoteForm* component from the material of this part.

5.7 Blog List Frontend, step 7

Let's add a button to each blog, which controls whether all of the details about the blog are shown or not.

Full details of the blog open when the button is clicked.



And the details are hidden when the button is clicked again.

At this point, the *like* button does not need to do anything.

The application shown in the picture has a bit of additional CSS to improve its appearance.

It is easy to add styles to the application as shown in part 2 using inline styles:

```
const Blog = ({ blog }) => {
  const blogStyle = {
    paddingTop: 10,
    paddingLeft: 2,
    border: 'solid',
    borderWidth: 1,
    marginBottom: 5
  }

  return (
    <div style={blogStyle}>
      <div>
        {blog.title} {blog.author}
      </div>
      // ...
```

```
    </div>
)}
```

NB: Even though the functionality implemented in this part is almost identical to the functionality provided by the *Togglable* component, it can't be used directly to achieve the desired behavior. The easiest solution would be to add a state to the blog component that controls if the details are being displayed or not.

5.8: Blog List Frontend, step 8

Implement the functionality for the like button. Likes are increased by making an HTTP `PUT` request to the unique address of the blog post in the backend.

Since the backend operation replaces the entire blog post, you will have to send all of its fields in the request body. If you wanted to add a like to the following blog post:

```
{
  _id: "5a43fde2cbd20b12a2c34e91",
  user: {
    _id: "5a43e6b6c37f3d065eaaa581",
    username: "mluukkai",
    name: "Matti Luukkainen"
  },
  likes: 0,
  author: "Joel Spolsky",
  title: "The Joel Test: 12 Steps to Better Code",
  url: "https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/"
},
```

You would have to make an HTTP PUT request to the address */api/blogs/5a43fde2cbd20b12a2c34e91* with the following request data:

```
{
  user: "5a43e6b6c37f3d065eaaa581",
  likes: 1,
  author: "Joel Spolsky",
  title: "The Joel Test: 12 Steps to Better Code",
  url: "https://www.joelonsoftware.com/2000/08/09/the-joel-test-12-steps-to-better-code/"
}
```

The backend has to be updated too to handle the user reference.

5.9: Blog List Frontend, step 9

We notice that something is wrong. When a blog is liked in the app, the name of the user that added the blog is not shown in its details:

When the browser is reloaded, the information of the person is displayed. This is not acceptable, find out where the problem is and make the necessary correction.

Of course, it is possible that you have already done everything correctly and the problem does not occur in your code. In that case, you can move on.

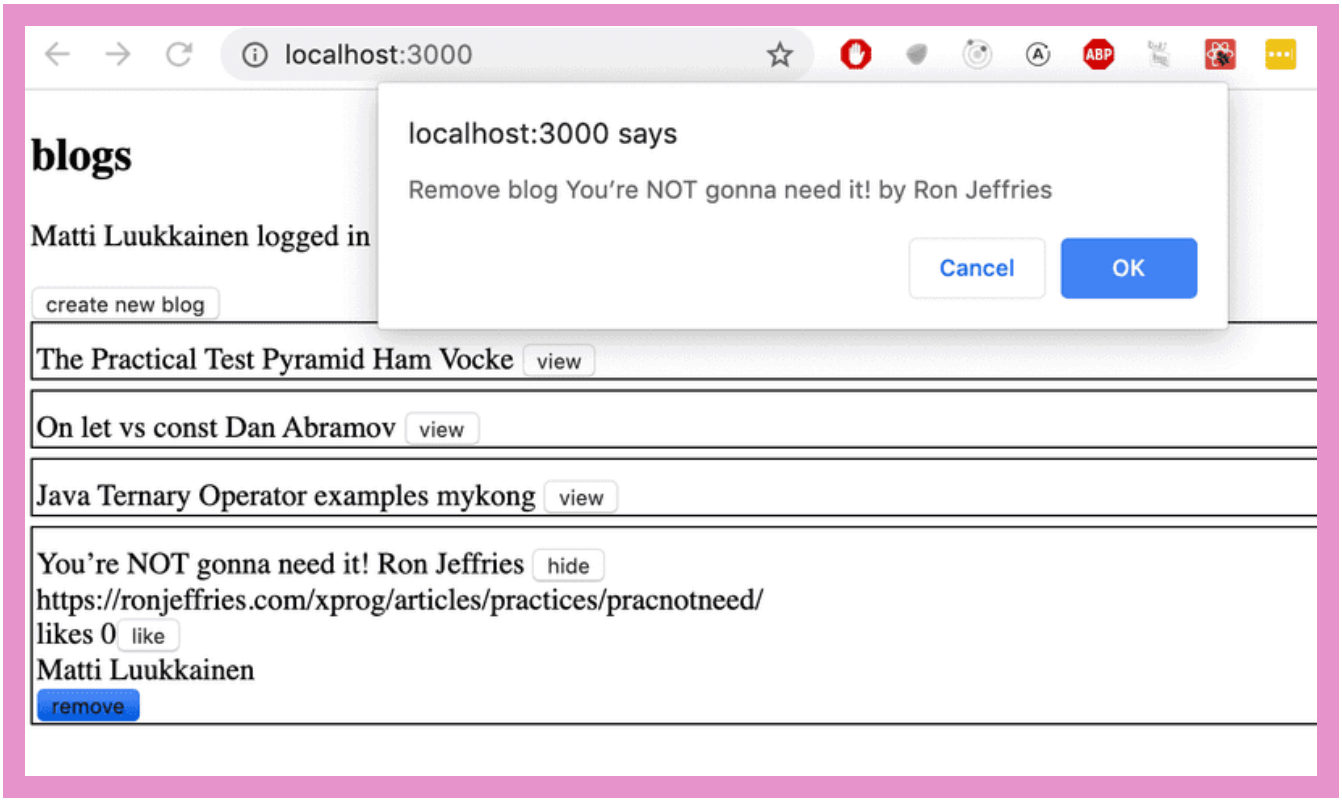
5.10: Blog List Frontend, step 10

Modify the application to sort the blog posts by the number of *likes*. The Sorting can be done with the array sort method.

5.11: Blog List Frontend, step 11

Add a new button for deleting blog posts. Also, implement the logic for deleting blog posts in the frontend.

Your application could look something like this:



The confirmation dialog for deleting a blog post is easy to implement with the `window.confirm` function.

Show the button for deleting a blog post only if the blog post was added by the user.

PropTypes

The *Togglable* component assumes that it is given the text for the button via the *buttonLabel* prop. If we forget to define it to the component:

```
<Togglable> buttonLabel forgotten... </Togglable>
```

copy

The application works, but the browser renders a button that has no label text.

We would like to enforce that when the *Togglable* component is used, the button label text prop must be given a value.

The expected and required props of a component can be defined with the `prop-types` package. Let's install the package:

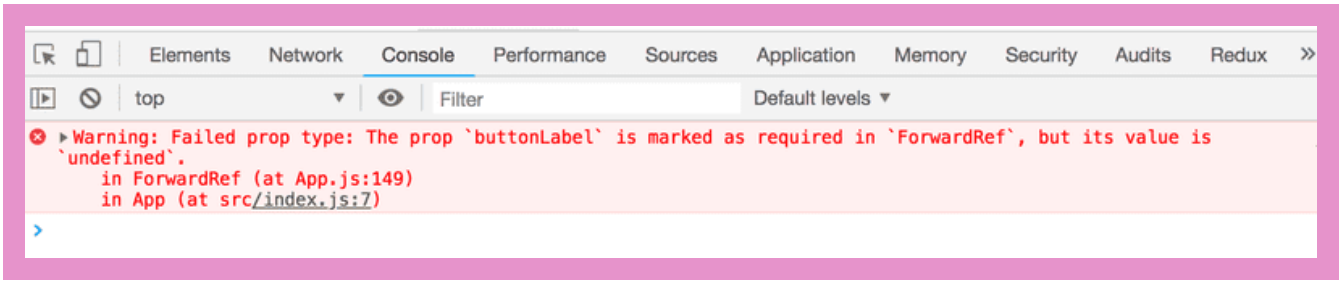
```
npm install prop-types
```

copy

We can define the *buttonLabel* prop as a mandatory or *required* string-type prop as shown below:

```
import PropTypes from 'prop-types'

const Togglable = React.forwardRef((props, ref) => {
  // ..
})

Togglable.propTypes = {
  buttonLabel: PropTypes.string.isRequired
}
```

The console will display the following error message if the prop is left undefined:



The application still works and nothing forces us to define props despite the PropTypes definitions. Mind you, it is extremely unprofessional to leave *any* red output in the browser console.

Let's also define PropTypes to the *LoginForm* component:

```
import PropTypes from 'prop-types'

const LoginForm = ({
   handleSubmit,
   handleUsernameChange,
   handlePasswordChange,
   username,
   password
  }) => {
    // ...
  }

LoginForm.propTypes = {
  handleSubmit: PropTypes.func.isRequired,
  handleUsernameChange: PropTypes.func.isRequired,
  handlePasswordChange: PropTypes.func.isRequired,
  username: PropTypes.string.isRequired,
  password: PropTypes.string.isRequired
}
```

If the type of a passed prop is wrong, e.g. if we try to define the *handleSubmit* prop as a string, then this will result in the following warning:

ESlint

In part 3 we configured the ESlint code style tool to the backend. Let's take ESlint to use in the frontend as well.

Vite has installed ESlint to the project by default, so all that's left for us to do is define our desired configuration in the *.eslintrc.cjs* file.

Let's create a *.eslintrc.cjs* file with the following contents:

```
module.exports = {
  root: true,
  env: {
    browser: true,
    es2020: true,
  },
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:react/jsx-runtime',
    'plugin:react-hooks/recommended',
  ],
  ignorePatterns: ['dist', '.eslintrc.cjs'],
  parserOptions: { ecmaVersion: 'latest', sourceType: 'module' },
  settings: { react: { version: '18.2' } },
  plugins: ['react-refresh'],
  rules: {
    "indent": [
        "error",
        2
    ],
    "linebreak-style": [
        "error",
        "unix"
    ],
    "quotes": [
        "error",
        "single"
    ],
    "semi": [
        "error",
        "never"
    ],
```

```
    "eqeqeq": "error",
    "no-trailing-spaces": "error",
    "object-curly-spacing": [
        "error", "always"
    ],
    "arrow-spacing": [
        "error", { "before": true, "after": true }
    ],
    "no-console": 0,
    "react/react-in-jsx-scope": "off",
    "react/prop-types": 0,
    "no-unused-vars": 0
  },
}
```

NOTE: If you are using Visual Studio Code together with ESLint plugin, you might need to add a workspace setting for it to work. If you are seeing `Failed to load plugin react: Cannot find module 'eslint-plugin-react'` additional configuration is needed. Adding the line `"eslint.workingDirectories": [{ "mode": "auto" }]` to settings.json in the workspace seems to work. See here for more information.

Let's create .eslintignore file with the following contents to the repository root

```
node_modules
dist
.eslintrc.cjs
vite.config.js
```

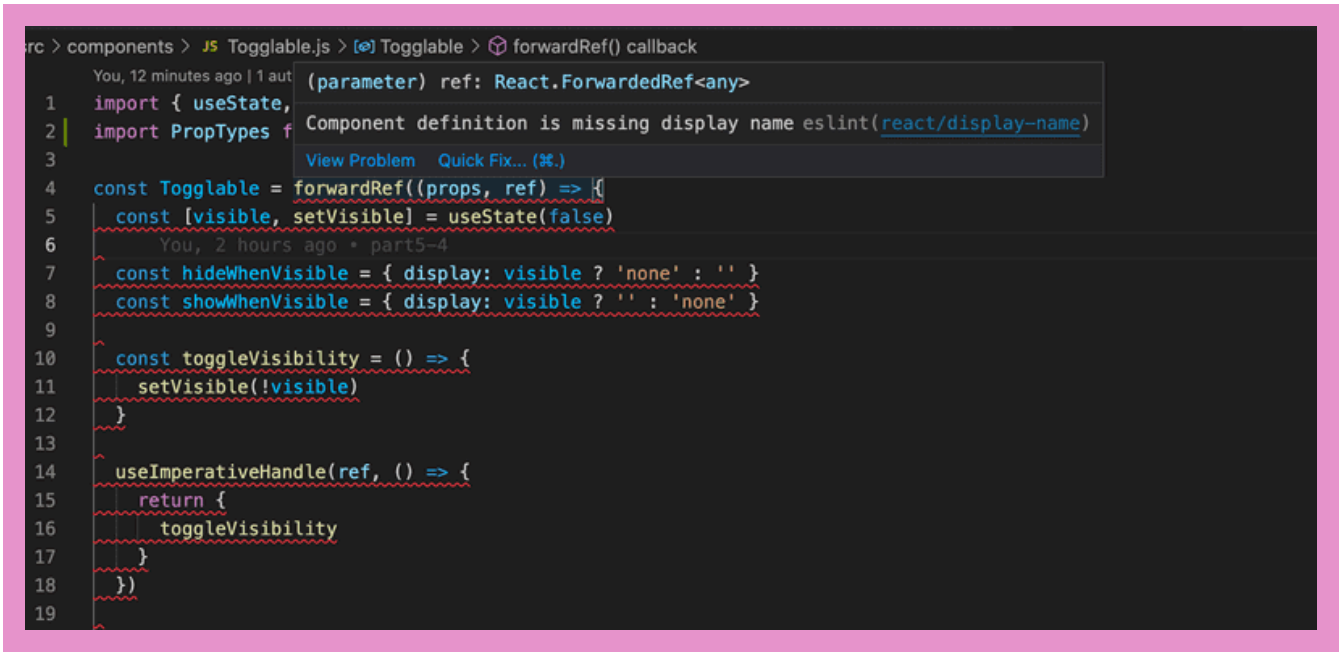
Now the directories `dist` and `node_modules` will be skipped when linting.

As usual, you can perform the linting either from the command line with the command

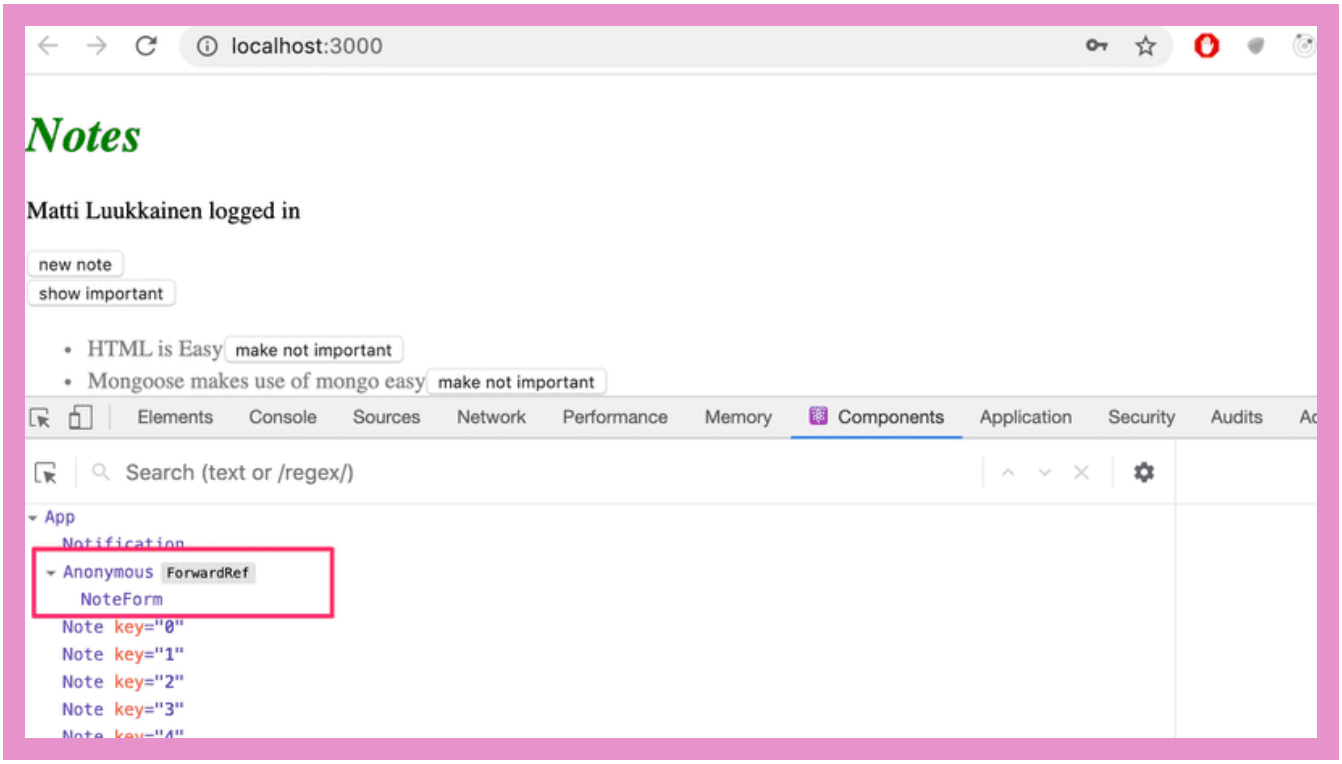
```
npm run lint
```

or using the editor's Eslint plugin.

Component `Togglable` causes a nasty-looking warning *Component definition is missing display name*:



The react-devtools also reveals that the component does not have a name:



Fortunately, this is easy to fix

```
import { useState, useImperativeHandle } from 'react'
import PropTypes from 'prop-types'

const Togglable = React.forwardRef((props, ref) => {
  // ...
})
```

```
Togglable.displayName = 'Togglable'

export default Togglable
```

You can find the code for our current application in its entirety in the *part5-7* branch of this GitHub repository.

Exercise 5.12.

5.12: Blog List Frontend, step 12

Define PropTypes for one of the components of your application, and add ESlint to the project. Define the configuration according to your liking. Fix all of the linter errors.

Vite has installed ESlint to the project by default, so all that's left for you to do is define your desired configuration in the *.eslintrc.cjs* file.

Propose changes to material

Part 5a
Previous part

Part 5c
Next part

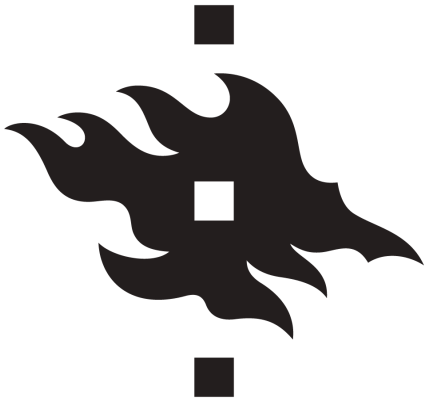
About course

Course contents

FAQ

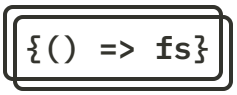
Partners

Challenge



UNIVERSITY OF HELSINKI


HOUSTON



c Testing React apps

The test library used in this part was changed on March 3, 2024 from Jest to Vitest. If you already started this part using Jest, you can see here the old content.

There are many different ways of testing React applications. Let's take a look at them next.

The course previously used the Jest library developed by Facebook to test React components. We are now using the new generation of testing tools from Vite developers called Vitest. Apart from the configurations, the libraries provide the same programming interface, so there is virtually no difference in the test code.

Let's start by installing Vitest and the jsdom library simulating a web browser:

```
npm install --save-dev vitest jsdom
```

copy

In addition to Vitest, we also need another testing library that will help us render components for testing purposes. The current best option for this is react-testing-library which has seen rapid growth in popularity in recent times. It is also worth extending the expressive power of the tests with the library jest-dom.

Let's install the libraries with the command:

```
npm install --save-dev @testing-library/react @testing-library/jest-dom
```

copy

Before we can do the first test, we need some configurations.

We add a script to the *package.json* file to run the tests:

```
{
  "scripts": {
    // ...
    "test": "vitest run"
  }
  // ...
}
```

Let's create a file `testSetup.js` in the project root with the following content

```
import { afterEach } from 'vitest'
import { cleanup } from '@testing-library/react'
import '@testing-library/jest-dom/vitest'

afterEach(() => {
  cleanup()
})
```

Now, after each test, the function `cleanup` is executed to reset jsdom, which is simulating the browser.

Expand the `vite.config.js` file as follows

```
export default defineConfig({
  // ...
  test: {
    environment: 'jsdom',
    globals: true,
    setupFiles: './testSetup.js',
  }
})
```

With `globals: true`, there is no need to import keywords such as `describe`, `test` and `expect` into the tests.

Let's first write tests for the component that is responsible for rendering a note:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important'
```

```
    : 'make important'

  return (
    <li className='note'>
      {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Notice that the *li* element has the value *note* for the CSS attribute className, that could be used to access the component in our tests.

Rendering the component for tests

We will write our test in the *src/components/Note.test.jsx* file, which is in the same directory as the component itself.

The first test verifies that the component renders the contents of the note:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-
library')
  expect(element).toBeDefined()
})
```

After the initial configuration, the test renders the component with the render function provided by the react-testing-library:

```
render(<Note note={note} />)
```

Normally React components are rendered to the DOM. The render method we used renders the components in a format that is suitable for tests without rendering them to the DOM.

We can use the object screen to access the rendered component. We use screen's method getByText to search for an element that has the note content and ensure that it exists:

```
  const element = screen.getByText('Component testing is done with react-testing-
library')
  expect(element).toBeDefined()
```

The existence of an element is checked using Vitest's `expect` command. Expect generates an assertion for its argument, the validity of which can be tested using various condition functions. Now we used `toBeDefined` which tests whether the `element` argument of expect exists.

Run the test with command `npm test`:

```
$ npm test

> notes-frontend@0.0.0 test
> vitest


 DEV  v1.3.1 /Users/mluukkai/opetus/2024-fs/part3/notes-frontend

 ✓ src/components/Note.test.jsx (1)
   ✓ renders content

 Test Files  1 passed (1)
      Tests  1 passed (1)
   Start at  17:05:37
   Duration  812ms (transform 31ms, setup 220ms, collect 11ms, tests 14ms, environment
395ms, prepare 70ms)


 PASS  Waiting for file changes...
```

Eslint complains about the keywords `test` and `expect` in the tests. The problem can be solved by installing `eslint-plugin-vitest-globals`:

```
npm install --save-dev eslint-plugin-vitest-globals
```

and enable the plugin by editing the `.eslint.cjs` file as follows:

```
module.exports = {
  root: true,
  env: {
    browser: true,
    es2020: true,
    "vitest-globals/env": true
  },
```

```
  extends: [
    'eslint:recommended',
    'plugin:react/recommended',
    'plugin:react/jsx-runtime',
    'plugin:react-hooks/recommended',
    'plugin:vitest-globals/recommended',
  ],
  // ...
}
```

Test file location

In React there are (at least) two different conventions for the test file's location. We created our test files according to the current standard by placing them in the same directory as the component being tested.

The other convention is to store the test files "normally" in a separate `test` directory. Whichever convention we choose, it is almost guaranteed to be wrong according to someone's opinion.

I do not like this way of storing tests and application code in the same directory. The reason we choose to follow this convention is that it is configured by default in applications created by Vite or create-react-app.

Searching for content in a component

The react-testing-library package offers many different ways of investigating the content of the component being tested. In reality, the `expect` in our test is not needed at all:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-library')

  expect(element).toBeDefined()
})
```

Test fails if `getByText` does not find the element it is looking for.

We could also use CSS-selectors to find rendered elements by using the method querySelector of the object container that is one of the fields returned by the render:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const { container } = render(<Note note={note} />)

  const div = container.querySelector('.note')
  expect(div).toHaveTextContent(
    'Component testing is done with react-testing-library'
  )
})
```

NB: A more consistent way of selecting elements is using a data attribute that is specifically defined for testing purposes. Using `react-testing-library`, we can leverage the getByTestId method to select elements with a specified `data-testid` attribute.

Debugging tests

We typically run into many different kinds of problems when writing our tests.

Object `screen` has method debug that can be used to print the HTML of a component to the terminal. If we change the test as follows:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  screen.debug()

  // ...

})
```

the HTML gets printed to the console:

```
console.log
  <body>
    <div>
      <li
        class="note"
      >
        Component testing is done with react-testing-library
        <button>
          make not important
        </button>
      </li>
    </div>
  </body>
```

It is also possible to use the same method to print a wanted element to console:

```
import { render, screen } from '@testing-library/react'
import Note from './Note'

test('renders content', () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Component testing is done with react-testing-
library')

  screen.debug(element)

  expect(element).toBeDefined()
})
```

Now the HTML of the wanted element gets printed:

```
  <li
    class="note"
  >
    Component testing is done with react-testing-library
    <button>
      make not important
```

```
    </button>
  </li>
```

Clicking buttons in tests

In addition to displaying content, the *Note* component also makes sure that when the button associated with the note is pressed, the `toggleImportance` event handler function gets called.

Let us install a library user-event that makes simulating user input a bit easier:

```
npm install --save-dev @testing-library/user-event
```

copy

Testing this functionality can be accomplished like this:

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import Note from './Note'

// ...

test('clicking the button calls event handler once', async () => {
  const note = {
    content: 'Component testing is done with react-testing-library',
    important: true
  }

  const mockHandler = vi.fn()

  render(
    <Note note={note} toggleImportance={mockHandler} />
  )

  const user = userEvent.setup()
  const button = screen.getByText('make not important')
  await user.click(button)

  expect(mockHandler.mock.calls).toHaveLength(1)
})
```

copy

There are a few interesting things related to this test. The event handler is a mock function defined with Vitest:

```
const mockHandler = vi.fn()
```

copy

A session is started to interact with the rendered component:

```
const user = userEvent.setup()
```

The test finds the button *based on the text* from the rendered component and clicks the element:

```
const button = screen.getByText('make not important')
await user.click(button)
```

Clicking happens with the method click of the userEvent-library.

The expectation of the test uses toHaveLength to verify that the *mock function* has been called exactly once:

```
expect(mockHandler.mock.calls).toHaveLength(1)
```

The calls to the mock function are saved to the array mock.calls within the mock function object.

Mock objects and functions are commonly used stub components in testing that are used for replacing dependencies of the components being tested. Mocks make it possible to return hardcoded responses, and to verify the number of times the mock functions are called and with what parameters.

In our example, the mock function is a perfect choice since it can be easily used for verifying that the method gets called exactly once.

Tests for the *Togglable* component

Let's write a few tests for the *Togglable* component. Let's add the *togglableContent* CSS classname to the div that returns the child components.

```
const Togglable = forwardRef((props, ref) => {
  // ...

  return (
    <div>
      <div style={hideWhenVisible}>
        <button onClick={toggleVisibility}>
          {props.buttonLabel}
        </button>
      </div>
      <div style={showWhenVisible} className="togglableContent">
        {props.children}
```

```
        <button onClick={toggleVisibility}>cancel</button>
      </div>
    </div>
  )
})
```

The tests are shown below:

```
import { render, screen } from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import Togglable from './Togglable'

describe('<Togglable />', () => {
  let container

  beforeEach(() => {
    container = render(
      <Togglable buttonLabel="show...">
        <div className="testDiv" >
          togglable content
        </div>
      </Togglable>
    ).container
  })

  test('renders its children', async () => {
    await screen.findAllByText('togglable content')
  })

  test('at start the children are not displayed', () => {
    const div = container.querySelector('.togglableContent')
    expect(div).toHaveStyle('display: none')
  })

  test('after clicking the button, children are displayed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    await user.click(button)

    const div = container.querySelector('.togglableContent')
    expect(div).not.toHaveStyle('display: none')
  })
})
```

The `beforeEach` function gets called before each test, which then renders the *Togglable* component and saves the field `container` of the returned value.

The first test verifies that the *Togglable* component renders its child component

```
<div className="testDiv">
  togglable content
</div>
```

The remaining tests use the toHaveStyle method to verify that the child component of the *Togglable* component is not visible initially, by checking that the style of the *div* element contains `{ display: 'none' }`. Another test verifies that when the button is pressed the component is visible, meaning that the style for hiding it *is no longer* assigned to the component.

Let's also add a test that can be used to verify that the visible content can be hidden by clicking the second button of the component:

```
describe('<Togglable />', () => {

  // ...

  test('toggled content can be closed', async () => {
    const user = userEvent.setup()
    const button = screen.getByText('show...')
    await user.click(button)

    const closeButton = screen.getByText('cancel')
    await user.click(closeButton)

    const div = container.querySelector('.togglableContent')
    expect(div).toHaveStyle('display: none')
  })
})
```

Testing the forms

We already used the `click` function of the user-event in our previous tests to click buttons.

```
const user = userEvent.setup()
const button = screen.getByText('show...')
await user.click(button)
```

We can also simulate text input with *userEvent*.

Let's make a test for the *NoteForm* component. The code of the component is as follows.

```
import { useState } from 'react'

const NoteForm = ({ createNote }) => {
```

```
  const [newNote, setNewNote] = useState('')

  const handleChange = (event) => {
    setNewNote(event.target.value)
  }

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true,
    })

    setNewNote('')
  }

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}

export default NoteForm
```

The form works by calling the function received as props `createNote`, with the details of the new note.

The test is as follows:

```
import { render, screen } from '@testing-library/react'
import NoteForm from './NoteForm'
import userEvent from '@testing-library/user-event'

test('<NoteForm /> updates parent state and calls onSubmit', async () => {
  const createNote = vi.fn()
  const user = userEvent.setup()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByRole('textbox')
  const sendButton = screen.getByText('save')

  await user.type(input, 'testing a form...')
  await user.click(sendButton)
```

```
  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing a form...')
})
```

Tests get access to the input field using the function `getByRole`.

The method `type` of the userEvent is used to write text to the input field.

The first test expectation ensures that submitting the form calls the `createNote` method. The second expectation checks that the event handler is called with the right parameters - that a note with the correct content is created when the form is filled.

It's worth noting that the good old `console.log` works as usual in the tests. For example, if you want to see what the calls stored by the mock-object look like, you can do the following

```
test('<NoteForm /> updates parent state and calls onSubmit', async() => {
  const user = userEvent.setup()
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByRole('textbox')
  const sendButton = screen.getByText('save')

  await user.type(input, 'testing a form...')
  await user.click(sendButton)

  console.log(createNote.mock.calls)
})
```

copy

In the middle of running the tests, the following is printed in the console:

```
[ [ { content: 'testing a form...', important: true } ] ]
```

copy

About finding the elements

Let us assume that the form has two input fields

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
```

copy

```
        <input
          value={newNote}
          onChange={handleChange}
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Now the approach that our test uses to find the input field

```
const input = screen.getByRole('textbox')
```

copy

would cause an error:

```
FAIL src/components/NoteForm.test.js
  ● <NoteForm /> updates parent state and calls onSubmit

    TestingLibraryElementError: Found multiple elements with the role "textbox"

    Here are the matching elements:

    Ignored nodes: comments, <script />, <style />
    <input
      value=""
    />

    Ignored nodes: comments, <script />, <style />
    <input />

    (If this is intentional, then use the `*AllBy*` variant of the query (like
queryAllByText`, `getAllByText`, or `findAllByText`)).
```

The error message suggests using *getAllByRole*. The test could be fixed as follows:

```
const inputs = screen.getAllByRole('textbox')

await user.type(inputs[0], 'testing a form...')
```

copy

Method *getAllByRole* now returns an array and the right input field is the first element of the array. However, this approach is a bit suspicious since it relies on the order of the input fields.

Quite often input fields have a *placeholder* text that hints user what kind of input is expected. Let us add a placeholder to our form:

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
          placeholder='write note content here'
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

Now finding the right input field is easy with the method getByPlaceholderText:

```
test('<NoteForm /> updates parent state and calls onSubmit', () => {
  const createNote = vi.fn()

  render(<NoteForm createNote={createNote} />)

  const input = screen.getByPlaceholderText('write note content here')
  const sendButton = screen.getByText('save')

  userEvent.type(input, 'testing a form...')
  userEvent.click(sendButton)

  expect(createNote.mock.calls).toHaveLength(1)
  expect(createNote.mock.calls[0][0].content).toBe('testing a form...')
})
```

The most flexible way of finding elements in tests is the method *querySelector* of the `container` object, which is returned by `render`, as was mentioned earlier in this part. Any CSS selector can be

used with this method for searching elements in tests.

Consider eg. that we would define a unique `id` to the input field:

```
const NoteForm = ({ createNote }) => {
  // ...

  return (
    <div className="formDiv">
      <h2>Create a new note</h2>

      <form onSubmit={addNote}>
        <input
          value={newNote}
          onChange={handleChange}
          id='note-input'
        />
        <input
          value={...}
          onChange={...}
        />
        <button type="submit">save</button>
      </form>
    </div>
  )
}
```

The input element could now be found in the test as follows:

```
const { container } = render(<NoteForm createNote={createNote} />)

const input = container.querySelector('#note-input')
```

However, we shall stick to the approach of using `getByPlaceholderText` in the test.

Let us look at a couple of details before moving on. Let us assume that a component would render text to an HTML element as follows:

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li className='note'>
      Your awesome note: {note.content}
      <button onClick={toggleImportance}>{label}</button>
    </li>
```

```
    )
}

export default Note
```

the `getByText` method that the test uses does *not* find the element

```
test('renders content', () => {
  const note = {
    content: 'Does not work anymore :(',
    important: true
  }

  render(<Note note={note} />)

  const element = screen.getByText('Does not work anymore :(')

  expect(element).toBeDefined()
})
```

The `getByText` method looks for an element that has the **same text** that it has as a parameter, and nothing more. If we want to look for an element that *contains* the text, we could use an extra option:

```
const element = screen.getByText(
  'Does not work anymore :(', { exact: false }
)
```

or we could use the `findByText` method:

```
const element = await screen.findByText('Does not work anymore :(')
```

It is important to notice that, unlike the other `ByText` methods, `findByText` returns a promise!

There are situations where yet another form of the `queryByText` method is useful. The method returns the element but *it does not cause an exception* if it is not found.

We could eg. use the method to ensure that something *is not rendered* to the component:

```
test('does not render this', () => {
  const note = {
    content: 'This is a reminder',
    important: true
  }
```

```
  render(<Note note={note} />)

  const element = screen.queryByText('do not want this thing to be rendered')
  expect(element).toBeNull()
})
```

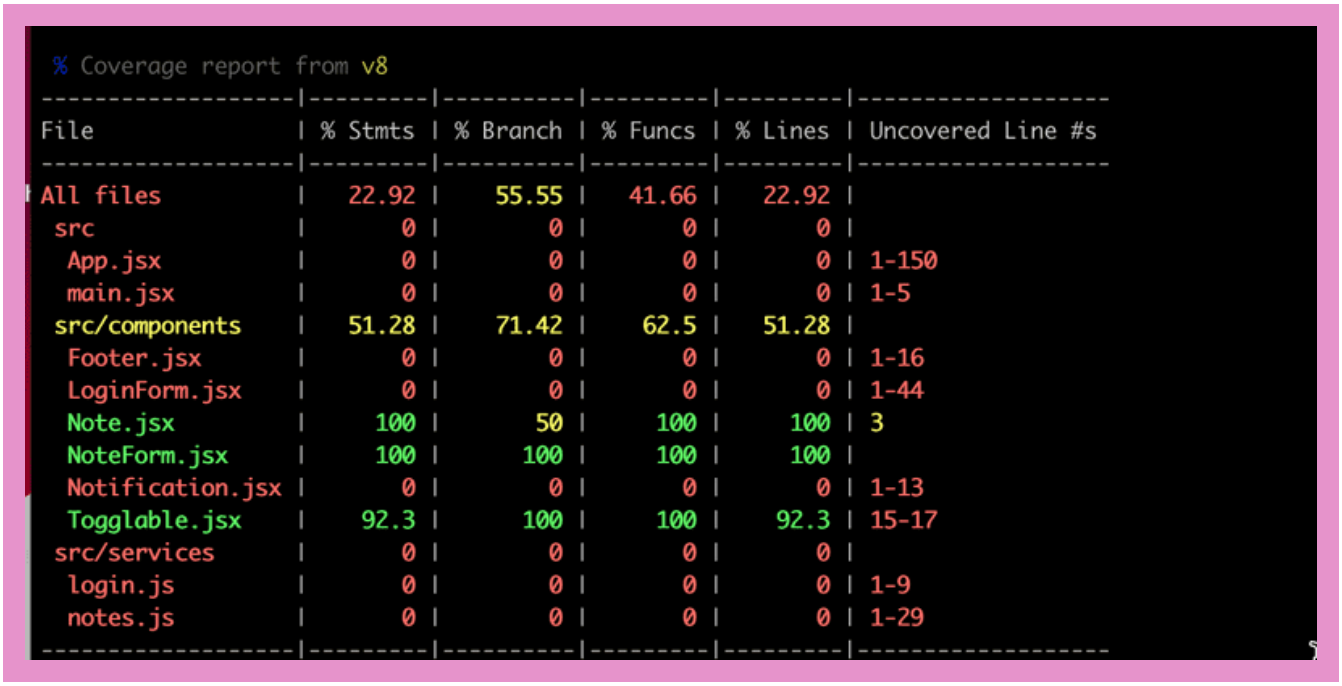
Test coverage

We can easily find out the coverage of our tests by running them with the command.

```
npm test -- --coverage
```

copy

The first time you run the command, Vitest will ask you if you want to install the required library `@vitest/coverage-v8`. Install it, and run the command again:



A HTML report will be generated to the *coverage* directory. The report will tell us the lines of untested code in each component:

You can find the code for our current application in its entirety in the *part5-8* branch of this GitHub repository.

Exercises 5.13.-5.16.

5.13: Blog List Tests, step 1

Make a test, which checks that the component displaying a blog renders the blog's title and author, but does not render its URL or number of likes by default.

Add CSS classes to the component to help the testing as necessary.

5.14: Blog List Tests, step 2

Make a test, which checks that the blog's URL and number of likes are shown when the button controlling the shown details has been clicked.

5.15: Blog List Tests, step 3

Make a test, which ensures that if the *like* button is clicked twice, the event handler the component received as props is called twice.

5.16: Blog List Tests, step 4

Make a test for the new blog form. The test should check, that the form calls the event handler it received as props with the right details when a new blog is created.

Frontend integration tests

In the previous part of the course material, we wrote integration tests for the backend that tested its logic and connected the database through the API provided by the backend. When writing these tests, we made the conscious decision not to write unit tests, as the code for that backend is fairly simple, and it is likely that bugs in our application occur in more complicated scenarios than unit tests are well suited for.

So far all of our tests for the frontend have been unit tests that have validated the correct functioning of individual components. Unit testing is useful at times, but even a comprehensive suite of unit tests is not enough to validate that the application works as a whole.

We could also make integration tests for the frontend. Integration testing tests the collaboration of multiple components. It is considerably more difficult than unit testing, as we would have to for example mock data from the server. We chose to concentrate on making end-to-end tests to test the whole application. We will work on the end-to-end tests in the last chapter of this part.

Snapshot testing

Vitest offers a completely different alternative to "traditional" testing called snapshot testing. The interesting feature of snapshot testing is that developers do not need to define any tests themselves, it is simple enough to adopt snapshot testing.

The fundamental principle is to compare the HTML code defined by the component after it has changed to the HTML code that existed before it was changed.

If the snapshot notices some change in the HTML defined by the component, then either it is new functionality or a "bug" caused by accident. Snapshot tests notify the developer if the HTML code of the component changes. The developer has to tell Jest if the change was desired or undesired. If the change to the HTML code is unexpected, it strongly implies a bug, and the developer can become aware of these potential issues easily thanks to snapshot testing.

Propose changes to material

Part 5b
Previous part

Part 5d
Next part

About course


HOUSTON

{() => fs}

Fullstack > Part 5 > End to end testing: Playwright

d End to end testing: Playwright

So far we have tested the backend as a whole on an API level using integration tests and tested some frontend components using unit tests.

Next, we will look into one way to test the system as a whole using *End to End* (E2E) tests.

We can do E2E testing of a web application using a browser and a testing library. There are multiple libraries available. One example is Selenium, which can be used with almost any browser. Another browser option is so-called headless browsers, which are browsers with no graphical user interface. For example, Chrome can be used in headless mode.

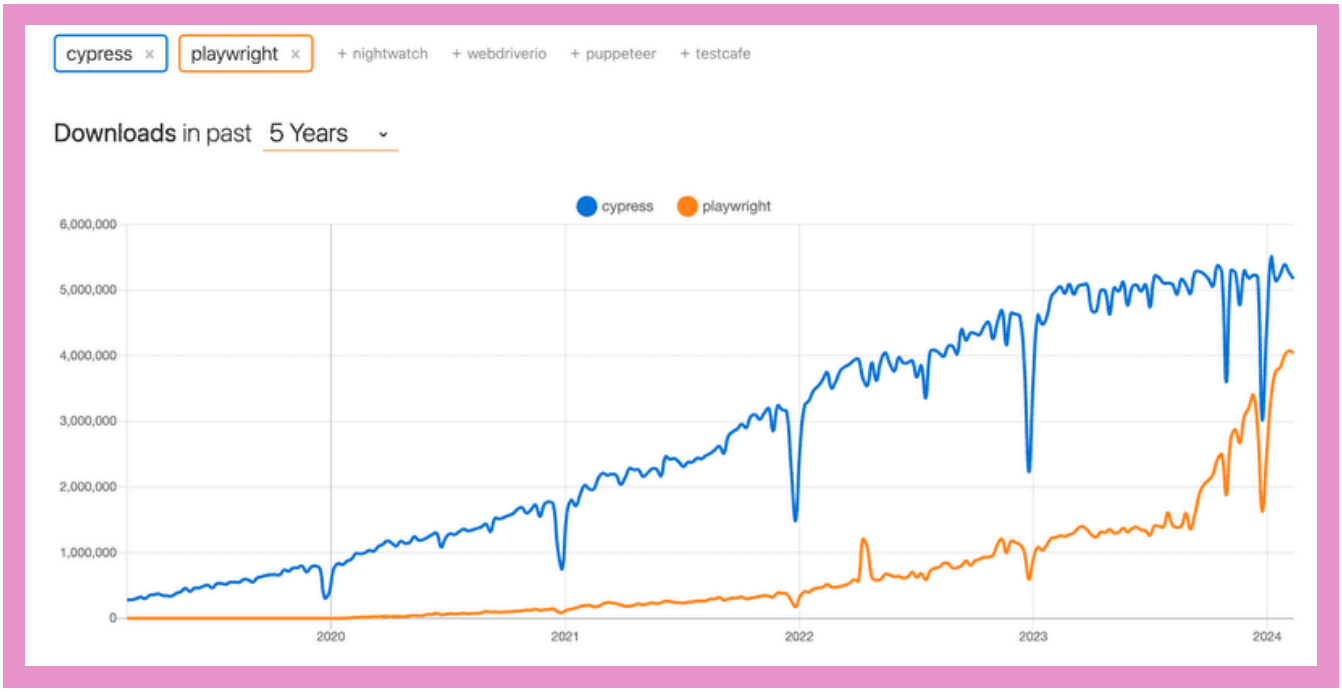
E2E tests are potentially the most useful category of tests because they test the system through the same interface as real users use.

They do have some drawbacks too. Configuring E2E tests is more challenging than unit or integration tests. They also tend to be quite slow, and with a large system, their execution time can be minutes or even hours. This is bad for development because during coding it is beneficial to be able to run tests as often as possible in case of code regressions.

E2E tests can also be flaky. Some tests might pass one time and fail another, even if the code does not change at all.

Perhaps the two easiest libraries for End to End testing at the moment are Cypress and Playwright.

From the statistics on npmtrends.com we see that Cypress, which dominated the market for the last five years, is still clearly the number one, but Playwright is on a rapid rise:



This course has been using Cypress for years. Now Playwright is a new addition. You can choose whether to complete the E2E testing part of the course with Cypress or Playwright. The operating principles of both libraries are very similar, so your choice is not very important. However, Playwright is now the preferred E2E library for the course.

If your choice is Playwright, please proceed. If you end up using Cypress, go here.

Playwright

So Playwright is a newcomer to the End to End tests, which started to explode in popularity towards the end of 2023. Playwright is roughly on a par with Cypress in terms of ease of use. The libraries are slightly different in terms of how they work. Cypress is radically different from most libraries suitable for E2E testing, as Cypress tests are run entirely within the browser. Playwright's tests, on the other hand, are executed in the Node process, which is connected to the browser via programming interfaces.

Many blogs have been written about library comparisons, e.g. this and this.

It is difficult to say which library is better. One advantage of Playwright is its browser support; Playwright supports Chrome, Firefox and Webkit-based browsers like Safari. Currently, Cypress includes support for all these browsers, although Webkit support is experimental and does not support all of Cypress features. At the time of writing (1.3.2024), my personal preference leans slightly towards Playwright.

Now let's explore Playwright.

Initializing tests

Unlike the backend tests or unit tests done on the React front-end, End to End tests do not need to be located in the same npm project where the code is. Let's make a completely separate project for the E2E

tests with the `npm init` command. Then install Playwright by running in the new project directory the command:

```
npm init playwright@latest
```

copy

The installation script will ask a few questions, answer them as follows:

```
→  e2e npm init playwright@latest
Getting started with writing end-to-end tests with Playwright:
Initializing project in '.'
✔ Do you want to use TypeScript or JavaScript? · JavaScript
✔ Where to put your end-to-end tests? · tests
✔ Add a GitHub Actions workflow? (y/N) · false
✔ Install Playwright browsers (can be done manually via 'npx playwright install')? (Y/n) · true
Installing Playwright Test (npm install --save-dev @playwright/test)…
```

Let's define an npm script for running tests and test reports in `package.json`:

```
{
  // ...
  "scripts": {
    "test": "playwright test",
    "test:report": "playwright show-report"
  },
  // ...
}
```

copy

During installation, the following is printed to the console:

```
And check out the following files:
  - ./tests/example.spec.js - Example end-to-end test
  - ./tests-examples/demo-todo-app.spec.js - Demo Todo App end-to-end tests
  - ./playwright.config.js - Playwright Test configuration
```

copy

that is, the location of a few example tests for the project that the installation has created.

Let's run the tests:

```
$ npm test

> notes-e2e@1.0.0 test
> playwright test
```

copy

```
Running 6 tests using 5 workers
  6 passed (3.9s)

To open last HTML report run:

  npx playwright show-report
```

The tests pass. A more detailed test report can be opened either with the command suggested by the output, or with the npm script we just defined:

```
npm run test:report
```

copy

Tests can also be run via the graphical UI with the command:

```
npm run test -- --ui
```

copy

Sample tests look like this:

```
const { test, expect } = require('@playwright/test');

test('has title', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Expect a title "to contain" a substring.
  await expect(page).toHaveTitle(/Playwright/);
});

test('get started link', async ({ page }) => {
  await page.goto('https://playwright.dev/');

  // Click the get started link.
  await page.getByRole('link', { name: 'Get started' }).click();

  // Expects page to have a heading with the name of Installation.
  await expect(page.getByRole('heading', { name: 'Installation' })).toBeVisible();
});
```

copy

The first line of the test functions says that the tests are testing the page at https://playwright.dev/ .

Testing our own code

Now let's remove the sample tests and start testing our own application.

Playwright tests assume that the system under test is running when the tests are executed. Unlike, for example, backend integration tests, Playwright tests *do not start* the system under test during testing.

Let's make an npm script for the *backend*, which will enable it to be started in testing mode, i.e. so that *NODE_ENV* gets the value *test*.

```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "NODE_ENV=test node --test",
    "start:test": "NODE_ENV=test node index.js"
  },
  // ...
}
```

Let's start the frontend and backend, and create the first test file for the application `tests/note_app.spec.js` :

```
const { test, expect } = require('@playwright/test')

test('front page can be opened', async ({ page }) => {
  await page.goto('http://localhost:5173')

  const locator = await page.getByText('Notes')
  await expect(locator).toBeVisible()
  await expect(page.getByText('Note app, Department of Computer Science, University of
Helsinki 2023')).toBeVisible()
})
```

First, the test opens the application with the method `page.goto`. After this, it uses the `page.getByText` to get a `locator` that corresponds to the element where the text *Notes* is found.

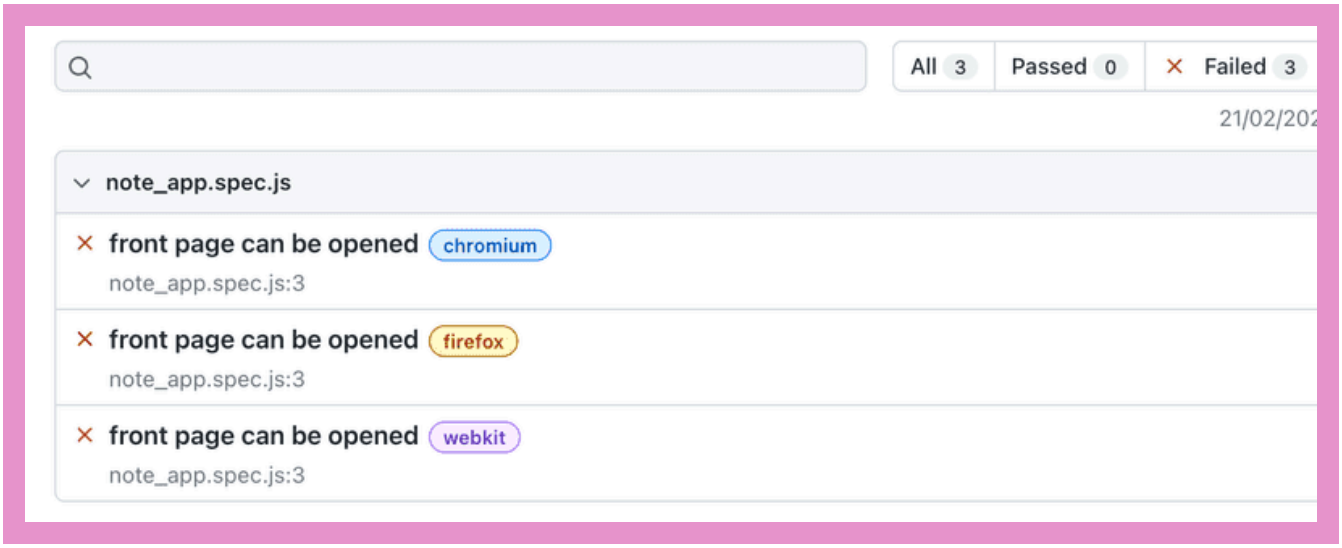
The method `toBeVisible` ensures that the element corresponding to the locator is visible at the page.

The second check is done without using the auxiliary variable.

We notice that the year has changed. Let's change the test as follows:

```
const { test, expect } = require('@playwright/test')

test('front page can be opened', async ({ page }) => {
  await page.goto('http://localhost:5173')

  const locator = await page.getByText('Notes')
  await expect(locator).toBeVisible()
  await expect(page.getByText('Note app, Department of Computer Science, University of Helsinki 2024')).toBeVisible()
})
```

As expected, the test fails. Playwright opens the test report in the browser and it becomes clear that Playwright has actually performed the tests with three different browsers: Chrome, Firefox and Webkit, i.e. the browser engine used by Safari:



By clicking on the report of one of the browsers, we can see a more detailed error message:

In the big picture, it is of course a very good thing that the testing takes place with all three commonly used browser engines, but this is slow, and when developing the tests it is probably best to carry them out mainly with only one browser. You can define the browser engine to be used with the command line parameter:

```
npm test -- --project chromium
```

copy

Now let's correct the outdated year in the frontend code that caused the error.

Before we continue, let's add a `describe` block to the tests:

```
const { test, describe, expect } = require('@playwright/test')

describe('Note app', () => {
  test('front page can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    const locator = await page.getByText('Notes')
    await expect(locator).toBeVisible()
    await expect(page.getByText('Note app, Department of Computer Science, University of
Helsinki 2024')).toBeVisible()
  })
})
```

copy

Before we move on, let's break the tests one more time. We notice that the execution of the tests is quite fast when they pass, but much slower if the they do not pass. The reason for this is that Playwright's policy is to wait for searched elements until they are rendered and ready for action. If the element is not found, a `TimeoutError` is raised and the test fails. Playwright waits for elements by default for 5 or 30 seconds depending on the functions used in testing.

When developing tests, it may be wiser to reduce the waiting time to a few seconds. According to the documentation, this can be done by changing the file `playwright.config.js` as follows:

```
module.exports = defineConfig({
  timeout: 3000,
  fullyParallel: false,
  workers: 1,
  // ...
})
```

We also made two other changes to the file, and specified that all tests be executed one at a time. With the default configuration, the execution happens in parallel, and since our tests use a database, parallel execution causes problems.

Writing on the form

Let's write a new test that tries to log into the application. Let's assume that a user is stored in the database, with username *mluukkai* and password *salainen*.

Let's start by opening the login form.

```
describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
  })
})
```

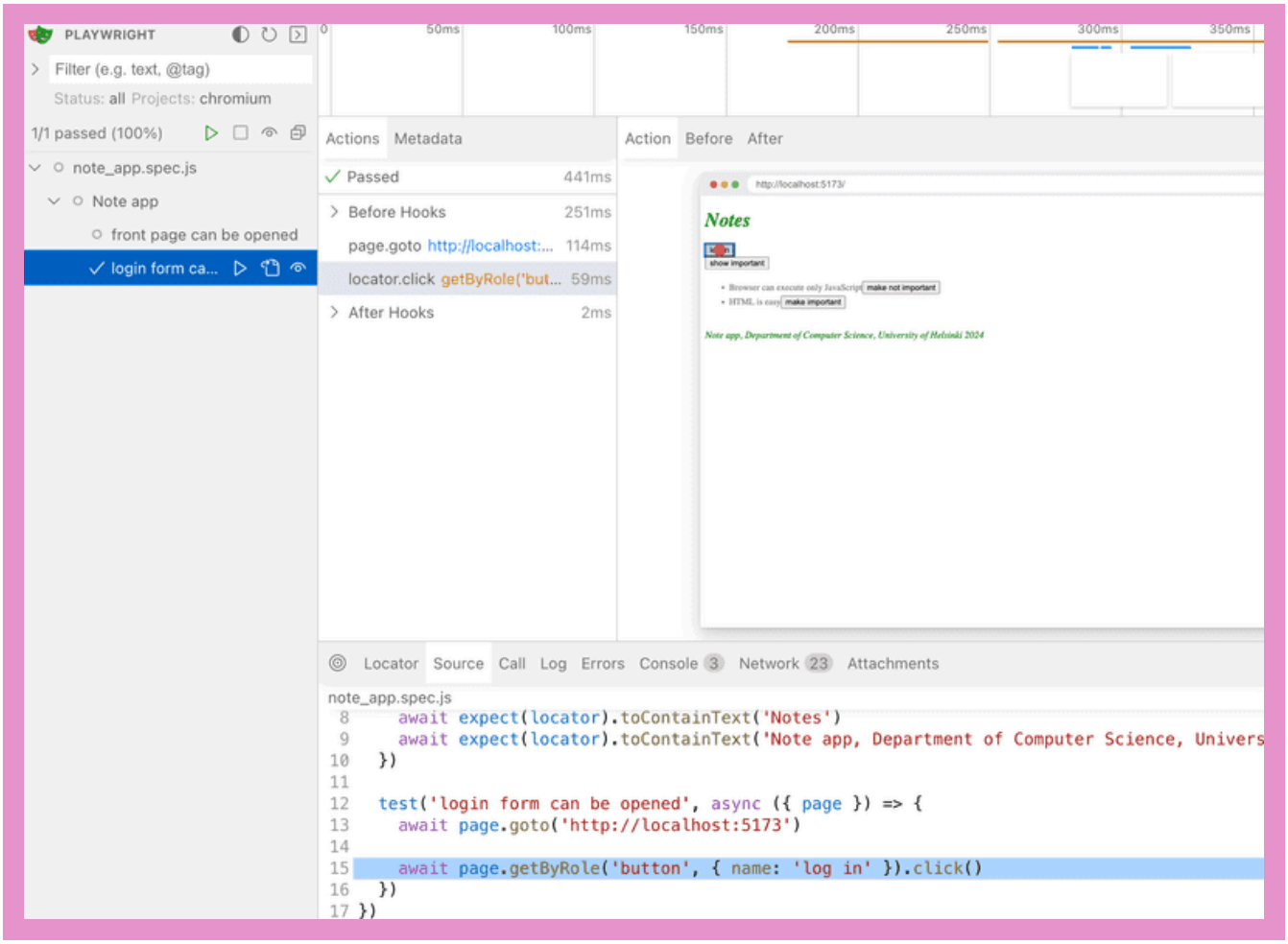
The test first uses the method page.getByRole to retrieve the button based on its text. The method returns the Locator corresponding to the Button element. Pressing the button is performed using the Locator method click.

When developing tests, you could use Playwright's UI mode, i.e. the user interface version. Let's start the tests in UI mode as follows:

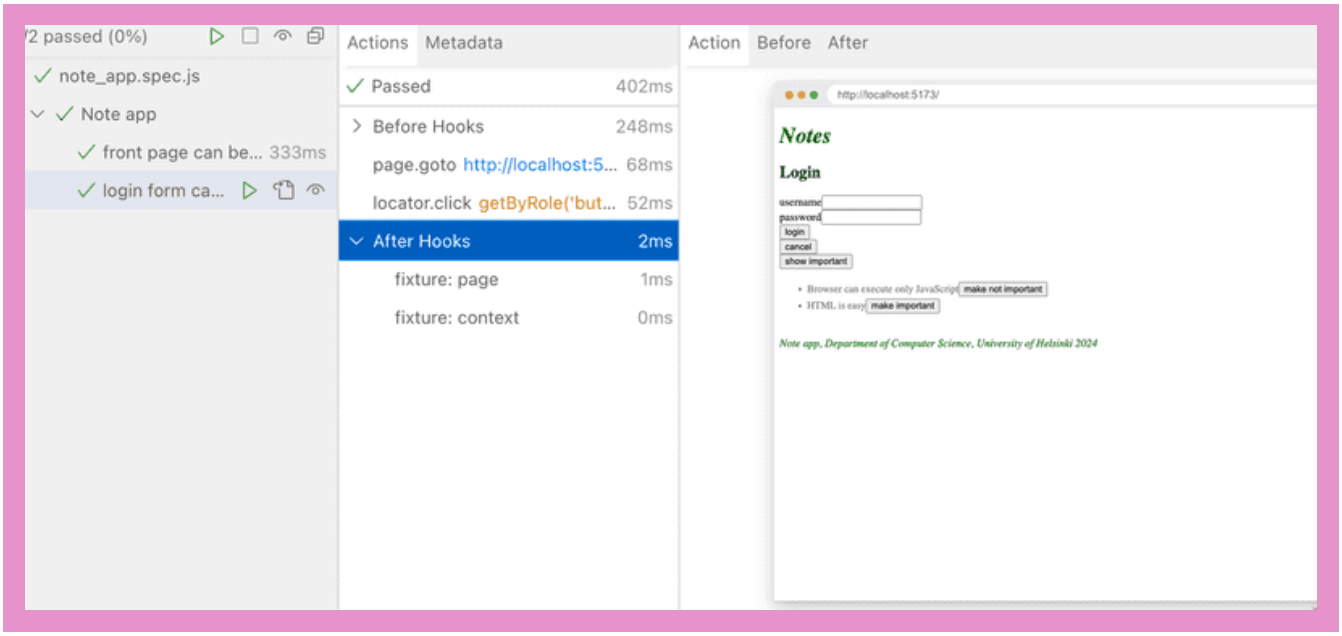
```
npm test -- --ui
```

copy

We now see that the test finds the button



After clicking, the form will appear



When the form is opened, the test should look for the text fields and enter the username and password in them. Let's make the first attempt using the method page.getByRole:

```
describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByRole('textbox').fill('mluukkai')
  })
})
```

This results to an error:

```
Error: locator.fill: Error: strict mode violation: getByRole('textbox') resolved to 2 elements:
  1) <input value=""/> aka locator('div').filter({ hasText: /^username$/ }).getByRole('textbox')
  2) <input value="" type="password"/> aka locator('input[type="password"]')
```

The problem now is that `getByRole` finds two text fields, and calling the fill method fails, because it assumes that there is only one text field found. One way around the problem is to use the methods first and last:

```
describe('Note app', () => {
  // ...
```

```
  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByRole('textbox').first().fill('mluukkai')
    await page.getByRole('textbox').last().fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

After writing in the text fields, the test presses the `login` button and checks that the application renders the logged-in user's information on the screen.

If there were more than two text fields, using the methods `first` and `last` would not be enough. One possibility would be to use the `all` method, which turns the found locators into an array that can be indexed:

copy

```
describe('Note app', () => {
  // ...
  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    const textboxes = await page.getByRole('textbox').all()
    await textboxes[0].fill('mluukkai')
    await textboxes[1].fill('salainen')

    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

Both this and the previous version of the test work. However, both are problematic to the extent that if the registration form is changed, the tests may break, as they rely on the fields to be on the page in a certain order.

A better solution is to define unique test id attributes for the fields, to search for them in the tests using the method `getByTestId`.

Let's expand the login form as follows

copy

```
const LoginForm = ({ ... }) => {
  return (
    <div>
      <h2>Login</h2>
```

```
      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            data-testid='username'
            value={username}
            onChange={handleUsernameChange}
          />
        </div>
        <div>
          password
          <input
            data-testid='password'
            type="password"
            value={password}
            onChange={handlePasswordChange}
          />
        </div>
        <button type="submit">
          login
        </button>
      </form>
    </div>
  )
}
```

Test changes as follows:

```
describe('Note app', () => {
  // ...

  test('login form can be opened', async ({ page }) => {
    await page.goto('http://localhost:5173')

    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('salainen')

    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

Note that passing the test at this stage requires that there is a user in the *test* database of the backend with username *mluukkai* and password *salainen*. Create a user if needed!

Since both tests start in the same way, i.e. by opening the page *http://localhost:5173*, it is recommended to isolate the common part in the *beforeEach* block that is executed before each test:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  beforeEach(async ({ page }) => {
    await page.goto('http://localhost:5173')
  })

  test('front page can be opened', async ({ page }) => {
    const locator = await page.getByText('Notes')
    await expect(locator).toBeVisible()
    await expect(page.getByText('Note app, Department of Computer Science, University of
Helsinki 2024')).toBeVisible()
  })

  test('login form can be opened', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()
    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })
})
```

Testing note creation

Next, let's create a test that adds a new note to the application:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  // ...

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await page.getByRole('button', { name: 'log in' }).click()
      await page.getByTestId('username').fill('mluukkai')
      await page.getByTestId('password').fill('salainen')
      await page.getByRole('button', { name: 'login' }).click()
    })

    test('a new note can be created', async ({ page }) => {
      await page.getByRole('button', { name: 'new note' }).click()
      await page.getByRole('textbox').fill('a note created by playwright')
      await page.getByRole('button', { name: 'save' }).click()
      await expect(page.getByText('a note created by playwright')).toBeVisible()
    })
  })
})
```

The test is defined in its own `describe` block. Creating a note requires that the user is logged in, which is handled in the `beforeEach` block.

The test trusts that when creating a new note, there is only one input field on the page, so it searches for it as follows:

```
page.getByRole('textbox')
```

copy

If there were more fields, the test would break. Because of this, it would be better to add a test-id to the form input and search for it in the test based on this id.

Note: the test will only pass the first time. The reason for this is that its expectation

```
await expect(page.getByText('a note created by playwright')).toBeVisible()
```

copy

causes problems when the same note is created in the application more than once. The problem will be solved in the next chapter.

The structure of the tests looks like this:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  // ....

  test('user can log in', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()
    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await page.getByRole('button', { name: 'log in' }).click()
      await page.getByTestId('username').fill('mluukkai')
      await page.getByTestId('password').fill('salainen')
      await page.getByRole('button', { name: 'login' }).click()
    })

    test('a new note can be created', async ({ page }) => {
      await page.getByRole('button', { name: 'new note' }).click()
      await page.getByRole('textbox').fill('a note created by playwright')
      await page.getByRole('button', { name: 'save' }).click()
      await expect(page.getByText('a note created by playwright')).toBeVisible()
    })
```

copy

```
  })
})
```

Since we have prevented the tests from running in parallel, Playwright runs the tests in the order they appear in the test code. That is, first the test *user can log in*, where the user logs into the application, is performed. After this the test *a new note can be created* gets executed, which also does a log in, in the *beforeEach* block. Why is this done, isn't the user already logged in thanks to the previous test? No, because the execution of *each* test starts from the browser's "zero state", all changes made to the browser's state by the previous tests are reset.

Controlling the state of the database

If the tests need to be able to modify the server's database, the situation immediately becomes more complicated. Ideally, the server's database should be the same each time we run the tests, so our tests can be reliably and easily repeatable.

As with unit and integration tests, with E2E tests it is best to empty the database and possibly format it before the tests are run. The challenge with E2E tests is that they do not have access to the database.

The solution is to create API endpoints for the backend tests. We can empty the database using these endpoints. Let's create a new router for the tests inside the *controllers* folder, in the *testing.js* file

```
const router = require('express').Router()
const Note = require('../models/note')
const User = require('../models/user')

router.post('/reset', async (request, response) => {
  await Note.deleteMany({})
  await User.deleteMany({})

  response.status(204).end()
})

module.exports = router
```

and add it to the backend only *if the application is run in test-mode*:

```
// ...

app.use('/api/login', loginRouter)
app.use('/api/users', usersRouter)
app.use('/api/notes', notesRouter)

if (process.env.NODE_ENV === 'test') {
  const testingRouter = require('./controllers/testing')
  app.use('/api/testing', testingRouter)
}
```

```
app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)

module.exports = app
```

After the changes, an HTTP POST request to the */api/testing/reset* endpoint empties the database. Make sure your backend is running in test mode by starting it with this command (previously configured in the package.json file):

```
npm run start:test
```

copy

The modified backend code can be found on the GitHub branch *part5-1*.

Next, we will change the `beforeEach` block so that it empties the server's database before tests are run.

Currently, it is not possible to add new users through the frontend's UI, so we add a new user to the backend from the beforeEach block.

```
describe('Note app', () => {
  beforeEach(async ({ page, request }) => {
    await request.post('http:localhost:3001/api/testing/reset')
    await request.post('http://localhost:3001/api/users', {
      data: {
        name: 'Matti Luukkainen',
        username: 'mluukkai',
        password: 'salainen'
      }
    })

    await page.goto('http://localhost:5173')
  })

  test('front page can be opened',  () => {
    // ...
  })

  test('user can login', () => {
    // ...
  })

  describe('when logged in', () => {
    // ...
  })
})
```

copy

During initialization, the test makes HTTP requests to the backend with the method `post` of the parameter `request`.

Unlike before, now the testing of the backend always starts from the same state, i.e. there is one user and no notes in the database.

Let's make a test that checks that the importance of the notes can be changed.

There are a few different approaches to taking the test.

In the following, we first look for a note and click on its button that has text *make not important*. After this, we check that the note contains the button with *make important*.

```
describe('Note app', () => {
  // ...

  describe('when logged in', () => {
    // ...

    describe('and a note exists', () => {
      beforeEach(async ({ page }) => {
        await page.getByRole('button', { name: 'new note' }).click()
        await page.getByRole('textbox').fill('another note by playwright')
        await page.getByRole('button', { name: 'save' }).click()
      })

      test('importance can be changed', async ({ page }) => {
        await page.getByRole('button', { name: 'make not important' }).click()
        await expect(page.getByText('make important')).toBeVisible()
      })
    })
  })
})
```

The first command first searches for the component where there is the text *another note by playwright* and inside it the button *make not important* and clicks on it.

The second command ensures that the text of the same button has changed to *make important*.

The current code for the tests is on GitHub, in branch *part5-1*.

Test for failed login

Now let's do a test that ensures that the login attempt fails if the password is wrong.

The first version of the test looks like this:

```
describe('Note app', () => {
  // ...

  test('login fails with wrong password', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
```

```
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('wrong')
    await page.getByRole('button', { name: 'login' }).click()

    await expect(page.getByText('wrong credentials')).toBeVisible()
  })

  // ...
)}
```

The test verifies with the method `page.getByText` that the application prints an error message.

The application renders the error message to an element containing the CSS class *error*:

```
const Notification = ({ message }) => {
  if (message === null) {
    return null
  }

  return (
    <div className="error">
      {message}
    </div>
  )
}
```

We could refine the test to ensure that the error message is printed exactly in the right place, i.e. in the element containing the CSS class *error*:

```
  test('login fails with wrong password', async ({ page }) => {
  // ...

  const errorDiv = await page.locator('.error')
  await expect(errorDiv).toContainText('wrong credentials')
})
```

So the test uses the `page.locator` method to find the component containing the CSS class *error* and stores it in a variable. The correctness of the text associated with the component can be verified with the expectation `toContainText`. Note that the `CSS class selector` starts with a dot, so the *error* class selector is *.error*.

It is possible to test the application's CSS styles with matcher `toHaveCSS`. We can, for example, make sure that the color of the error message is red, and that there is a border around it:

```
  test('login fails with wrong password', async ({ page }) => {
  // ...
```

```
    const errorDiv = await page.locator('.error')
    await expect(errorDiv).toContainText('wrong credentials')
    await expect(errorDiv).toHaveCSS('border-style', 'solid')
    await expect(errorDiv).toHaveCSS('color', 'rgb(255, 0, 0)')
})
```

Colors must be defined to Playwright as `rgb` codes.

Let's finalize the test so that it also ensures that the application **does not render** the text describing a successful login *'Matti Luukkainen logged in'*:

```
test('login fails with wrong password', async ({ page }) =>{
  await page.getByRole('button', { name: 'log in' }).click()
  await page.getByTestId('username').fill('mluukkai')
  await page.getByTestId('password').fill('wrong')
  await page.getByRole('button', { name: 'login' }).click()

  const errorDiv = await page.locator('.error')
  await expect(errorDiv).toContainText('wrong credentials')
  await expect(errorDiv).toHaveCSS('border-style', 'solid')
  await expect(errorDiv).toHaveCSS('color', 'rgb(255, 0, 0)')

  await expect(page.getByText('Matti Luukkainen logged in')).not.toBeVisible()
})
```

Running tests one by one

By default, Playwright always runs all tests, and as the number of tests increases, it becomes time-consuming. When developing a new test or debugging a broken one, the test can be defined instead than with the command *test*, with the command *test.only*, in which case Playwright will run only that test:

```
describe(() => {
  // this is the only test executed!
  test.only('login fails with wrong password', async ({ page }) => {
    // ...
  })

  // this test is skipped...
  test('user can login with correct credentials', async ({ page }) => {
    // ...
  }

  // ...
})
```

When the test is ready, *only* can and **should** be deleted.

Another option to run a single test is to use a command line parameter:

```
npm test -- -g "login fails with wrong password"
```

copy

Helper functions for tests

Our application tests currently look like this:

```
const { test, describe, expect, beforeEach } = require('@playwright/test')

describe('Note app', () => {
  // ...

  test('user can login with correct credentials', async ({ page }) => {
    await page.getByRole('button', { name: 'log in' }).click()
    await page.getByTestId('username').fill('mluukkai')
    await page.getByTestId('password').fill('salainen')
    await page.getByRole('button', { name: 'login' }).click()
    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })

  test('login fails with wrong password', async ({ page }) =>{
    // ...
  })

  describe('when logged in', () => {
    beforeEach(async ({ page, request }) => {
      await page.getByRole('button', { name: 'log in' }).click()
      await page.getByTestId('username').fill('mluukkai')
      await page.getByTestId('password').fill('salainen')
      await page.getByRole('button', { name: 'login' }).click()
    })

    test('a new note can be created', async ({ page }) => {
      // ...
    })

    // ...
  })
})
```

copy

First, the login function is tested. After this, another `describe` block contains a set of tests that assume that the user is logged in, the login is handled inside the initializing `beforeEach` block.

As already stated earlier, each test is executed starting from the initial state (where the database is cleared and one user is created there), so even though the test is defined after another test in the code, it does not start from the same state where the tests in the code executed earlier have left!

It is also worth striving for having non-repetitive code in tests. Let's isolate the code that handles the login as a helper function, which is placed e.g. in the file `tests/helper.js` :

```
const loginWith = async (page, username, password)  => {
  await page.getByRole('button', { name: 'log in' }).click()
  await page.getByTestId('username').fill(username)
  await page.getByTestId('password').fill(password)
  await page.getByRole('button', { name: 'login' }).click()
}

export { loginWith }
```

The test becomes simpler and clearer:

```
const { loginWith } = require('./helper')

describe('Note app', () => {
  test('user can log in', async ({ page }) => {
    await loginWith(page, 'mluukkai', 'salainen')
    await expect(page.getByText('Matti Luukkainen logged in')).toBeVisible()
  })

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await loginWith(page, 'mluukkai', 'salainen')
    })

  test('a new note can be created', () => {
    // ...
  })

  // ...
})
```

Playwright also offers a solution where the login is performed once before the tests, and each test starts from a state where the application is already logged in. In order for us to take advantage of this method, the initialization of the application's test data should be done a bit differently than now. In the current solution, the database is reset before each test, and because of this, logging in just once before the tests is impossible. In order for us to use the pre-test login provided by Playwright, the user should be initialized only once before the tests. We stick to our current solution for the sake of simplicity.

The corresponding repeating code actually also applies to creating a new note. For that, there is a test that creates a note using a form. Also in the `beforeEach` initialization block of the test that tests changing the importance of the note, a note is created using the form:

```
describe('Note app', function() {
  // ...

  describe('when logged in', () => {
    test('a new note can be created', async ({ page }) => {
```

```
      await page.getByRole('button', { name: 'new note' }).click()
      await page.getByRole('textbox').fill('a note created by playwright')
      await page.getByRole('button', { name: 'save' }).click()
      await expect(page.getByText('a note created by playwright')).toBeVisible()
    })

    describe('and a note exists', () => {
      beforeEach(async ({ page }) => {
        await page.getByRole('button', { name: 'new note' }).click()
        await page.getByRole('textbox').fill('another note by playwright')
        await page.getByRole('button', { name: 'save' }).click()
      })

      test('it can be made important', async ({ page }) => {
        // ...
      })
    })
  })
})
```

Creation of a note is also isolated as its helper function. The file `tests/helper.js` expands as follows:

```
const loginWith = async (page, username, password)  => {
  await page.getByRole('button', { name: 'log in' }).click()
  await page.getByTestId('username').fill(username)
  await page.getByTestId('password').fill(password)
  await page.getByRole('button', { name: 'login' }).click()
}

const createNote = async (page, content) => {
  await page.getByRole('button', { name: 'new note' }).click()
  await page.getByRole('textbox').fill(content)
  await page.getByRole('button', { name: 'save' }).click()
}

export { loginWith, createNote }
```

The tests are simplified as follows:

```
describe('Note app', () => {
  // ...

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      await loginWith(page, 'mluukkai', 'salainen')
    })

    test('a new note can be created', async ({ page }) => {
      await createNote(page, 'a note created by playwright', true)
```

```
      await expect(page.getByText('a note created by playwright')).toBeVisible()
    })

    describe('and a note exists', () => {
      beforeEach(async ({ page }) => {
        await createNote(page, 'another note by playwright', true)
      })

      test('importance can be changed', async ({ page }) => {
        await page.getByRole('button', { name: 'make not important' }).click()
        await expect(page.getByText('make important')).toBeVisible()
      })
    })
  })
})
```

There is one more annoying feature in our tests. The frontend address *http:localhost:5173* and the backend address *http:localhost:3001* are hardcoded for tests. Of these, the address of the backend is actually useless, because a proxy has been defined in the Vite configuration of the frontend, which forwards all requests made by the frontend to the address *http:localhost:5173/api* to the backend:

```
export default defineConfig({
  server: {
    proxy: {
      '/api': {
        target: 'http://localhost:3001',
        changeOrigin: true,
      },
    }
  },
  // ...
})
```

So we can replace all the addresses in the tests from `http://localhost:3001/api/...` to `http://localhost:5173/api/...`.

We can now define the `baseUrl` for the application in the tests configuration file *playwright.config.js*:

```
module.exports = defineConfig({
  // ...
  use: {
    baseURL: 'http://localhost:5173',
  },
  // ...
}
```

All the commands in the tests that use the application url, e.g.

```
await page.goto('http://localhost:5173')
await page.post('http://localhost:5173/api/tests/reset')
```

can be transformed into:

```
await page.goto('/')
await page.post('/api/tests/reset')
```

The current code for the tests is on GitHub, branch *part5-2*.

Note importance change revisited

Let's take a look at the test we did earlier, which verifies that it is possible to change the importance of a note.

Let's change the initialization block of the test so that it creates two notes instead of one:

```
describe('when logged in', () => {
  // ...
  describe('and several notes exists', () => {
    beforeEach(async ({ page }) => {
      await createNote(page, 'first note', true)
      await createNote(page, 'second note', true)
    })

    test('one of those can be made nonimportant', async ({ page }) => {
      const otherNoteElement = await page.getByText('first note')

      await otherNoteElement
        .getByRole('button', { name: 'make not important' }).click()
      await expect(otherNoteElement.getByText('make important')).toBeVisible()
    })
  })
})
```

The test first searches for the element corresponding to the first created note using the method `page.getByText` and stores it in a variable. After this, a button with the text `make not important` is searched inside the element and the button is pressed. Finally, the test verifies that the button's text has changed to `make important`.

The test could also have been written without the auxiliary variable:

```
test('one of those can be made nonimportant', async ({ page }) => {
  await page.getByText('first note')
    .getByRole('button', { name: 'make not important' }).click()

  await expect(page.getByText('first note').getByText('make important'))
    .toBeVisible()
})
```

Let's change the `Note` component so that the note text is rendered inside a `span` element

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li className='note'>
      <span>{note.content}</span>
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Tests break! The reason for the problem is that the command `await page.getByText('second note')` now returns a `span` element containing only text, and the button is outside of it.

One way to fix the problem is as follows:

```
test('one of those can be made nonimportant', async ({ page }) => {
  const otherNoteText = await page.getByText('first note')
  const otherNoteElement = await otherNoteText.locator('..')

  await otherNoteElement.getByRole('button', { name: 'make not important' }).click()
  await expect(otherNoteElement.getByText('make important')).toBeVisible()
})
```

The first line now looks for the `span` element containing the text associated with the first created note. In the second line, the function `locator` is used and `..` is given as an argument, which retrieves the element's parent element. The locator function is very flexible, and we take advantage of the fact that accepts as argument not only CSS selectors but also XPath selector. It would be possible to express the same with CSS, but in this case XPath provides the simplest way to find the parent of an element.

Of course, the test can also be written using only one auxiliary variable:

```
test('one of those can be made nonimportant', async ({ page }) => {
  const secondNoteElement = await page.getByText('second note').locator('..')
  await secondNoteElement.getByRole('button', { name: 'make not important' }).click()
  await expect(secondNoteElement.getByText('make important')).toBeVisible()
})
```

Let's change the test so that three notes are created, the importance is changed in the second created note:

```
describe('when logged in', () => {
  beforeEach(async ({ page }) => {
    await loginWith(page, 'mluukkai', 'salainen')
  })

  test('a new note can be created', async ({ page }) => {
    await createNote(page, 'a note created by playwright', true)
    await expect(page.getByText('a note created by playwright')).toBeVisible()
  })

  describe('and a note exists', () => {
    beforeEach(async ({ page }) => {
      await createNote(page, 'first note', true)
      await createNote(page, 'second note', true)
      await createNote(page, 'third note', true)
    })

    test('importance can be changed', async ({ page }) => {
      const otherNoteText = await page.getByText('second note')
      const otherdNoteElement = await otherNoteText.locator('..')

      await otherdNoteElement.getByRole('button', { name: 'make not important' }).click()
      await expect(otherdNoteElement.getByText('make important')).toBeVisible()
    })
  })
})
```

For some reason the test starts working unreliably, sometimes it passes and sometimes it doesn't. It's time to roll up your sleeves and learn how to debug tests.

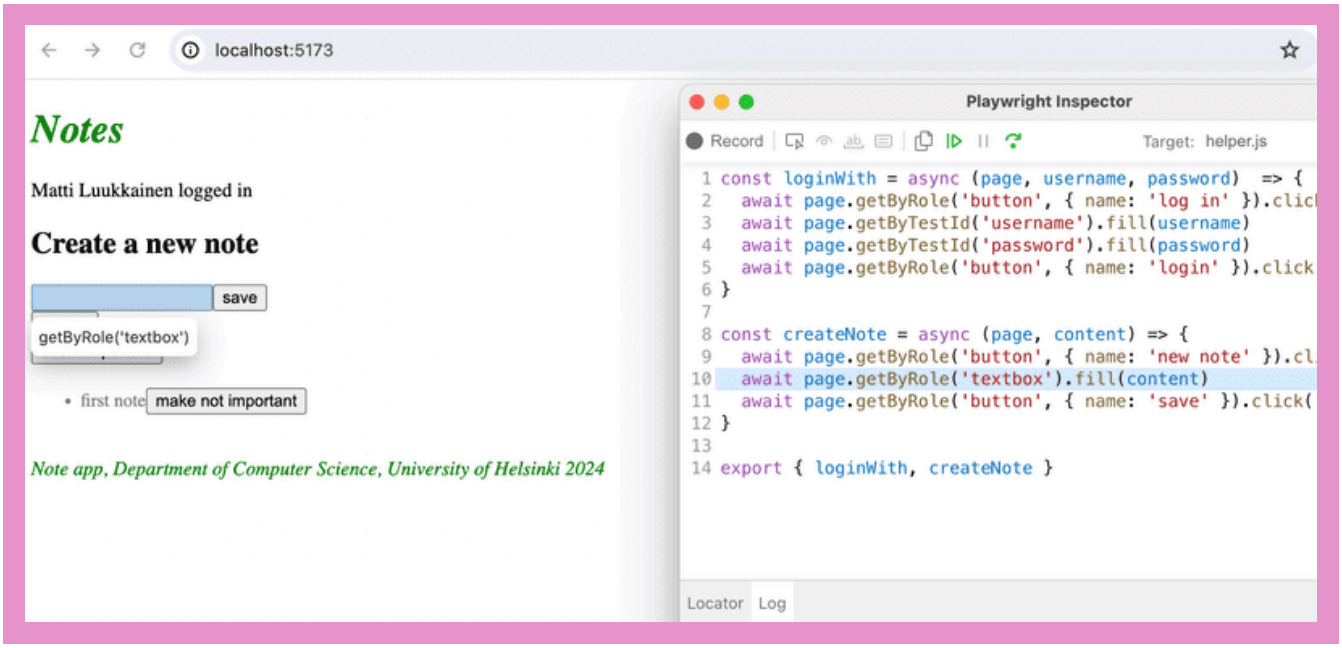
Test development and debugging

If, and when the tests don't pass and you suspect that the fault is in the tests instead of in the code, you should run the tests in debug mode.

The following command runs the problematic test in debug mode:

```
npm test -- -g'importance can be changed' --debug
```

Playwright-inspector shows the progress of the tests step by step. The arrow-dot button at the top takes the tests one step further. The elements found by the locators and the interaction with the browser are visualized in the browser:



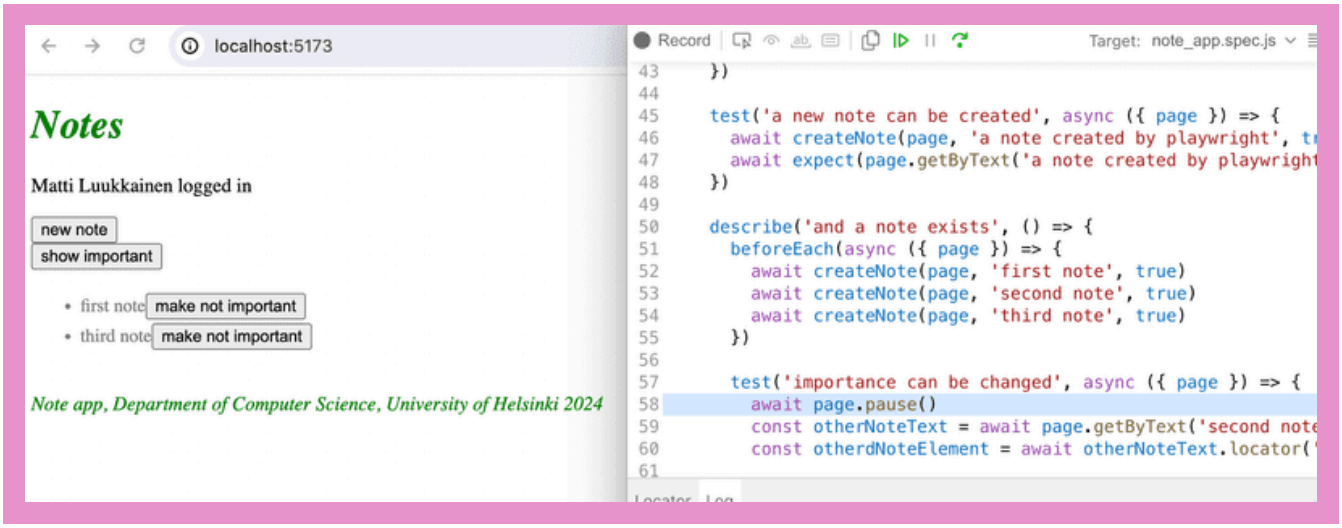
By default, debug steps through the test command by command. If it is a complex test, it can be quite a burden to step through the test to the point of interest. This can be avoided by using the command `await page.pause()`:

```
describe('Note app', () => {
  beforeEach(async ({ page, request }) => {
    // ...
  }

  describe('when logged in', () => {
    beforeEach(async ({ page }) => {
      // ...
    })

    describe('and several notes exists', () => {
      beforeEach(async ({ page }) => {
        await createNote(page, 'first note')
        await createNote(page, 'second note')
        await createNote(page, 'third note')
      })

      test('one of those can be made nonimportant', async ({ page }) => {
        await page.pause()
        const otherNoteText = await page.getByText('second note')
        const otherdNoteElement = await otherNoteText.locator('..')
```

```
        await otherdNoteElement.getByRole('button', { name: 'make not important'
}).click()
        await expect(otherdNoteElement.getByText('make important')).toBeVisible()
      })
    })
  })
})
```

Now in the test you can go to `page.pause()` in one step, by pressing the green arrow symbol in the inspector.

When we now run the test and jump to the `page.pause()` command, we find an interesting fact:



It seems that the browser *does not render* all the notes created in the block `beforeEach`. What is the problem?

The reason for the problem is that when the test creates one note, it starts creating the next one even before the server has responded, and the added note is rendered on the screen. This in turn can cause some notes to be lost (in the picture, this happened to the second note created), since the browser is re-rendered when the server responds, based on the state of the notes at the start of that insert operation.

The problem can be solved by "slowing down" the insert operations by using the waitFor command after the insert to wait for the inserted note to render:

```
const createNote = async (page, content) => {
  await page.getByRole('button', { name: 'new note' }).click()
  await page.getByRole('textbox').fill(content)
  await page.getByRole('button', { name: 'save' }).click()
  await page.getByText(content).waitFor()
}
```

Instead of, or alongside debugging mode, running tests in UI mode can be useful. As already mentioned, tests are started in UI mode as follows:

```
npm run test -- --ui
```

copy

Almost the same as UI mode is use of the Playwright's Trace Viewer. The idea is that a "visual trace" of the tests is saved, which can be viewed if necessary after the tests have been completed. A trace is saved by running the tests as follows:

```
npm run test -- --trace on
```

copy

If necessary, Trace can be viewed with the command

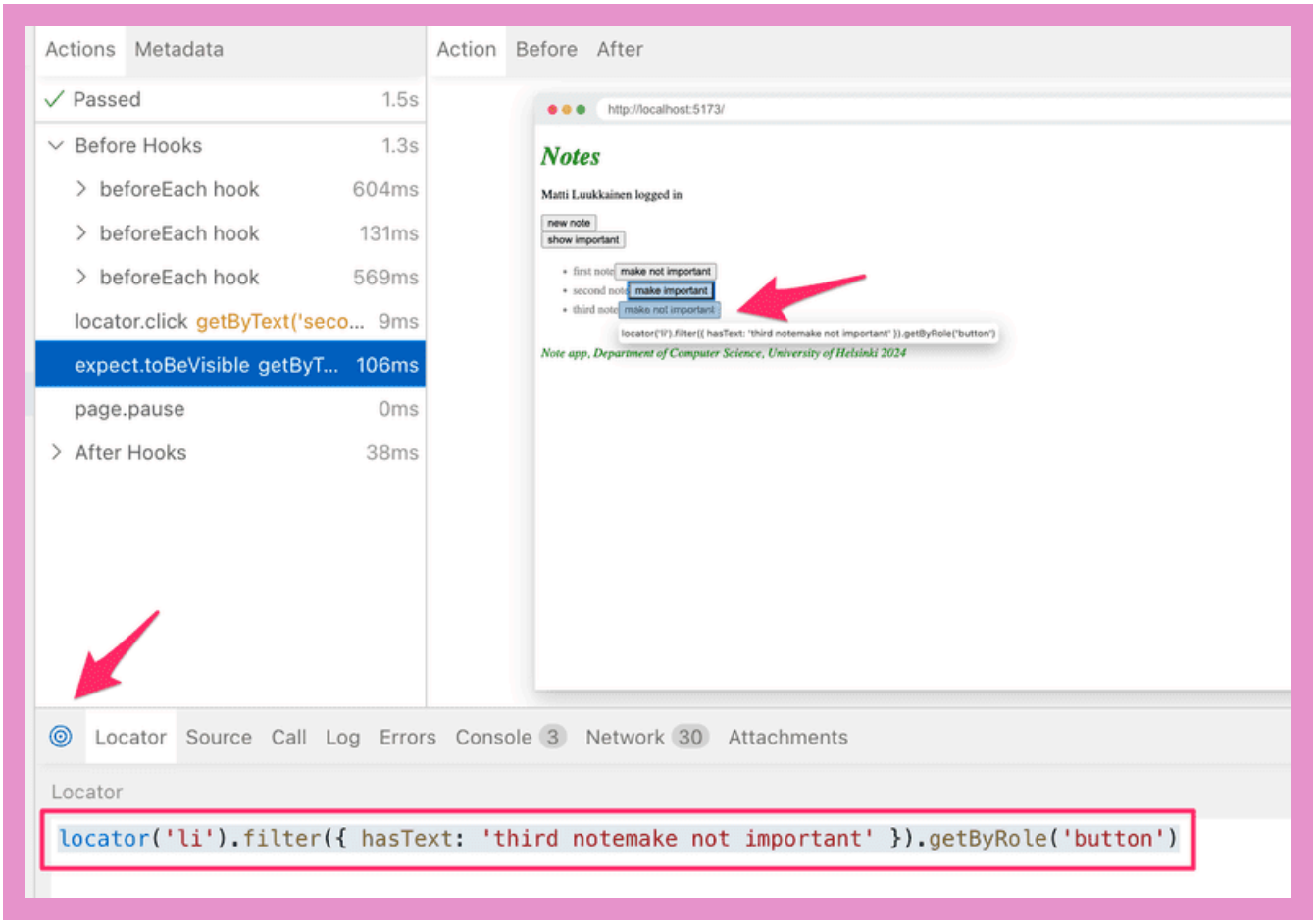
```
npx playwright show-report
```

copy

or with the npm script we defined `npm run test:report`

Trace looks practically the same as running tests in UI mode.

UI mode and Trace Viewer also offer the possibility of assisted search for locators. This is done by pressing the double circle on the left side of the lower bar, and then by clicking on the desired user interface element. Playwright displays the element locator:



Playwright suggests the following as the locator for the third note

```
page.locator('li').filter({ hasText: 'third note' }).getByRole('button')
```

The method page.locator is called with the argument `li` , i.e. we search for all li elements on the page, of which there are three in total. After this, using the locator.filter method, we narrow down to the li element that contains the text *third note* and the button element inside it is taken using the locator.getByRole method.

The locator generated by Playwright is somewhat different from the locator used by our tests, which was

```
page.getByText('first note').locator('..').getByRole('button', { name: 'make not important' })
```

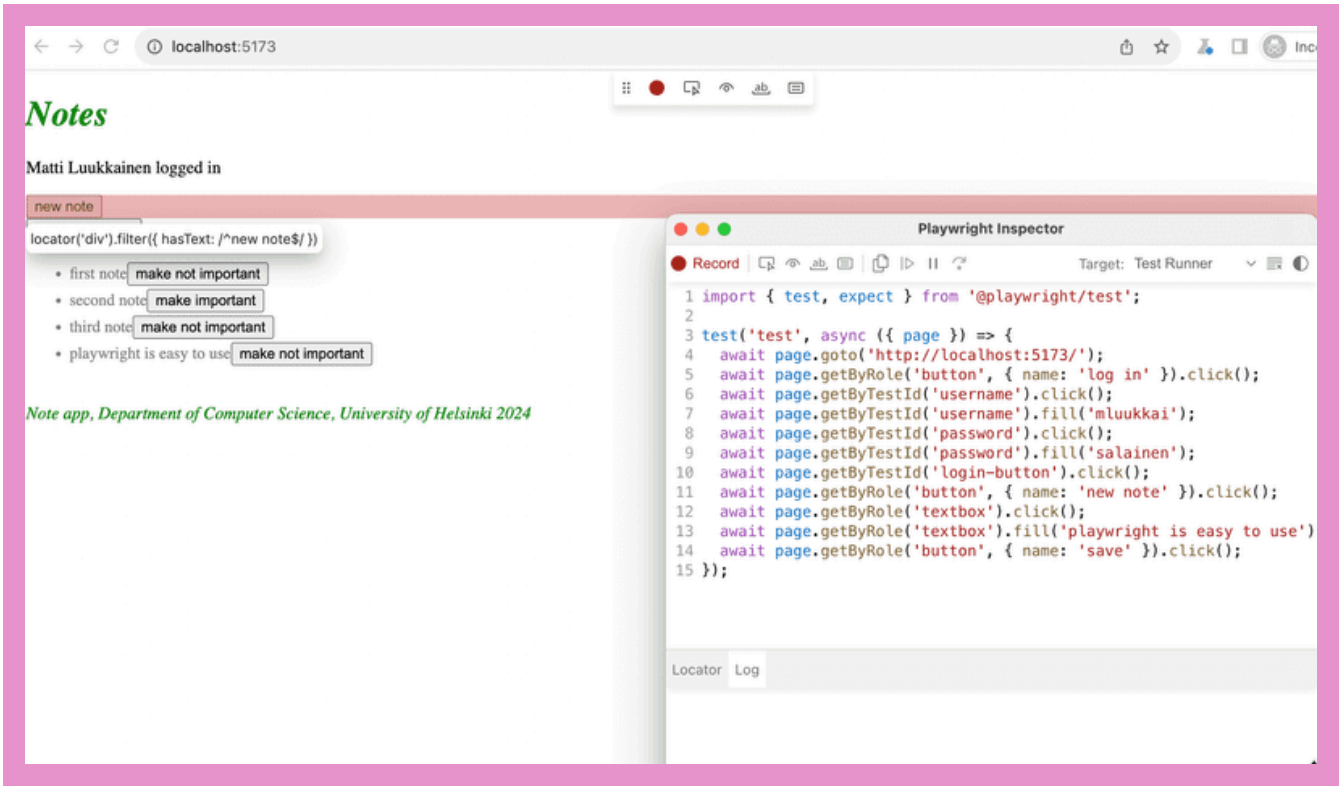
Which of the locators is better is probably a matter of taste.

Playwright also includes a test generator that makes it possible to "record" a test through the user interface. The test generator is started with the command:

```
npx playwright codegen http://localhost:5173/
```

copy

When the `Record` mode is on, the test generator "records" the user's interaction in the Playwright inspector, from where it is possible to copy the locators and actions to the tests:



Instead of the command line, Playwright can also be used via the VS Code plugin. The plugin offers many convenient features, e.g. use of breakpoints when debugging tests.

To avoid problem situations and increase understanding, it is definitely worth browsing Playwright's high-quality documentation. The most important sections are listed below:

- the section about locators gives good hints for finding elements in test
- section actions tells how it is possible to simulate the interaction with the browser in tests
- the section about assertions demonstrates the different expectations Playwright offers for testing

In-depth details can be found in the API description, particularly useful are the class Page corresponding to the browser window of the application under test, and the class Locator corresponding to the elements searched for in the tests.

The final version of the tests is in full on GitHub, in branch *part5-3*.

The final version of the frontend code is in its entirety on GitHub, in branch *part5-9*.

Exercises 5.17.-5.23.

In the last exercises of this part, let's do some E2E tests for the blog application. The material above should be enough to do most of the exercises. However, you should definitely read Playwright's documentation and API description, at least the sections mentioned at the end of the previous chapter.

5.17: Blog List End To End Testing, step 1

Create a new npm project for tests and configure Playwright there.

Make a test to ensure that the application displays the login form by default.

The body of the test should be as follows:

```
const { test, expect, beforeEach, describe } = require('@playwright/test')

describe('Blog app', () => {
  beforeEach(async ({ page }) => {
    await page.goto('http://localhost:5173')
  })

  test('Login form is shown', async ({ page }) => {
    // ...
  })
})
```

5.18: Blog List End To End Testing, step 2

Do the tests for login. Test both successful and failed login. For tests, create a user in the `beforeEach` block.

The body of the tests expands as follows

```
const { test, expect, beforeEach, describe } = require('@playwright/test')

describe('Blog app', () => {
  beforeEach(async ({ page, request }) => {
    // empty the db here
    // create a user for the backend here
    // ...
  })

  test('Login form is shown', async ({ page }) => {
    // ...
  })
```

```
  describe('Login', () => {
    test('succeeds with correct credentials', async ({ page }) => {
      // ...
    })

    test('fails with wrong credentials', async ({ page }) => {
      // ...
    })
  })
})
```

The `beforeEach` block must empty the database using, for example, the reset method we used in the material.

5.19: Blog List End To End Testing, step 3

Create a test that verifies that a logged in user can create a blog. The body of the test may look like the following

```
describe('When logged in', () => {
  beforeEach(async ({ page }) => {
    // ...
  })

  test('a new blog can be created', async ({ page }) => {
    // ...
  })
})
```

The test should ensure that the created blog is visible in the list of blogs.

5.20: Blog List End To End Testing, step 4

Do a test that makes sure the blog can be edited.

5.21: Blog List End To End Testing, step 5

Make a test that ensures that the user who added the blog can delete the blog. If you use the `window.confirm` dialog in the delete operation, you may have to Google how to use the dialog in the Playwright tests.

5.22: Blog List End To End Testing, step 6

Make a test that ensures that only the user who added the blog sees the blog's delete button.

5.23: Blog List End To End Testing, step 7

Do a test that ensures that the blogs are arranged in the order according to the likes, the blog with the most likes first.

This task is significantly more challenging than the previous ones.

This was the last task of the section and it's time to push the code to GitHub and mark the completed tasks in the exercise submission system.

Propose changes to material

Part 5c
Previous part

Part 5e
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON

{() => fs}

Fullstack > Part 5 > End to end testing: Cypress

e End to end testing: Cypress

Cypress has been the most popular E2E testing library for the past few years, but Playwright is rapidly gaining ground. This course has been using Cypress for years. Now Playwright is a new addition. You can choose whether to complete the E2E testing part of the course with Cypress or Playwright. The operating principles of both libraries are very similar, so your choice is not very important. However, Playwright is now the preferred E2E library for the course.

If your choice is Cypress, please proceed. If you end up using Playwright, go here.

Cypress

E2E library Cypress has become popular within the last years. Cypress is exceptionally easy to use, and when compared to Selenium, for example, it requires a lot less hassle and headache. Its operating principle is radically different than most E2E testing libraries because Cypress tests are run completely within the browser. Other libraries run the tests in a Node process, which is connected to the browser through an API.

Let's make some end-to-end tests for our note application.

Unlike the backend tests or unit tests done on the React front-end, End to End tests do not need to be located in the same npm project where the code is. Let's make a completely separate project for the E2E tests with the `npm init` command. Then install Cypress to *the new project* as a development dependency

```
npm install --save-dev cypress
```

copy

and by adding an npm-script to run it:

```
{
  // ...
  "scripts": {
    "cypress:open": "cypress open"
  },
  // ...
}
```

We also made a small change to the script that starts the application, without the change Cypress can not access the app.

Unlike the frontend's unit tests, Cypress tests can be in the frontend or the backend repository, or even in their separate repository.

The tests require that the system being tested is running. Unlike our backend integration tests, Cypress tests *do not start* the system when they are run.

Let's add an npm script to *the backend* which starts it in test mode, or so that *NODE_ENV* is *test*.

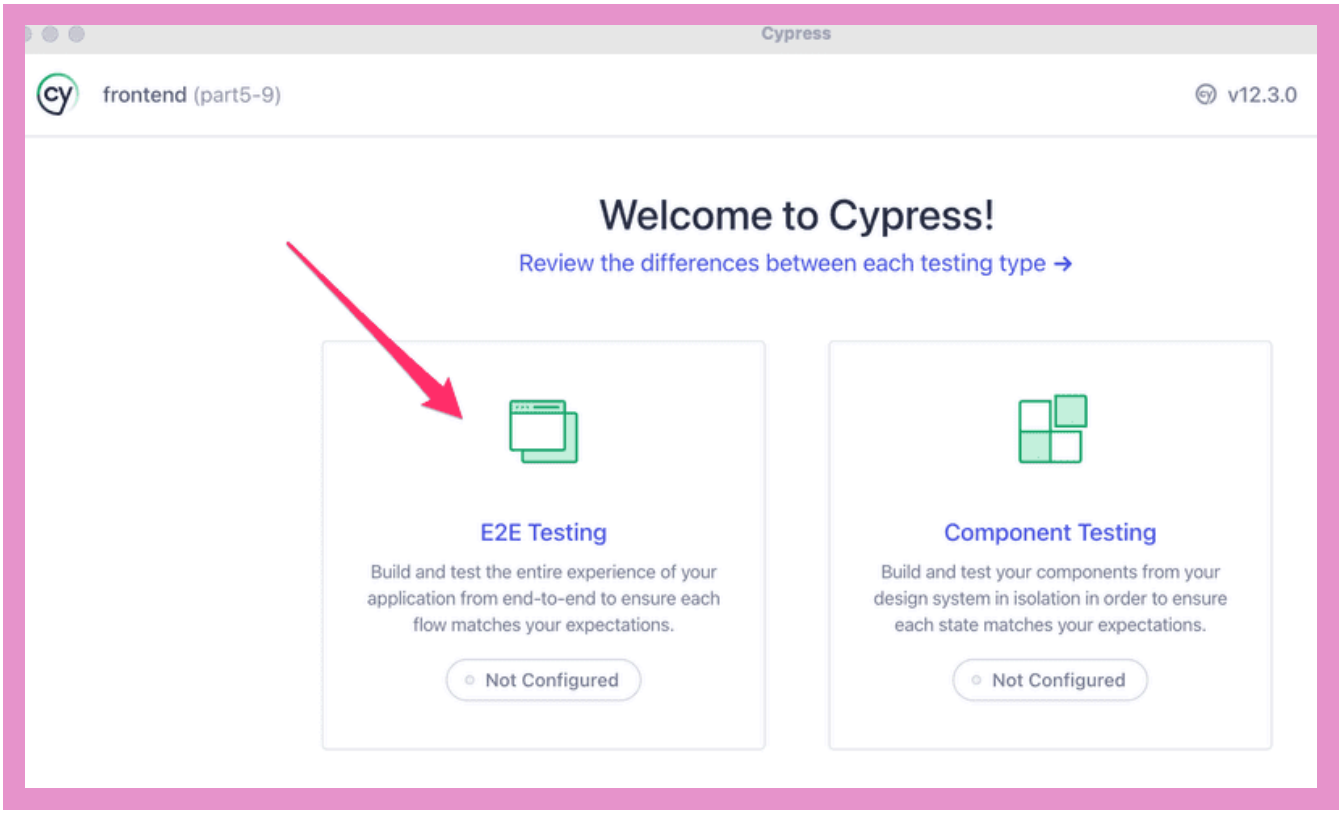
```
{
  // ...
  "scripts": {
    "start": "NODE_ENV=production node index.js",
    "dev": "NODE_ENV=development nodemon index.js",
    "build:ui": "rm -rf build && cd ../frontend/ && npm run build && cp -r build
../backend",
    "deploy": "fly deploy",
    "deploy:full": "npm run build:ui && npm run deploy",
    "logs:prod": "fly logs",
    "lint": "eslint .",
    "test": "jest --verbose --runInBand",
    "start:test": "NODE_ENV=test node index.js"
  },
  // ...
}
```

NB To get Cypress working with WSL2 one might need to do some additional configuring first. These two links are great places to start.

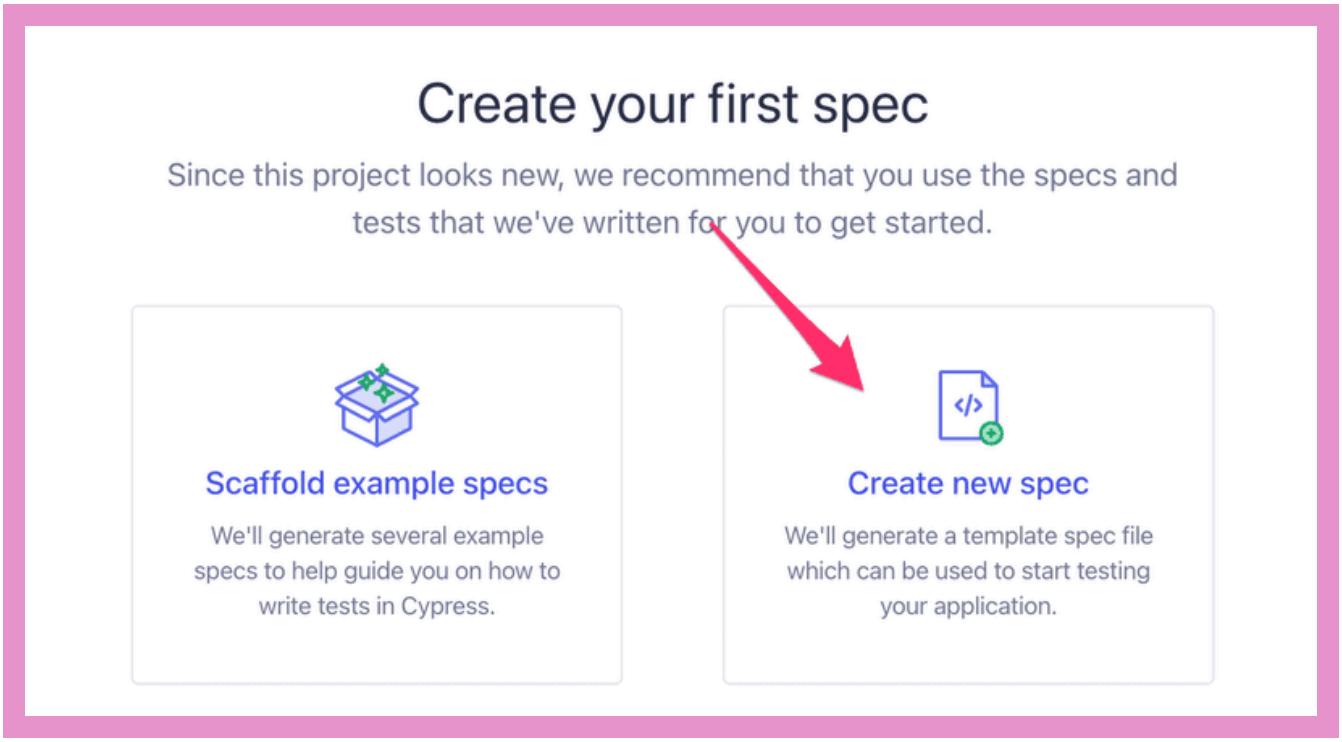
When both the backend and frontend are running, we can start Cypress with the command

```
npm run cypress:open
```

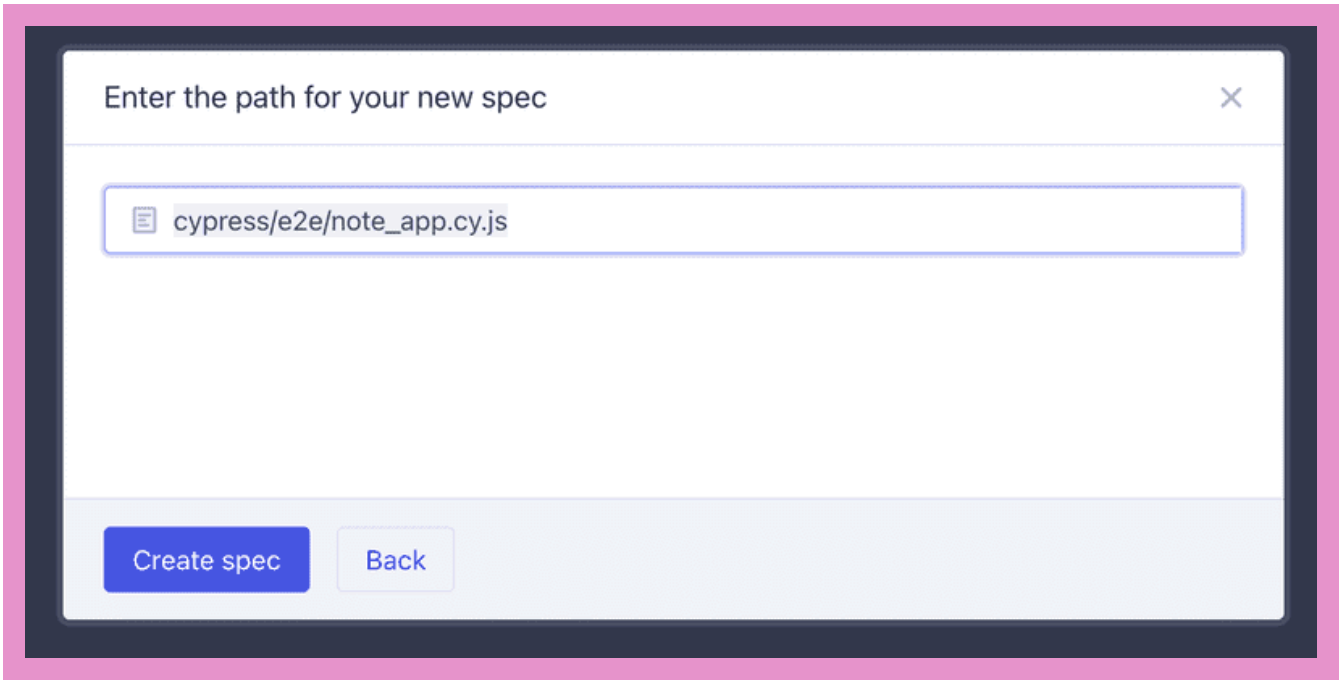
Cypress asks what type of tests we are doing. Let us answer "E2E Testing":



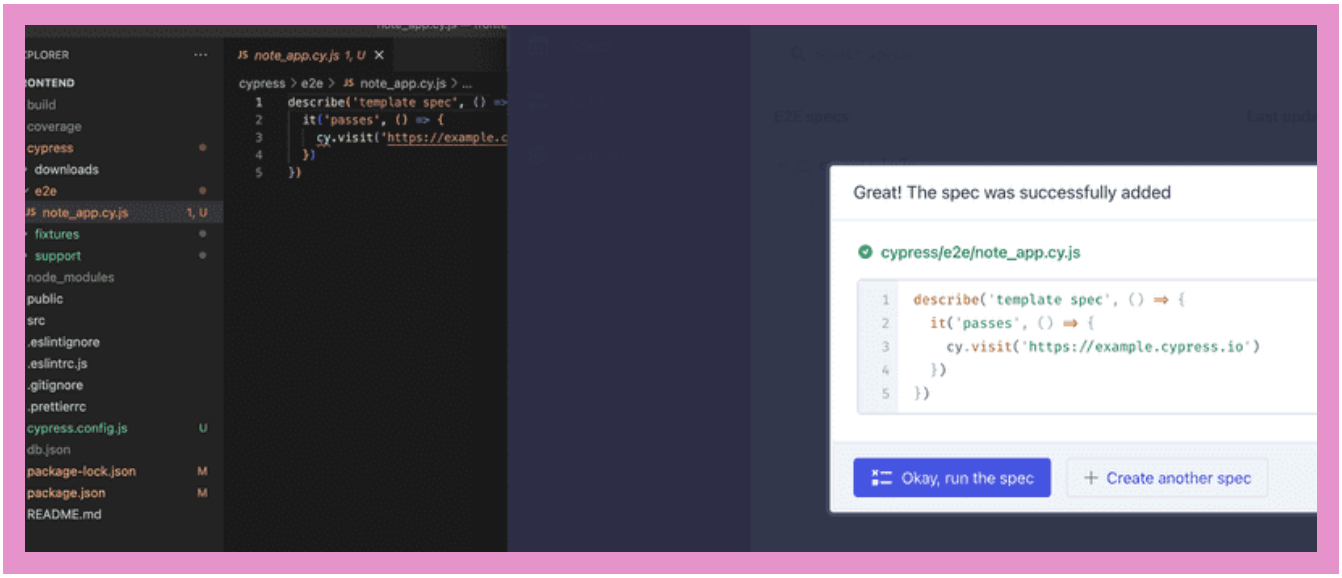
Next a browser is selected (e.g. Chrome) and then we click "Create new spec":



Let us create the test file *cypress/e2e/note_app.cy.js*:



We could edit the tests in Cypress but let us rather use VS Code:



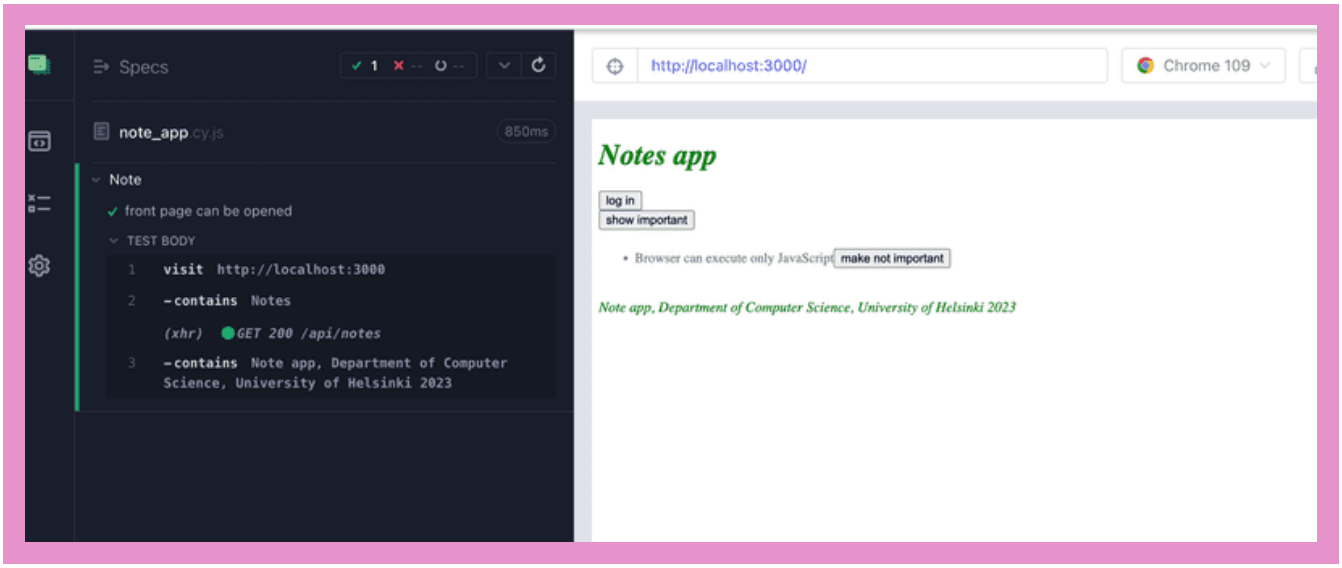
We can now close the edit view of Cypress.

Let us change the test content as follows:

```
describe('Note app', function() {
  it('front page can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })
})
```

The test is run by clicking on the test in Cypress:

Running the test shows how the application behaves as the test is run:



The structure of the test should look familiar. They use *describe* blocks to group different test cases, just like Jest. The test cases have been defined with the *it* method. Cypress borrowed these parts from the Mocha testing library that it uses under the hood.

cy.visit and cy.contains are Cypress commands, and their purpose is quite obvious. cy.visit opens the web address given to it as a parameter in the browser used by the test. cy.contains searches for the string it received as a parameter in the page.

We could have declared the test using an arrow function

```
describe('Note app', () => {
  it('front page can be opened', () => {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })
})
```

copy

However, Mocha recommends that arrow functions are not used, because they might cause some issues in certain situations.

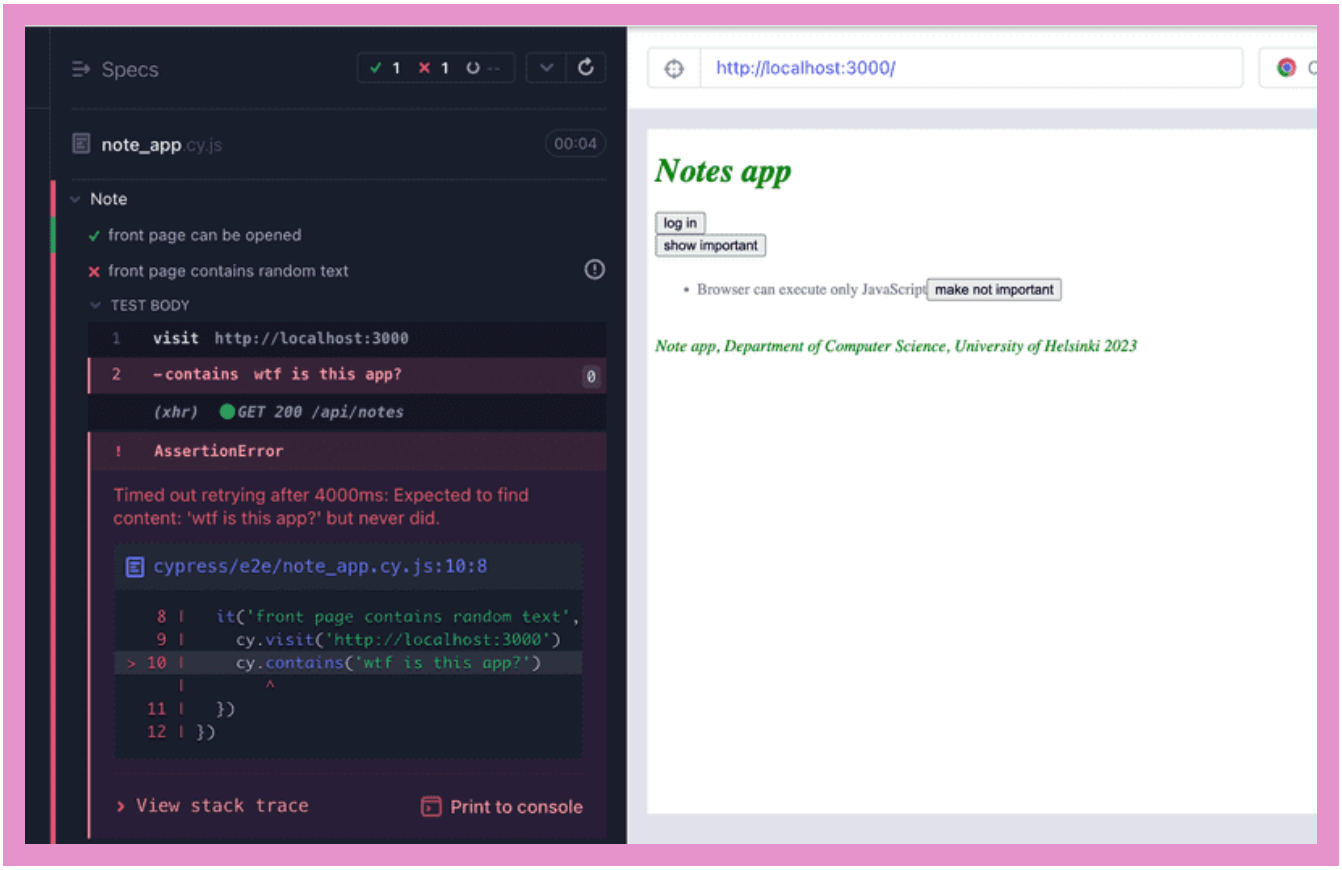
If *cy.contains* does not find the text it is searching for, the test does not pass. So if we extend our test like so

```
describe('Note app', function() {
  it('front page can be opened',  function() {
    cy.visit('http://localhost:5173')
    cy.contains('Notes')
```

copy

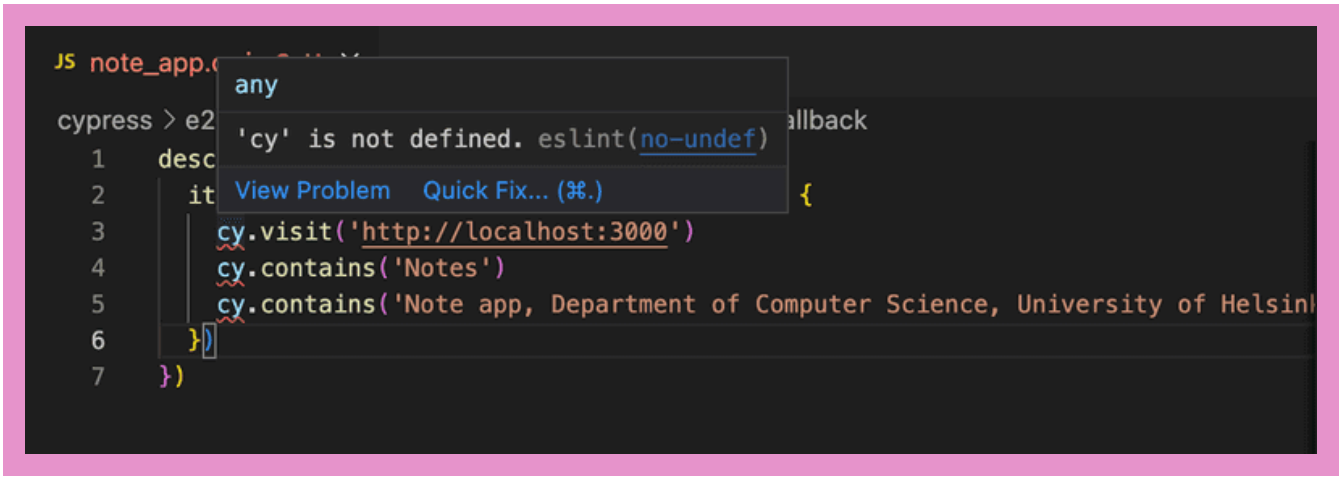
```
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })

  it('front page contains random text', function() {
    cy.visit('http://localhost:5173')
    cy.contains('wtf is this app?')
  })
})
```

the test fails



Let's remove the failing code from the test.

The variable `cy` our tests use gives us a nasty Eslint error



We can get rid of it by installing eslint-plugin-cypress as a development dependency

```
npm install eslint-plugin-cypress --save-dev
```

copy

and changing the configuration in *.eslintrc.cjs* like so:

```
module.exports = {
  "env": {
    browser: true,
    es2020: true,
    "jest/globals": true,
    "cypress/globals": true
  },
  "extends": [
    // ...
  ],
  "parserOptions": {
    // ...
  },
  "plugins": [
      "react", "jest", "cypress"
  ],
  "rules": {
    // ...
  }
}
```

copy

Writing to a form

Let's extend our tests so that our new test tries to log in to our application. We assume our backend contains a user with the username *mluukkai* and password *salainen*.

The test begins by opening the login form.

```
describe('Note app',  function() {
  // ...

  it('login form can be opened', function() {
    cy.visit('http://localhost:5173')
    cy.contains('log in').click()
  })
})
```

The test first searches for the login button by its text and clicks the button with the command `cy.click`.

Both of our tests begin the same way, by opening the page *http://localhost:5173*, so we should extract the shared code into a *beforeEach* block run before each test:

```
describe('Note app', function() {
  beforeEach(function() {
    cy.visit('http://localhost:5173')
  })

  it('front page can be opened', function() {
    cy.contains('Notes')
    cy.contains('Note app, Department of Computer Science, University of Helsinki 2023')
  })

  it('login form can be opened', function() {
    cy.contains('log in').click()
  })
})
```

The login field contains two *input* fields, which the test should write into.

The `cy.get` command allows for searching elements by CSS selectors.

We can access the first and the last input field on the page, and write to them with the command `cy.type` like so:

```
it('user can login', function () {
  cy.contains('log in').click()
  cy.get('input:first').type('mluukkai')
  cy.get('input:last').type('salainen')
})
```

The test works. The problem is if we later add more input fields, the test will break because it expects the fields it needs to be the first and the last on the page.

It would be better to give our inputs unique *IDs* and use those to find them. We change our login form like so:

```
const LoginForm = ({ ... }) => {
  return (
    <div>
      <h2>Login</h2>
      <form onSubmit={handleSubmit}>
        <div>
          username
          <input
            id='username'
            value={username}
            onChange={handleUsernameChange}
          />
        </div>
        <div>
          password
          <input
            id='password'
            type="password"
            value={password}
            onChange={handlePasswordChange}
          />
        </div>
        <button id="login-button" type="submit">
          login
        </button>
      </form>
    </div>
  )
}
```

We also added an ID to our submit button so we can access it in our tests.

The test becomes:

```
describe('Note app',  function() {
  // ..
  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })
})
```

The last row ensures that the login was successful.

Note that the CSS's ID selector is #, so if we want to search for an element with the ID *username* the CSS selector is *#username*.

Please note that passing the test at this stage requires that there is a user in the test database of the backend test environment, whose username is *mluukkai* and the password is *salainen*. Create a user if needed!

Testing new note form

Next, let's add tests to test the "new note" functionality:

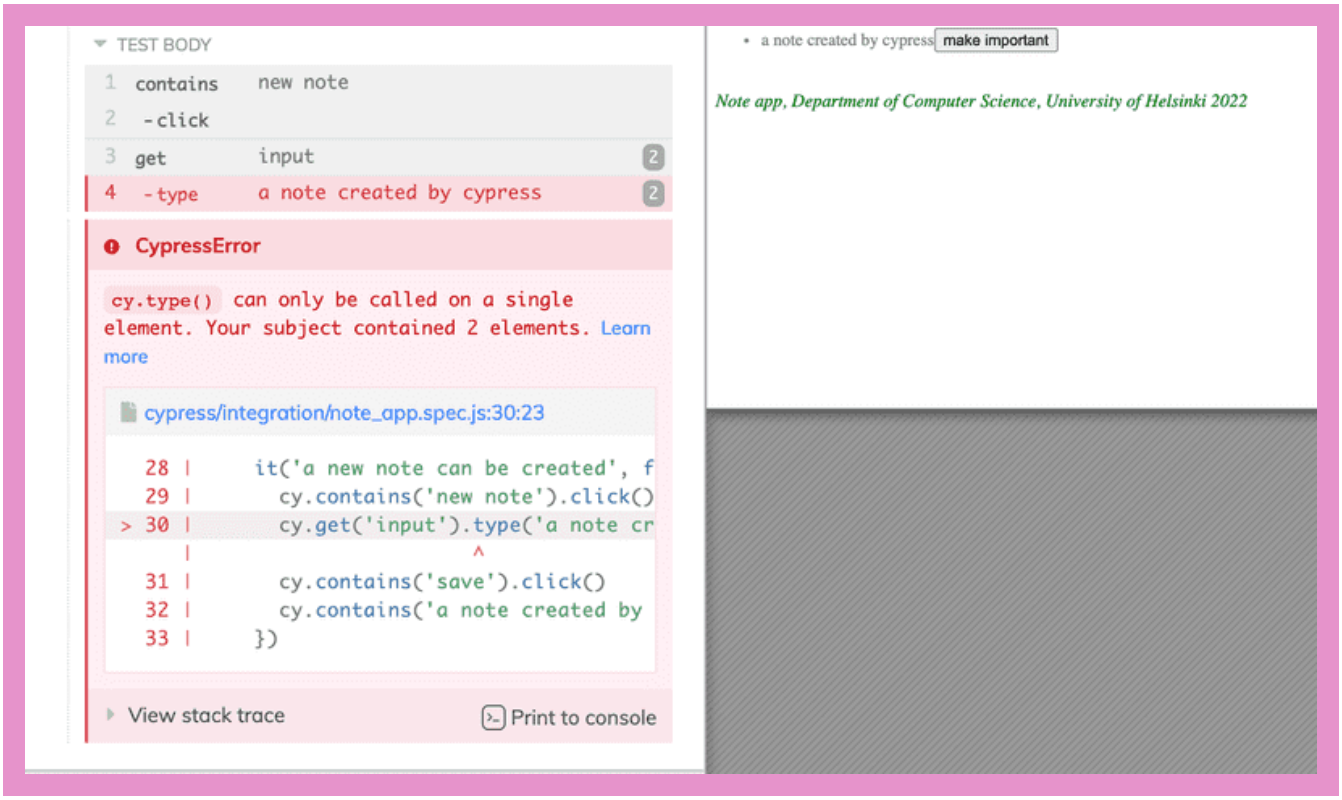
```
describe('Note app', function() {
  // ..
  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()
      cy.contains('a note created by cypress')
    })
  })
})
```

The test has been defined in its own *describe* block. Only logged-in users can create new notes, so we added logging in to the application to a *beforeEach* block.

The test trusts that when creating a new note the page contains only one input, so it searches for it like so:

```
cy.get('input')
```

If the page contained more inputs, the test would break



Due to this problem, it would again be better to give the input an *ID* and search for the element by its ID.

The structure of the tests looks like so:

```
describe('Note app', function() {
  // ...

  it('user can log in', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })
```

```
  })
})
```

Cypress runs the tests in the order they are in the code. So first it runs *user can log in*, where the user logs in. Then cypress will run *a new note can be created* for which a *beforeEach* block logs in as well. Why do this? Isn't the user logged in after the first test? No, because *each* test starts from zero as far as the browser is concerned. All changes to the browser's state are reversed after each test.

Controlling the state of the database

If the tests need to be able to modify the server's database, the situation immediately becomes more complicated. Ideally, the server's database should be the same each time we run the tests, so our tests can be reliably and easily repeatable.

As with unit and integration tests, with E2E tests it is best to empty the database and possibly format it before the tests are run. The challenge with E2E tests is that they do not have access to the database.

The solution is to create API endpoints for the backend tests. We can empty the database using these endpoints. Let's create a new router for the tests inside the *controllers* folder, in the *testing.js* file

```
const testingRouter = require('express').Router()
const Note = require('../models/note')
const User = require('../models/user')

testingRouter.post('/reset', async (request, response) => {
  await Note.deleteMany({})
  await User.deleteMany({})

  response.status(204).end()
})

module.exports = testingRouter
```

and add it to the backend only *if the application is run in test-mode*:

```
// ...

app.use('/api/login', loginRouter)
app.use('/api/users', usersRouter)
app.use('/api/notes', notesRouter)

if (process.env.NODE_ENV === 'test') {
  const testingRouter = require('./controllers/testing')
  app.use('/api/testing', testingRouter)
}

app.use(middleware.unknownEndpoint)
app.use(middleware.errorHandler)
```

```
module.exports = app
```

After the changes, an HTTP POST request to the */api/testing/reset* endpoint empties the database. Make sure your backend is running in test mode by starting it with this command (previously configured in the package.json file):

```
npm run start:test
```

copy

The modified backend code can be found on the GitHub branch *part5-1*.

Next, we will change the *beforeEach* block so that it empties the server's database before tests are run.

Currently, it is not possible to add new users through the frontend's UI, so we add a new user to the backend from the beforeEach block.

```
describe('Note app', function() {
   beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/testing/reset')
    const user = {
      name: 'Matti Luukkainen',
      username: 'mluukkai',
      password: 'salainen'
    }
    cy.request('POST', 'http://localhost:3001/api/users/', user)
    cy.visit('http://localhost:5173')
  })

  it('front page can be opened', function() {
    // ...
  })

  it('user can login', function() {
    // ...
  })

  describe('when logged in', function() {
    // ...
  })
})
```

copy

During the formatting, the test does HTTP requests to the backend with cy.request.

Unlike earlier, now the testing starts with the backend in the same state every time. The backend will contain one user and no notes.

Let's add one more test for checking that we can change the importance of notes.

A while ago we changed the frontend so that a new note is important by default, so the *important* field is *true*:

```
const NoteForm = ({ createNote }) => {
  // ...

  const addNote = (event) => {
    event.preventDefault()
    createNote({
      content: newNote,
      important: true
    })

    setNewNote('')
  }
  // ...
}
```

There are multiple ways to test this. In the following example, we first search for a note and click its *make not important* button. Then we check that the note now contains a *make important* button.

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    // ...

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.contains('new note').click()
        cy.get('input').type('another note cypress')
        cy.contains('save').click()
      })

      it('it can be made not important', function () {
        cy.contains('another note cypress')
          .contains('make not important')
          .click()

        cy.contains('another note cypress')
          .contains('make important')
      })
    })
  })
})
```

The first command searches for a component containing the text *another note cypress*, and then for a *make not important* button within it. It then clicks the button.

The second command checks that the text on the button has changed to *make important*.

Failed login test

Let's make a test to ensure that a login attempt fails if the password is wrong.

Cypress will run all tests each time by default, and as the number of tests increases, it starts to become quite time-consuming. When developing a new test or when debugging a broken test, we can define the test with *it.only* instead of *it*, so that Cypress will only run the required test. When the test is working, we can remove *.only*.

First version of our tests is as follows:

```
describe('Note app', function() {
  // ...

  it.only('login fails with wrong password', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('wrong')
    cy.get('#login-button').click()

    cy.contains('wrong credentials')
  })

  // ...
)}
```

The test uses cy.contains to ensure that the application prints an error message.

The application renders the error message to a component with the CSS class *error*:

```
const Notification = ({ message }) => {
  if (message === null) {
    return null
  }

  return (
    <div className="error">
      {message}
    </div>
  )
}
```

We could make the test ensure that the error message is rendered to the correct component, that is, the component with the CSS class *error*:

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error').contains('wrong credentials')
})
```

First, we use cy.get to search for a component with the CSS class *error*. Then we check that the error message can be found in this component. Note that the CSS class selectors start with a full stop, so the selector for the class *error* is *.error*.

We could do the same using the should syntax:

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error').should('contain', 'wrong credentials')
})
```

Using should is a bit trickier than using *contains*, but it allows for more diverse tests than *contains* which works based on text content only.

A list of the most common assertions which can be used with `should` can be found here.

We can, for example, make sure that the error message is red and it has a border:

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error').should('contain', 'wrong credentials')
  cy.get('.error').should('have.css', 'color', 'rgb(255, 0, 0)')
  cy.get('.error').should('have.css', 'border-style', 'solid')
})
```

Cypress requires the colors to be given as rgb.

Because all tests are for the same component we accessed using cy.get, we can chain them using and.

```
it('login fails with wrong password', function() {
  // ...

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
```

```
    .and('have.css', 'border-style', 'solid')
})
```

Let's finish the test so that it also checks that the application does not render the success message *'Matti Luukkainen logged in'*:

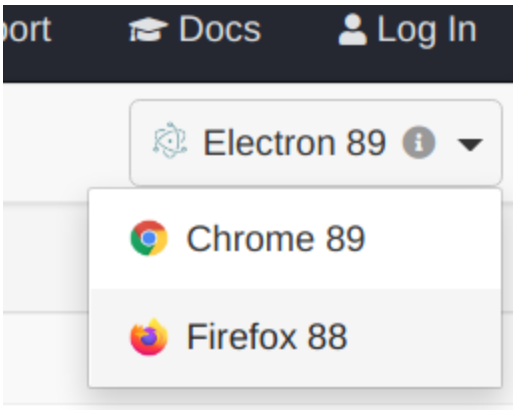
```
it('login fails with wrong password', function() {
  cy.contains('log in').click()
  cy.get('#username').type('mluukkai')
  cy.get('#password').type('wrong')
  cy.get('#login-button').click()

  cy.get('.error')
    .should('contain', 'wrong credentials')
    .and('have.css', 'color', 'rgb(255, 0, 0)')
    .and('have.css', 'border-style', 'solid')

  cy.get('html').should('not.contain', 'Matti Luukkainen logged in')
})
```

The command *should* is most often used by chaining it after the command *get* (or another similar command that can be chained). The *cy.get('html')* used in the test practically means the visible content of the entire application.

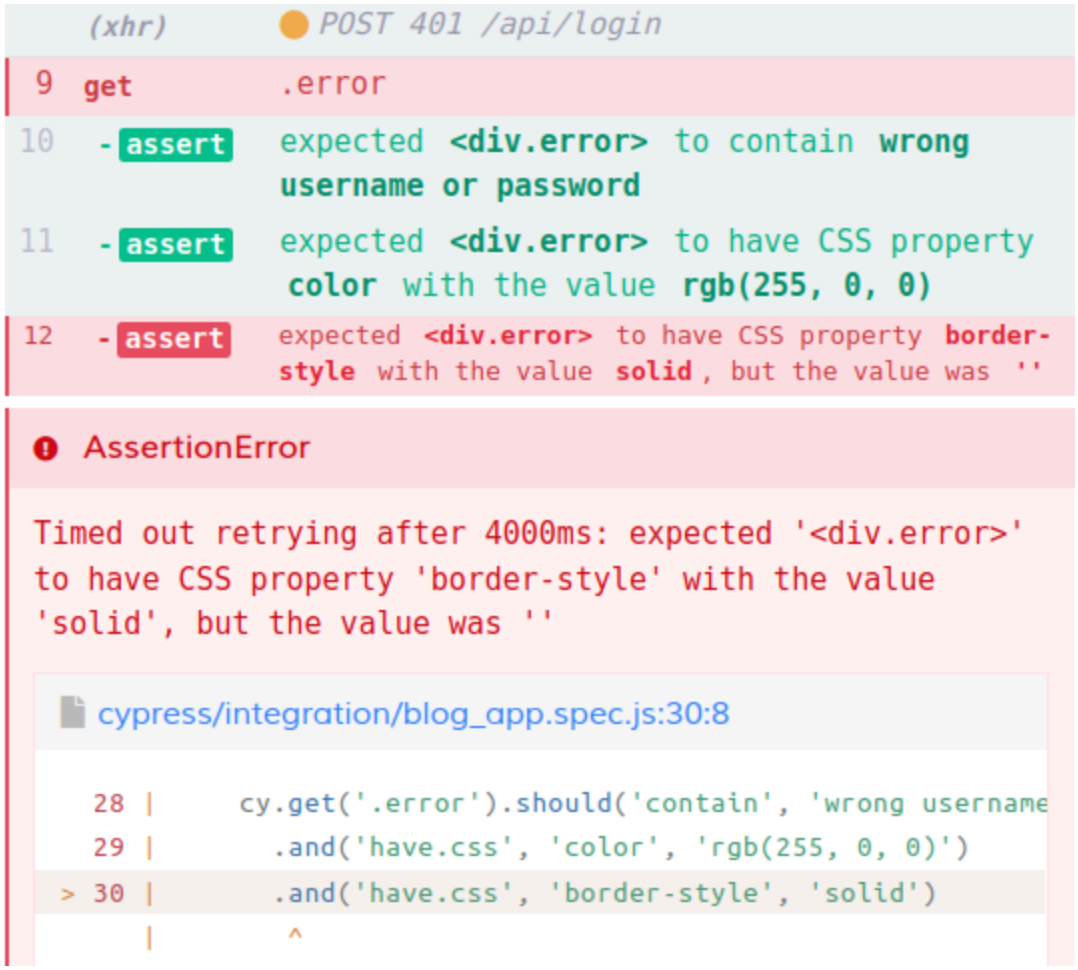
We would also check the same by chaining the command *contains* with the command *should* with a slightly different parameter:

```
cy.contains('Matti Luukkainen logged in').should('not.exist')
```

NOTE: Some CSS properties behave differently on Firefox. If you run the tests with Firefox:



then tests that involve, for example, `border-style`, `border-radius` and `padding`, will pass in Chrome or Electron, but fail in Firefox:



Bypassing the UI

Currently, we have the following tests:

```
describe('Note app', function() {
  it('user can login', function() {
    cy.contains('log in').click()
    cy.get('#username').type('mluukkai')
    cy.get('#password').type('salainen')
    cy.get('#login-button').click()

    cy.contains('Matti Luukkainen logged in')
  })

  it('login fails with wrong password', function() {
    // ...
  })

  describe('when logged in', function() {
    beforeEach(function() {
      cy.contains('log in').click()
      cy.get('input:first').type('mluukkai')
```

```
      cy.get('input:last').type('salainen')
      cy.get('#login-button').click()
    })

    it('a new note can be created', function() {
      // ...
    })

  })
})
```

First, we test logging in. Then, in their own describe block, we have a bunch of tests, which expect the user to be logged in. User is logged in in the *beforeEach* block.

As we said above, each test starts from zero! Tests do not start from the state where the previous tests ended.

The Cypress documentation gives us the following advice: Fully test the login flow – but only once. So instead of logging in a user using the form in the *beforeEach* block, we are going to bypass the UI and do a HTTP request to the backend to log in. The reason for this is that logging in with a HTTP request is much faster than filling out a form.

Our situation is a bit more complicated than in the example in the Cypress documentation because when a user logs in, our application saves their details to the localStorage. However, Cypress can handle that as well. The code is the following:

```
describe('when logged in', function() {
  beforeEach(function() {
    cy.request('POST', 'http://localhost:3001/api/login', {
      username: 'mluukkai', password: 'salainen'
    }).then(response => {
      localStorage.setItem('loggedNoteappUser', JSON.stringify(response.body))
      cy.visit('http://localhost:5173')
    })
  })

  it('a new note can be created', function() {
    // ...
  })

  // ...
})
```

We can access to the response of a cy.request with the `then` method. Under the hood *cy.request*, like all Cypress commands, are asynchronous. The callback function saves the details of a logged-in user to localStorage, and reloads the page. Now there is no difference to a user logging in with the login form.

If and when we write new tests to our application, we have to use the login code in multiple places, we should make it a custom command.

Custom commands are declared in *cypress/support/commands.js*. The code for logging in is as follows:

```
Cypress.Commands.add('login', ({ username, password }) => {
  cy.request('POST', 'http://localhost:3001/api/login', {
    username, password
  }).then(({ body }) => {
    localStorage.setItem('loggedNoteappUser', JSON.stringify(body))
    cy.visit('http://localhost:5173')
  })
})
```

Using our custom command is easy, and our test becomes cleaner:

```
describe('when logged in', function() {
  beforeEach(function() {
    cy.login({ username: 'mluukkai', password: 'salainen' })
  })

  it('a new note can be created', function() {
    // ...
  })

  // ...
})
```

The same applies to creating a new note now that we think about it. We have a test, which makes a new note using the form. We also make a new note in the *beforeEach* block of the test that changes the importance of a note:

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      cy.contains('new note').click()
      cy.get('input').type('a note created by cypress')
      cy.contains('save').click()

      cy.contains('a note created by cypress')
    })

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.contains('new note').click()
        cy.get('input').type('another note cypress')
        cy.contains('save').click()
      })
```

```
      it('it can be made important', function () {
        // ...
      })
    })
  })
})
```

Let's make a new custom command for making a new note. The command will make a new note with an HTTP POST request:

```
Cypress.Commands.add('createNote', ({ content, important }) => {
  cy.request({
    url: 'http://localhost:3001/api/notes',
    method: 'POST',
    body: { content, important },
    headers: {
      'Authorization': `Bearer
${JSON.parse(localStorage.getItem('loggedNoteappUser')).token}`
    }
  })

  cy.visit('http://localhost:5173')
})
```

The command expects the user to be logged in and the user's details to be saved to localStorage.

Now the note beforeEach block becomes:

```
describe('Note app', function() {
  // ...

  describe('when logged in', function() {
    it('a new note can be created', function() {
      // ...
    })

    describe('and a note exists', function () {
      beforeEach(function () {
        cy.createNote({
          content: 'another note cypress',
          important: true
        })
      })

      it('it can be made important', function () {
        // ...
      })
    })
```

```
  })
})
```

There is one more annoying feature in our tests. The application address *http:localhost:5173* is hardcoded in many places.

Let's define the *baseUrl* for the application in the Cypress pre-generated configuration file *cypress.config.js*:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
    },
    baseUrl: 'http://localhost:5173'
  },
})
```

copy

All the commands in the tests use the address of the application

```
cy.visit('http://localhost:5173' )
```

copy

can be transformed into

```
cy.visit('')
```

copy

The backend's hardcoded address *http://localhost:3001* is still in the tests. Cypress documentation recommends defining other addresses used by the tests as environment variables.

Let's expand the configuration file *cypress.config.js* as follows:

```
const { defineConfig } = require("cypress")

module.exports = defineConfig({
  e2e: {
    setupNodeEvents(on, config) {
    },
    baseUrl: 'http://localhost:5173',
    env: {
      BACKEND: 'http://localhost:3001/api'
    }
```

copy

```
  },
})
```

Let's replace all the backend addresses from the tests in the following way

```
describe('Note ', function() {
  beforeEach(function() {

    cy.request('POST', `${Cypress.env('BACKEND')}/testing/reset`)
    const user = {
      name: 'Matti Luukkainen',
      username: 'mluukkai',
      password: 'secret'
    }
    cy.request('POST', `${Cypress.env('BACKEND')}/users`, user)
    cy.visit('')
  })
  // ...
})
```

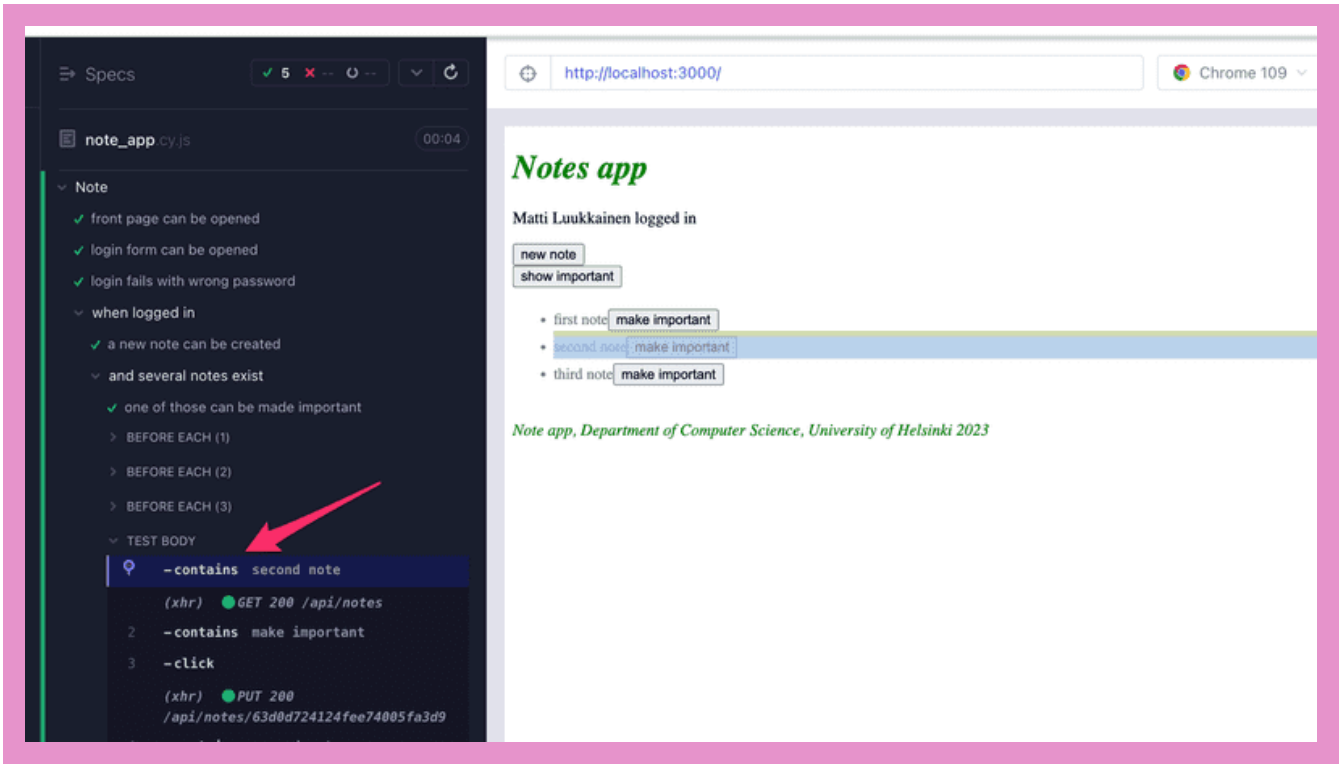
Changing the importance of a note

Lastly, let's take a look at the test we did for changing the importance of a note. First, we'll change the beforeEach block so that it creates three notes instead of one:

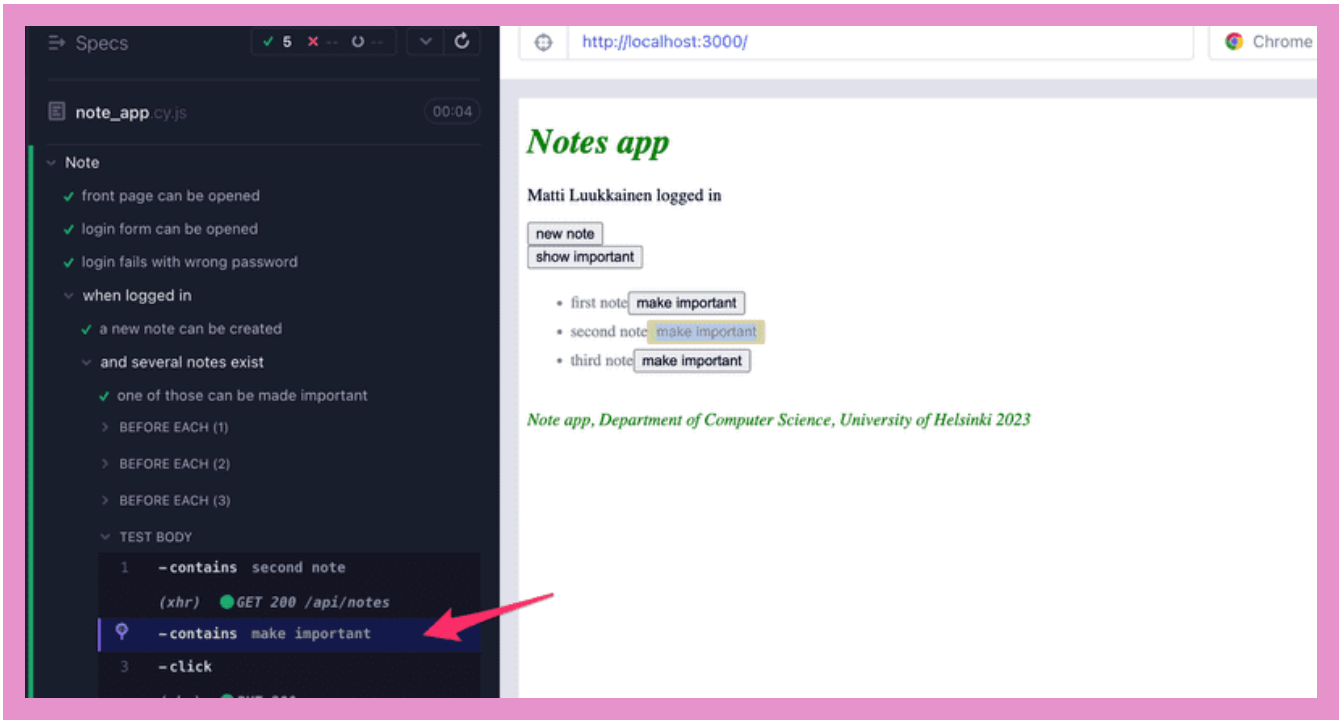
```
describe('when logged in', function() {
  describe('and several notes exist', function () {
    beforeEach(function () {
      cy.login({ username: 'mluukkai', password: 'salainen' })
      cy.createNote({ content: 'first note', important: false })
      cy.createNote({ content: 'second note', important: false })
      cy.createNote({ content: 'third note', important: false })
    })

    it('one of those can be made important', function () {
      cy.contains('second note')
        .contains('make important')
        .click()

      cy.contains('second note')
        .contains('make not important')
    })
  })
})
```

How does the cy.contains command actually work?

When we click the `cy.contains('second note')` command in Cypress Test Runner, we see that the command searches for the element containing the text *second note*:



By clicking the next line `.contains('make important')` we see that the test uses the 'make important' button corresponding to the *second note*:

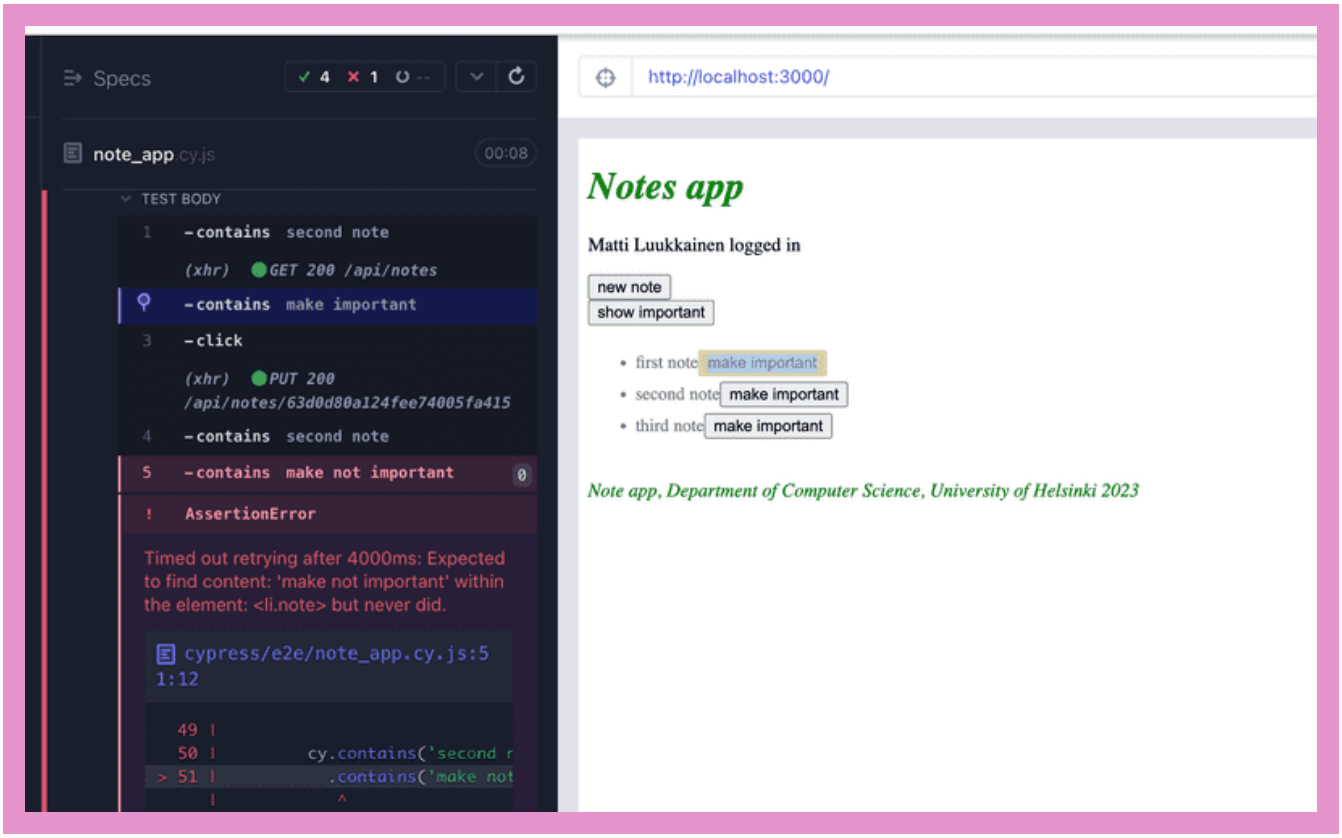


When chained, the second *contains* command *continues* the search from within the component found by the first command.

If we had not chained the commands, and instead write:

```
cy.contains('second note')
cy.contains('make important').click()
```

the result would have been entirely different. The second line of the test would click the button of a wrong note:

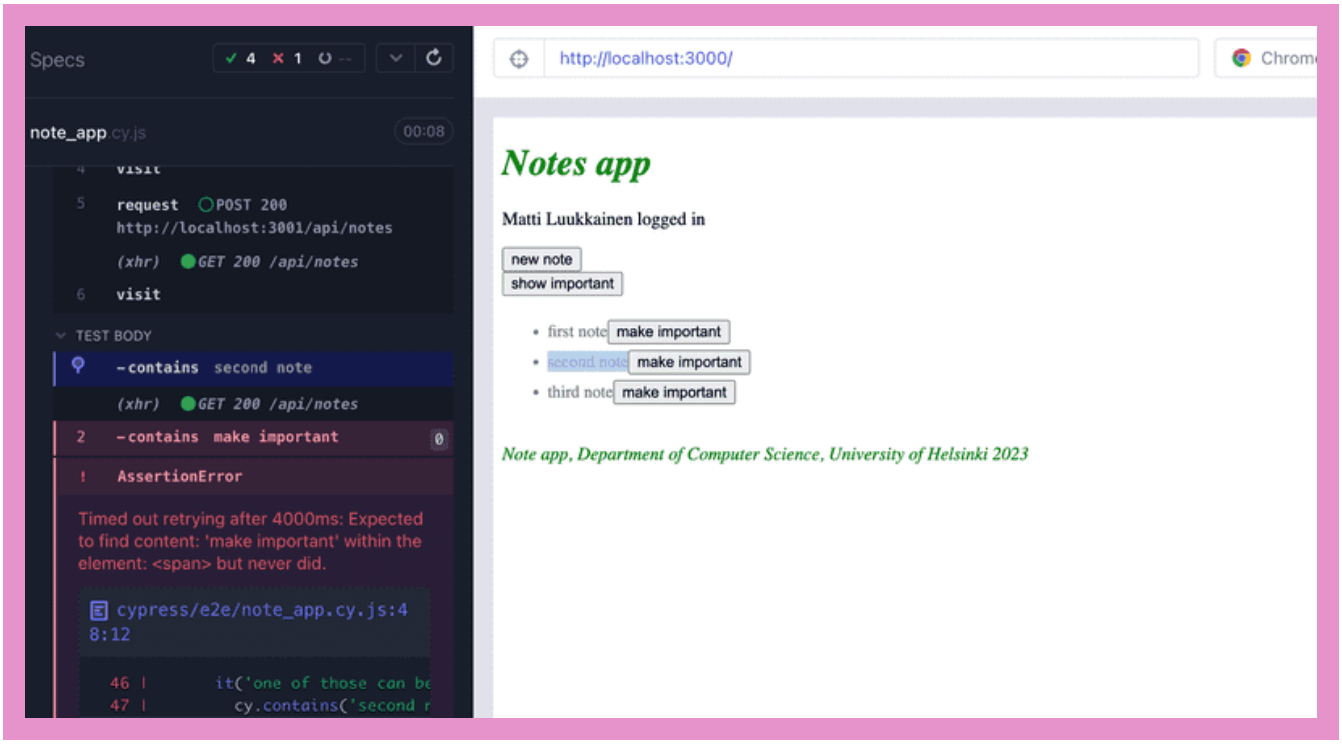


When coding tests, you should check in the test runner that the tests use the right components!

Let's change the `Note` component so that the text of the note is rendered to a *span*.

```
const Note = ({ note, toggleImportance }) => {
  const label = note.important
    ? 'make not important' : 'make important'

  return (
    <li className='note'>
      <span>{note.content}</span>
      <button onClick={toggleImportance}>{label}</button>
    </li>
  )
}
```

Our tests break! As the test runner reveals, `cy.contains('second note')` now returns the component containing the text, and the button is not in it.



One way to fix this is the following:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').click()
  cy.contains('second note').parent().find('button')
    .should('contain', 'make not important')
})
```

In the first line, we use the `parent` command to access the parent element of the element containing *second note* and find the button from within it. Then we click the button and check that the text on it changes.

Note that we use the command `find` to search for the button. We cannot use `cy.get` here, because it always searches from the *whole* page and would return all 5 buttons on the page.

Unfortunately, we have some copy-paste in the tests now, because the code for searching for the right button is always the same.

In these kinds of situations, it is possible to use the `as` command:

```
it('one of those can be made important', function () {
  cy.contains('second note').parent().find('button').as('theButton')
  cy.get('@theButton').click()
  cy.get('@theButton').should('contain', 'make not important')
})
```

Now the first line finds the right button and uses *as* to save it as *theButton*. The following lines can use the named element with *cy.get('@theButton')*.

Running and debugging the tests

Finally, some notes on how Cypress works and debugging your tests.

Because of the form of the Cypress tests, it gives the impression that they are normal JavaScript code, and we could for example try this:

```
const button = cy.contains('log in')
button.click()
debugger
cy.contains('logout').click()
```

copy

This won't work, however. When Cypress runs a test, it adds each `cy` command to an execution queue. When the code of the test method has been executed, Cypress will execute each command in the queue one by one.

Cypress commands always return `undefined`, so `button.click()` in the above code would cause an error. An attempt to start the debugger would not stop the code between executing the commands, but before any commands have been executed.

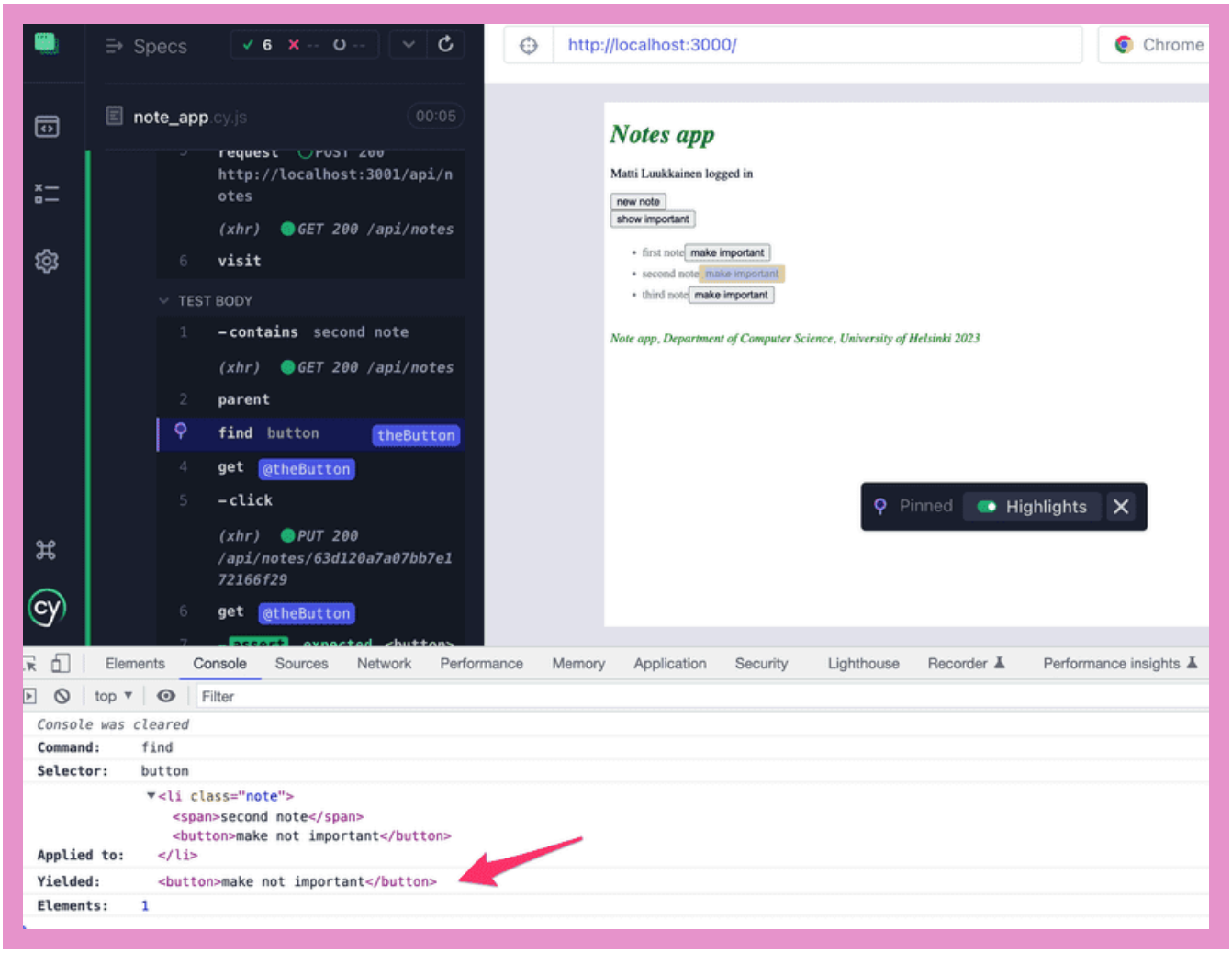
Cypress commands are *like promises*, so if we want to access their return values, we have to do it using the `then` command. For example, the following test would print the number of buttons in the application, and click the first button:

```
it('then example', function() {
  cy.get('button').then( buttons => {
    console.log('number of buttons', buttons.length)
    cy.wrap(buttons[0]).click()
  })
})
```

copy

Stopping the test execution with the debugger is possible. The debugger starts only if Cypress test runner's developer console is open.

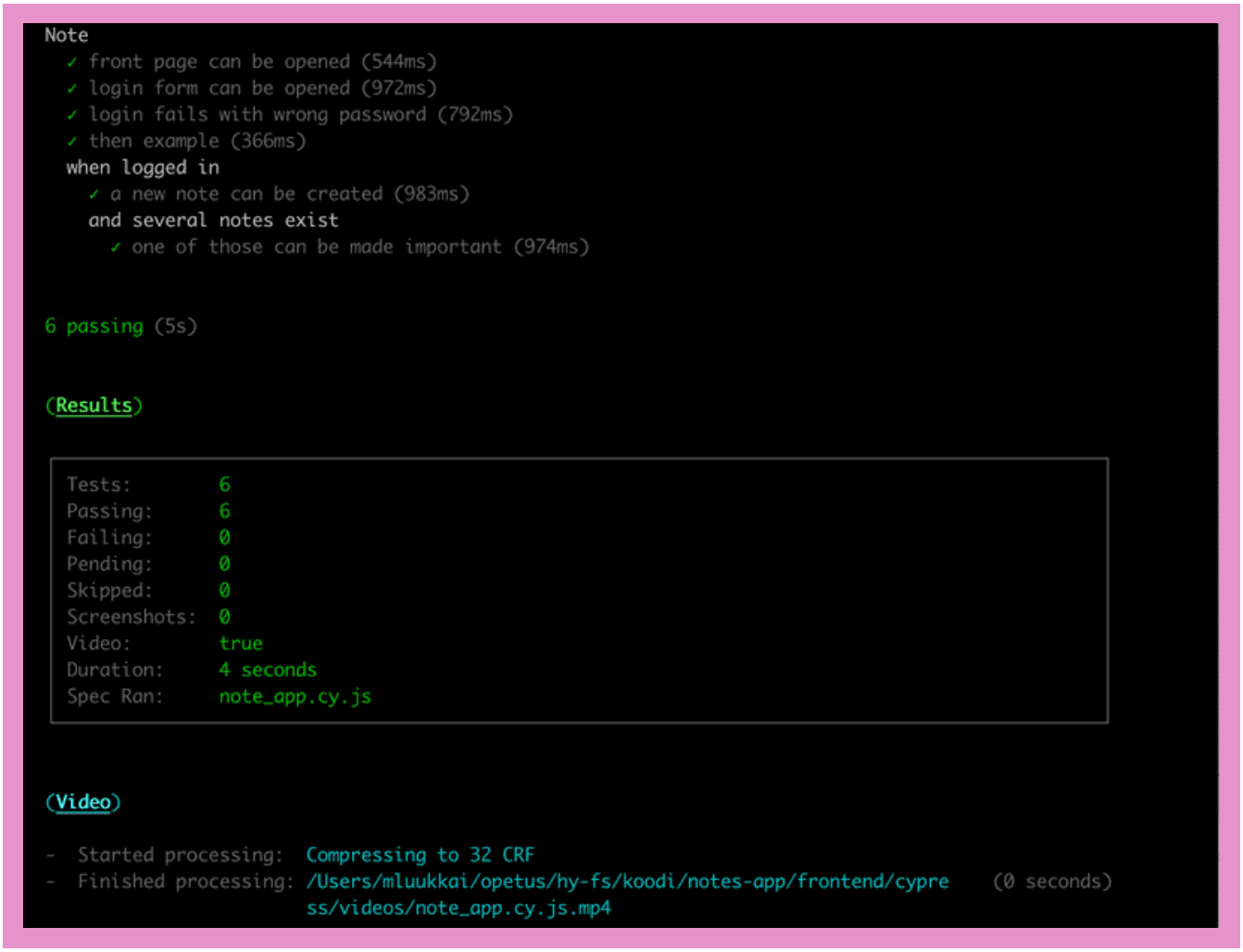
The developer console is all sorts of useful when debugging your tests. You can see the HTTP requests done by the tests on the Network tab, and the console tab will show you information about your tests:



So far we have run our Cypress tests using the graphical test runner. It is also possible to run them from the command line. We just have to add an npm script for it:

```
"scripts": {
  "cypress:open": "cypress open",
  "test:e2e": "cypress run"
},
```

Now we can run our tests from the command line with the command *npm run test:e2e*



Note that videos of the test execution will be saved to *cypress/videos/*, so you should probably git ignore this directory. It is also possible to turn off the making of videos.

Tests are found in GitHub.

Final version of the frontend code can be found on the GitHub branch *part5-9*.

Exercises 5.17.-5.23.

In the last exercises of this part, we will do some E2E tests for our blog application. The material of this part should be enough to complete the exercises. You **must check out the Cypress documentation**. It is probably the best documentation I have ever seen for an open-source project.

I especially recommend reading Introduction to Cypress, which states

> *This is the single most important guide for understanding how to test with Cypress. Read it. Understand it.*

5.17: Blog List End To End Testing, step 1

Configure Cypress for your project. Make a test for checking that the application displays the login form by default.

The structure of the test must be as follows:

```
describe('Blog app', function() {
  beforeEach(function() {
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })
})
```

5.18: Blog List End To End Testing, step 2

Make tests for logging in. Test both successful and unsuccessful login attempts. Make a new user in the *beforeEach* block for the tests.

The test structure extends like so:

```
describe('Blog app', function() {
  beforeEach(function() {
    // empty the db here
    // create a user for the backend here
    cy.visit('http://localhost:5173')
  })

  it('Login form is shown', function() {
    // ...
  })

  describe('Login',function() {
    it('succeeds with correct credentials', function() {
      // ...
    })

    it('fails with wrong credentials', function() {
      // ...
    })
  })
})
```

The `beforeEach` block must empty the database using, for example, the reset method we used in the material.

Optional bonus exercise: Check that the notification shown with unsuccessful login is displayed red.

5.19: Blog List End To End Testing, step 3

Make a test that verifies a logged-in user can create a new blog. The structure of the test could be as follows:

```
describe('Blog app', function() {
  // ...

  describe('When logged in', function() {
    beforeEach(function() {
      // ...
    })

    it('A blog can be created', function() {
      // ...
    })
  })

})
```

The test has to ensure that a new blog is added to the list of all blogs.

5.20: Blog List End To End Testing, step 4

Make a test that confirms users can like a blog.

5.21: Blog List End To End Testing, step 5

Make a test for ensuring that the user who created a blog can delete it.

5.22: Blog List End To End Testing, step 6

Make a test for ensuring that only the creator can see the delete button of a blog, not anyone else.

5.23: Blog List End To End Testing, step 7

Make a test that checks that the blogs are ordered by likes, with the most liked blog being first.

This exercise is quite a bit trickier than the previous ones. One solution is to add a certain class for the element which wraps the blog's content and use the `eq` method to get the blog element in a specific index:

```
cy.get('.blog').eq(0).should('contain', 'The title with the most likes')
cy.get('.blog').eq(1).should('contain', 'The title with the second most likes')
```

Note that you might end up having problems if you click a like button many times in a row. It might be that cypress does the clicking so fast that it does not have time to update the app state in between the clicks. One remedy for this is to wait for the number of likes to update in between all clicks.

This was the last exercise of this part, and it's time to push your code to GitHub and mark the exercises you completed in the exercise submission system.

Propose changes to material

Part 5d
Previous part

Part 6
Next part

About course

Course contents

FAQ

Partners

Challenge


UNIVERSITY OF HELSINKI


HOUSTON