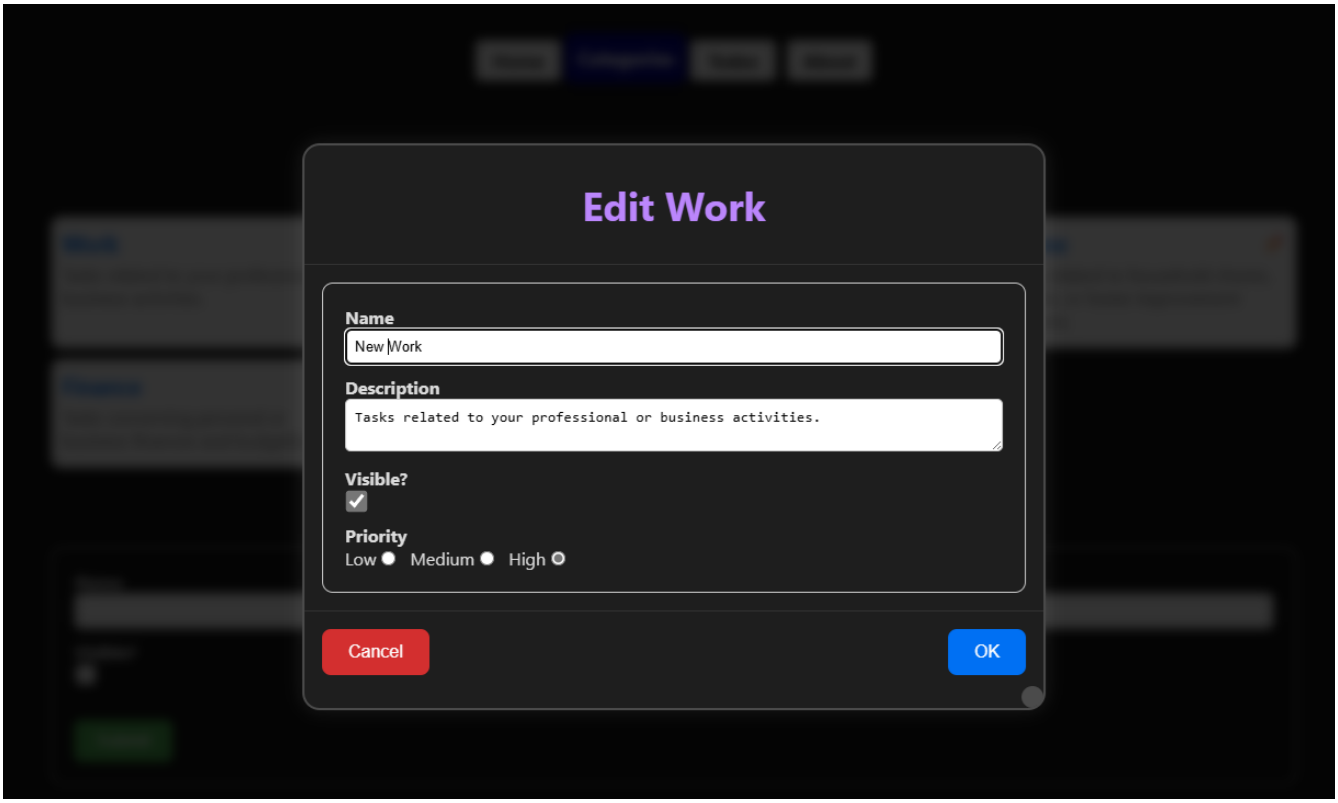
Daten editieren und Komponentenkommunikation



Link zum Programm: Edit_Form20241212.zip, im Repo unter 30_TodoApp/Edit_Form.

Die Komponente `ModalDialog`

Oft benötigt man eine Komponente, die einen modalen Dialog anzeigt. Er soll über der Seite angezeigt werden und nette Animationen besitzen. Anstatt in jeder Page und Komponente den Dialog neu zu implementieren, ist es sinnvoll, eine eigene Komponente zu erstellen.

```
<ModalDialog
    title="A nice title"
    onOk={() => doSomething()} onCancel={() => doSomething()}>
    <p>Some content</p>
</ModalDialog>
```

Der Dialog wird — wie das Beispiel zeigt — mit Parametern konfiguriert. Die `onOk` und `onCancel` Funktionen werden aufgerufen, wenn der Benutzer auf den entsprechenden Button klickt. Der Titel und der Inhalt des Dialogs werden ebenfalls übergeben. Der Inhalt steht zwischen den Tags der Komponente.

Wie können wir nun auf diese Informationen zugreifen? React gibt die Argumente der Komponente

als Objekt an die Funktion weiter. Um typsicher arbeiten zu können, definieren wir einen Type, der die Argumente beschreibt. Die Funktionen haben den Typ `() ⇒ void`, da sie keine Argumente erwarten und keinen Wert zurückgeben. Der Inhalt des Dialogs kann beliebig sein, daher verwenden wir den Typ `ReactNode` für `children`.

src/app/components/ModalDialog.tsx

```
1  import { ReactNode } from "react";
2  import styles from "./ModalDialog.module.css";
3
4  type ModalDialogProps = {
5    title: string;
6    onOk?: () => void;
7    onCancel?: () => void;
8    children: ReactNode;
9  }
10
11 export default function ModalDialog({ title, onOk, onCancel, children }:
   ModalDialogProps) {
12   return (
13     <div className={styles.overlay}>
14       <div className={styles.modal}>
15         <div className={styles.header}>
16           <h2>{title}</h2>
17         </div>
18         <div className={styles.content}>
19           {children}
20         </div>
21         <div className={styles.footer}>
22           <button className={styles.cancelButton} onClick={onCancel}>
23             Cancel
24           </button>
25           <button className={styles.okButton} onClick={onOk}>
26             OK
27           </button>
28         </div>
29       </div>
30     </div>
31   );
32 }
```

Source files in der Edit Form App

- ModalDialog.tsx
- ModalDialog.module.css

Verwendung der `ModalDialog` Komponente für das Edit Formular

Nachdem wir die `ModalDialog` Komponente erstellt haben, können wir sie in der `CategoryList` Komponente verwenden. Wir wollen das Bearbeiten einer Kategorie in einem Dialog anzeigen. Dafür geben wir eine neu zu schreibende Komponente `CategoryEdit` als Content der `ModalDialog` Komponente weiter. Dies sieht dann so aus:

src/app/categories/CategoryList.tsx

```
"use client";
// Some imports

export default function CategoryList({ categories }: { categories: Category[] }) {
  const [selectedCategory, setSelectedCategory] = useState<Category | null>(null);
  const categoryEditRef = useRef<CategoryEditRef>(null);

  return (
    <div className={styles.categories}>
      <ul>
        {categories.map(category => (
          { /* ... */ }
              <span onClick={() => setSelectedCategory(category)}>
                
              </span>
          { /* ... */ }
        ))}
      </ul>

      {selectedCategory && (
        <ModalDialog
          title={`Edit ${selectedCategory.name}`}
          onOk={() => categoryEditRef.current?.startSubmit()}
          onCancel={() => setSelectedCategory(null)}>

            <CategoryEdit
              category={selectedCategory} ref={categoryEditRef}
              onSubmitted={() => setSelectedCategory(null)} />

        </ModalDialog>
      )}
    </div>
  );
}
```

Zuerst wird bei jeder Kategorie ein Edit-Icon angezeigt. Beim Klicken wird mit dem State `selectedCategory` die ausgewählte Kategorie gespeichert. Wenn eine Kategorie ausgewählt wurde, wird der Dialog angezeigt. Die Anweisung `selectedCategory && ⋯` sorgt dafür, dass der Dialog nur angezeigt wird, wenn eine Kategorie ausgewählt wurde. Die Handler `onCancel` setzt die ausgewählte

Kategorie zurück und "schließt" den Dialog, indem `selectedCategpry` auf `null` gesetzt wird. Die Komponente `CategoryEdit` besitzt auch ein Event: `onSubmitted`. Dieses wird aufgerufen, wenn die Daten erfolgreich an die API gesendet wurden.

Der Handler für `onOk` liest sich etwas seltsam. Wir haben nämlich ein Problem: In einem Formular gibt es üblicherweise einen Submit-Button, der das Formular abschickt. Der OK Button wird allerdings im Dialog angezeigt, und nicht im Formular. Wir müssen daher das Senden der Formulardaten über JavaScript auslösen. Daher muss die Komponente `CategoryEdit` eine Methode `startSubmit` besitzen, die das Senden der Daten auslöst und die wir in der `CategoryList` Komponente aufrufen können.

Source files in der Edit Form App

- CategoryList.tsx
- CategoryList.module.css

Die Komponente `CategoryEdit`

Erweitern der API

In der Datei `categoryApiClient.ts` haben wir bereits Funktionen, die mit der API kommunizieren. Hier fügen wir die Funktion `editCategory` hinzu, die den PUT-Request an die API sendet. Der Aufbau ist wie bei den anderen Funktionen: Es wird ein `FormData` Objekt übergeben, das die Daten der Kategorie enthält. Beachte, dass die GUID auch aus dem Formular extrahiert wird. Es wird mit dem Typ `hidden` in einem `input` Element im Formular gespeichert. Danach wird der PUT Request an `/api/categories/{guid}` gesendet. Im Fehlerfall wird ein `ErrorResponse` Objekt zurückgegeben, damit das Formular - wie beim Hinzufügen von Katagorien - den Fehler anzeigen kann.

src/app/categories/categoryApiClient.ts

```
1  export async function editCategory(formData: FormData): Promise<ErrorResponse |
   undefined> {
2      // Extrahiere Daten aus dem Formular
3      const guid = formData.get("guid");
4      if (!guid) {
5          return createErrorResponse(new Error("Invalid guid"));
6      }
7      const data = {
8          guid: guid,
9          name: formData.get("name"),
10         description: formData.get("description"),
11         isVisible: !!formData.get("isVisible"),     // converts null to false.
12         priority: formData.get("priority"),
13     };
14
15     try {
16         // Sende einen PUT-Request an die API
17         await axiosInstance.put(`categories/${guid}`, data);
```

```
18         revalidatePath("/categories");
19     } catch (e) {
20         return createErrorResponse(e);
21     }
22 }
```

 Source code: categoryApiClient.ts

Erstellen der `CategoryEdit` Komponente

Die Komponente `CategoryEdit` ähnelt der `CategoryAdd` Komponente. Im Formular müssen wir aber Werte vorbelegen, die bereits in der Kategorie gespeichert sind. Dies geschieht mit `defaultValue` bzw. `defaultChecked` bei den `input` Elementen. Wenn wir das Attribut `value` setzen, aber keinen `onChange`-Handler definieren, entsteht ein Fehler. Daher brauchen wir diese speziellen Attribute, um die Werte zu setzen.

refs als Parameter

Dieses Feature gibt es ab React 18. In React 17 und darunter benötigt man die Funktion `forwardRef`, um Refs an Komponenten weiterzugeben.[1]

Damit wir in der parent Komponente `CategoryList` die Methode `startSubmit` aufrufen können, benötigen wir ein *ref* auf die `CategoryEdit` Komponente. Zusätzlich müssen wir die Methode `startSubmit` in der `CategoryEdit` Komponente implementieren und bereitstellen, welche die parent Komponente an geeigneter Stelle aufrufen kann. Zuerst erstellen wir Typen für die Props und Refs:

```
1 export type CategoryEditRef = {
2   startSubmit: () => void;
3 }
4
5 type CategoryEditProps = {
6   category: Category;
7   onSubmitted: () => void;
8   ref?: React.Ref<CategoryEditRef>;
9 }
```

Der erste Typ ist wichtig, damit wir bei `useRef` in `CategoryList` den Typ angeben können. Deswegen wird er auch mit `export` exportiert. Der zweite Typ beschreibt die Props der Komponente. Damit wir die Funktion `startSubmit` im ref zurückgeben können, benötigen wir die Funktion `useImperativeHandle`. Es erlaubt uns, eigene Funktionen des Refs zu definieren.[2]

```
1 useImperativeHandle(ref, () => ({
2   startSubmit: () => {
3     formRef.current?.requestSubmit();
4   },
5 }));
```

Unsere Komponente sieht nun so aus:

/src/app/categories/CategoryEdit.tsx

```
1 // Imports
2 export type CategoryEditRef = {
3   startSubmit: () => void;
4 }
5
6 type CategoryEditProps = {
7   category: Category;
8   onSubmitted: () => void;
9   ref?: React.Ref<CategoryEditRef>; // Ref as prop
10 }
11
12 async function handleSubmit(
13   event: FormEvent,
14   setError: Dispatch<SetStateAction<ErrorResponse>>,
15   onSubmitted: () => void
16 ) {
17   event.preventDefault();
18   const response = await editCategory(new FormData(event.target as
   HTMLFormElement));
19   if (isErrorResponse(response)) {
20     setError(response);
21   } else {
22     onSubmitted();
23   }
24 }
25
26 export default function CategoryEdit(props: CategoryEditProps) {
27   const { category, onSubmitted, ref } = props;
28   const formRef = useRef<HTMLFormElement>(null);
29   const [error, setError] = useState<ErrorResponse>(createEmptyErrorResponse());
30
31   // UseImperativeHandle for custom methods exposed to the parent
32   useImperativeHandle(ref, () => ({
33     startSubmit: () => {
34       formRef.current?.requestSubmit();
35     },
36   }));
37
38   useEffect(() => {
39     if (error.message) {
40       alert(error.message);
41     }
42   }, [error]);
43
44   return (
45     <div>
46       <form
```

```
47          onSubmit={(e) => handleSubmit(e, setError, onSubmitted)}
48          ref={formRef}
49          className={styles.categoryEdit}
50        >
51          <input type="hidden" name="guid" value={category.guid} />
52          <div>
53            <div>Name</div>
54            <div>
55              <input type="text" name="name" defaultValue={category.name} required />
56            </div>
57            <div>
58              {error.validations.name && (
59                <span className={styles.error}>{error.validations.name}</span>
60              )}
61            </div>
62          </div>
63          { /* ... */ }
64        </form>
65      </div>
66    );
67  }
```

`CategoryEdit` Komponente ohne `useImperativeHandle`

`useImperativeHandle` soll nur wenn nötig verwendet werden. Es gibt auch Alternativen. So kann man z. B. die `formRef` in der parent Komponente erstellen und als Parameter übergeben. Dies bindet aber die Komponenten stärker aneinander. Oft ist es auch ausreichend, den State als Parameter zu übergeben. Dies nennt sich *lifting state up* und ist ein gängiges Muster in React. Mehr Informationen sind in der React Doku unter Sharing State Between Components abrufbar.

Wenn wir eine `formRef` in der parent Komponente erstellen, können wir sie als Parameter übergeben. Mit `formRef.current?.requestSubmit()` können wir das Formular abschicken. Nachteil: Die parent Komponente muss wissen, dass die `CategoryEdit` Komponente ein Formular enthält. Das ist eine stärkere Kopplung zwischen den Komponenten.

Konkret würde die Version ohne `useImperativeHandle` so aussehen:

CategoryList.tsx (with form ref as parameter)

```
1  "use client";
2  export default function CategoryList({ categories }: { categories: Category[] }) {
3    // Reference to the form element in CategoryEdit component
4    const formRef = useRef<HTMLFormElement>(null);
5    return (
6        { /* ... */ }
7          <ModalDialog
8            onOk={() => formRef.current?.requestSubmit()}>
```

```
 9
10           <CategoryEdit category={selectedCategory}
11             formRef={formRef}
12             onSubmitted={() => setSelectedCategory(null)} />
13         </ModalDialog>
14   );
15 }
```

CategoryEdit.tsx (without useImperativeHandle)

```
 1 type CategoryEditProps = {
 2   category: Category;
 3   onSubmitted: () => void;
 4   formRef?: React.Ref<HTMLFormElement>;
 5 }
 6
 7 export default function CategoryEdit(props: CategoryEditProps) {
 8   const { category, onSubmitted, formRef, ref } = props;
 9   // ...
10     return (
11       { /* ... */ }
12       <form ref={formRef} onSubmit={handleSubmit}>
13       { /* ... */ }
14     )
15 }
```

Source files in der Edit Form App

- CategoryEdit.tsx
- CategoryEdit.module.css

[1] https://react.dev/reference/react/forwardRef

[2] https://react.dev/reference/react/useImperativeHandle