Assignment: React Todo List

In this exercise you will create a Todo list to which you can add items.

Difficulty: ☆☆☆☆

Learning objectives:

- I can install the dependencies of React project
- I can add new dependencies to the project
- I can create functional React components
- I am familiar how to link a css file style into a React component
- I can use the useState hook
- I can use the useEffect hook
- I know how React components communicate
- I can implement page routing with react-router-dom
- I can issue a GET http request with axios

Estimated time required: 90 minutes

Disclaimer: this exercise has been created using React 18.2. The front-end development world is a very rapidly changing world, so some things may be out of date already. If you run into issues, try to search solutions for your specific version or revert to React version 18.2.

1. Setup environment

Although you are free to choose your own IDE, the default and recommended one is Visual Studio Code.

In order to run React, you need to install NPM (Node Package Manager).

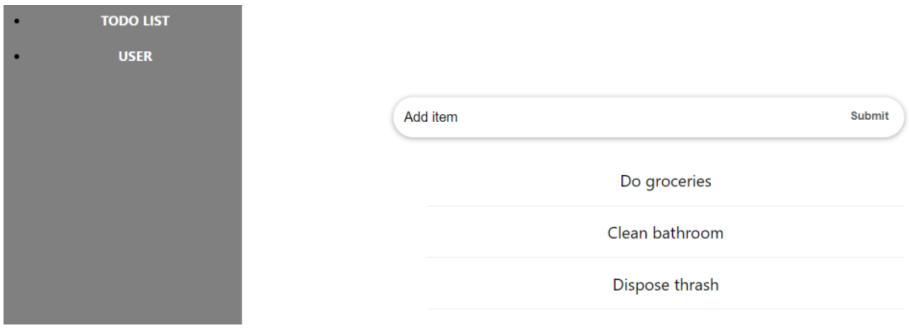
1. Verify you have installed NPM by running `npm -v`
2. If you get something different than the version number, install Node.js using this tutorial and repeat step 1.
3. Download the todolist starter project from Canvas
4. Run `cd <projectname>` to change directory into your project folder.
5. Run `npm i` or `npm install` to install all dependencies. This will take all dependencies in package.json and install them in the node_modules folder. Never commit or upload the node_modules folder as it is very large!
6. Run `npm run dev` to run your app!

Field Code Changed

Note: Different from back-end development you can see your code changes live in the browser. Just save the source code file and you will see the latest changes (or bugs) in your browser immediately!

2. Create your project structure

Your application will look like the image below.



First think about what **components** your application will have. You will have pages (TodoPage, UserPage) and components: the input field and the list of todo items. We can view each todo item as a separate component.

3. Create the TodoList component

Have a look at the starter project, in particular the files **TodoPage.jsx** and **TodoList.jsx** and make sure to understand the code before continuing. In particular:

```
<TodoList todoItems={todoItems} />
```

In the return statement of **TodoPage.jsx** we pass the todoItems list as a property to TodoList.

```
function TodoList(props) {

    return (
        <ul>
            {props.todoItems.map(todoItem => (
                <TodoItem key={todoItem.id} todoItem={todoItem} />
            ))}
        </ul>
    )
}
```

In TodoList we use the props (= todoItems) to create a TodoItem for each element in the list. Recall that Javascript supports functional programming, so typically you use a *map* statement instead of a *for* loop you would use in imperative programming (such as C#).

Next, in the TodoItem component we take the todoItem from the props and display its title.

```
function TodoItem(props) {

    return (
        <li className={styles.item}>{props.todoItem.title}</li>
    )
}
```

4. Complete the InputItem component

Next we will complete the *InputItem* component where you can add a new todo item to the list. Have a look at the code in **InputItem.jsx** and see whether you understand it.

1. Complete the function *textChanged* by saving the text that the user has provided in the state of the component.
2. Complete the function *handleSubmit* by adding the item to the *todoItems* list. Note that this list is located in the **TodoPage** component! While in imperative programming (C#) you would typically return a value, in functional programming callbacks are more commonly used. The function *addItem* in **TodoPage** should help.

Check whether your input field is functional. Your solution will probably look like:

```
    const handleSubmit = e => {
        e.preventDefault();
        props.addItem(title)
    }

    const textChanged = e => {
        setTitle(e.target.value)
    }
```

Note that you get a warning about duplicate ID's which occurs because all the ID's are hard-coded. You could generate random ID's using the library "uuid", but this is not the focus of this exercise.

Field Code Changed

For your individual project you will probably need some input validation. You can do this via very basic if-statement checks, but if your forms get larger and more complicated we strongly recommend using the Formik and Yup libraries.

Field Code Changed

Field Code Changed

Note: communication like the one implemented above (from *InputItem* to *TodoPage*) is also known as child to parent communication. Check this article to know more.

Field Code Changed

5. Implementing a second page with React Router

We're now going to add a second page to our application, a simple user's page. To navigate between different pages, we need a package called React Router.

Field Code Changed

Afterwards the implementation involves two parts:

- indicating which page should be displayed accordingly to an URL ("/" for the todo page, "/user" for the user's page)
- Adding links on the navigation bar that refer to that particular path.

Lets do it step-by-step:

Formatted: English (United States)

1. Run `npm install react-router-dom --save` to install React Router.
2. Add a route to indicate that the userpage is located at "/user".

   For the first part we need to implement a Router. Typically this component is placed on the top-level of your app (in **App.jsx**) to enable routing everywhere. The code snippet below indicates that the root should route to the TodoPage. Note that the NavBar is in the Router component as well.

```
    <Router>
        <NavBar />
        <Routes>
            <Route path="/" element={<TodoPage />} />
        </Routes>
    </Router>
```

Based on this code now add a route with path */user* to the element *UserPage*.

3. Using the *links* variable in the **NavBar** component, add *a <NavLink>* component to ensure navigation works.

   Here we need to go into the **NavBar** component and add hyperlinks to refer to the correct pages. This works very similar to the way <a href> works in HTML, where you would put <a href="/path"> text for link </a>. Instead of <a href>, in React we use a <NavLink> component.

```
<NavLink to={link.path}>
    {link.text}
</NavLink>
```

Field Code Changed

6. Creating a user page by applying effects

In the next part we will display the name of the user in the **UserPage** component. For the purposes of this exercise we will get the username from a public API generating random usernames, but usually you obtain them from the server-side of your application. Using randomly generated names has the nice benefit that we can quickly see when a component gets rerendered, which will be important later.

1. Run `npm install axios --save` to install axios, a package used for performing HTTP requests.

Field Code Changed

Fetching data (such as usernames) is a **side effect** of our **UserPage** component and hence is usually implemented using *Effect Hooks*.

Field Code Changed

2. In the **UserPage** component, understand the code in *fetchUsernames* and call it using the *useEffect* hook.

You should see names being rendered, but there is an issue! Names keep being generated all the time!

useEffect runs after every render. Recall that every time the component's state is changed (such as username), the component is re-rendered. This will trigger another useEffect call, changing the state again and resulting in an infinite loop.

To solve this, we make use of the second argument of *useEffect*: a list of dependencies. If any variable in the list changes, the effect will run again. Since we only want the effect to run once, we provide an empty list []. This is how the code should look like:

Field Code Changed

```
    useEffect(() => {
        fetchUsernames();
    }, []);
```

3. Is *fetchUsernames* really being called once?

Have a look at the result. You should see three renders quickly after each other: The first one is “Welcome undefined”, after which two different names are being displayed. Let’s investigate this issue now.

The effect is called twice resulting in two generated names (you can also see in the Network tab that two API calls are being made). This results from React running in StrictMode. In Strict Mode, React runs additional checks and warnings. They run in development mode only and do **not** impact the production build, so usually it’s a good idea to leave it on. Unfortunately, it also causes the code to be ran twice, including our *Effect Hook*.

This can be verified by removing *<React.StrictMode>* tag from main.jsx:

```
root.render(
    <App />
);
```

But again, because it doesn’t affect your production build, it’s better keep *<React.StrictMode>* on and only be aware of this strange behavior. Besides that, extra GET requests shouldn’t do any harm during development.

7. Extend your app

This exercise was created to touch the most important concepts of React. If you want to practice more, you can extend the app by adding more functionality such as:

- Deleting todo items
- Adding a checkbox and a property “done” to check off items.
- Adding a counter or percentage of how many tasks have been finished.

Field Code Changed

Field Code Changed

Field Code Changed