Eric Keränen 26.5.2021

React Todo-App Documentation

Introduction

This is a short documentation of my Todo-App with React to summarise what have I done, how to install it, how does it work and what I have learned. My goal was to create a React Todo-App to practice coding React with hooks and functional components and Tailwindcss. I followed a tutorial by Brian Design on Youtube.

Features

The Todo-App gives the user the opportunity to add todos into a list. When a todo is added to the list it can be set as completed or it can be deleted or edited. When a todo is set as complete, it will have a line going through it and have lower opacity. The completed todos will stay in the list and can still be interacted with. Only when they are deleted, do they vanish from the list.

For the theme of the app, I went for a somewhat natural theme. I used mostly green colours and kept it relatively clutterless and minimalistic.

Future development plans for this Todo-App would be to implement a local storage system so the todos would be saved to the browsers storage for later use. I did not implement it in this project since I put all my time into the React and Tailwindcss code and to understand them and use them effectively. Other plans would be to instead use a database over the local storage.

How to install

First you will have to have Node.js installed which can be downloaded from the link below.

https://nodejs.org/en/

To use the application, you navigate into a directory on your terminal and clone the project repository with the following command:

```
git clone https://github.com/Mahamurahti/React-Roadmap.git
```

Then you need to navigate into the todo folder with:

```
cd React/todo/
```

When in the folder, you need to install all the dependencies with (could take a small while):

```
npm install
```

After installing dependencies, you can start the application by typing:

```
npm start
```

This will open the application into localhost:3000.

Application logic

In the application we have three components:

1. TodoList, which holds the title, the input field and all the todo items inside of it. In this section we will refer to this component as **list**.

2. TodoInput, where the user can type in a todo. In this section we will refer to this component as **input**.
3. TodoItem, which displays the information of a todo to the user. In this section we will refer to this component as **item**.

When the application starts, it renders the list which will display the title and the input component, but no items, since the list component is empty of todos. When the user types in the input, a handleChange function will keep the input variable up to date. When the *Add*-button is pressed, the input component submits the content of the input field with a function it gets from the list component, since the list is the parent. Then the submitted item will be saved into the list component which in turn gives the list of todos to the item component. The item component will then render the just typed content, with a few buttons accompanying it.

The *checkmark*-button will set the item to a completed state. The button will call a completeTodo function from the list component to set the todo with the correct id to completed, then render the list again. When an item is set as completed, nothing else will change except its appearance.

The *delete*-button will delete and item from the list. The button will call a deleteTodo function from the list component. This function will just set the list again by filtering away the item with the correct id. After setting the list, the list component re-renders.

The *edit*-button will first set the edit variable to the clicked item in the item component, which will re-render the component. During this re-render the item component checks if any of the todos inside of it match the edit variables id. Upon finding a match it does not render the item component, but instead renders the input component. This input component will have a different appearance to indicate to the user that the item can be edited. When the item is edited, the user can press the *Update*-button, which calls a submitEdit function from the item component which in turn will call an editTodo function from the list component. The editTodo function then sets all the items as is, except for the just edited item which will render with the new value in it.

What have I learned?

I learned a lot about hooks, rendering and a little bit about tailwindcss. **Functional components** also seemed a lot easier and understandable than the class system with React. This-keyword in the class system of React was very confusing and getting rid of it was very helpful. **JSX syntax** I also learned fast since it is basically HTML with easier ways to add JS into the mix in my opinion.

Hooks I studied thoroughly through React documentation and their tutorials, before tackling this project. Only hook which at the start I did not know was the **useRef** hook, but now I have a basic level of understanding about it. **UseState** I used a lot and know that it returns always a variable and a “setter” for it. The “setter” function also causes a re-render for the component. The **useEffect** hook was also quite handy in conjunction with useRef, when focusing the cursor into the input field.

Tailwindcss was very tricky to get into first since all of it is inline css mostly, which I do not use. But after getting the basics under control from Tailwindcss documentation, it seemed handy. I would say tailwindcss is useful, but still I am more lenient towards vanilla css or sass since I am so used to them. Tailwindcss saves a lot of lines of code and is faster after you know all the rules (since they are shrinken versions of the normal rules e.g. `background-color: rgba(254, 202, 202, 1);` is equal to `bg-red-200` in tailwindcss), which is a very big plus.