Compound Components with React

Inhalt

Was sind Compound Components?

Compound Components können als ein Pattern angesehen werden, welches den **State** und das **Verhalten** einer Gruppe von Komponenten umschliesst, aber dennoch die Kontrolle des Rendering der einzelnen Teile dem Developer überlässt.

Das Ziel von Compound Components ist es, eine ausdrucksstärkere und flexiblere API für die Kommunikation zwischen der übergeordneten und der untergeordneten Komponenten bereitzustellen.

Ein gutes Beispiel ist das `<select>` in Kombination mit `<option>` in HTML:

```
<select>
  <option value="value1">name1</option>
  <option value="value2">name2</option>
  <option value="value3">name3</option>
</select>
```

Die beiden Elemente kommunizieren im Hintergrund miteinander. Das `<select>` beinhaltet den State und die `<option>`s geben aber vor was alles ausgewählt werden kann. Dies ermöglicht ein expressives Markup und die Optionen müssen nicht über einen Attribut mitgeben werden.

```
<!-- Mit einem Attribut 😱 -->
<select options="name1:value1;name2:value2;name3:value3"></select>
```

Das Problem

Angenommen wir müssen ein Menu erstellen. Das Menu hat einen Button um es zu öffnen und zu schliessen. Es beinhaltet einerseits Links und andererseits Buttons. Beim Klick auf diese sollte die gewünschte Aktion getriggert werden und das Menu sollte wieder geschlossen werden.

Schauen wir verschiedene Implementationsarten an:

Konfiguration

```
function Example() {
  const menuConfig = {
    menuButton: {
      text: "Menu",
    },
    menuList: [{
      kind: "item",
```

```
      text: "Download",
      onClick: () => { /* ... */ }
    }, {
      kind: "link",
      text: "Details anzeigen",
      href: "details"
    }],
    closeOnItemClick: true,
  };

  return (
    <Menu {...menuConfig}> />
  );
}
```

In dem Beispiel erstellen wir ein kompliziertes Konfigurationsobjekt, welches alle Informationen bereits beinhaltet die für das Rendering und für die Funktionalität von allen Komponenten innerhalb des `<Menu>` gebraucht werden. Dies ist ein weit verbreitetes Pattern.

Der State ob das Menu offen ist oder geschlossen, wird von der `<Menu>`-Komponente selbst verwaltet, dazu haben wir die Möglichkeit, dass die Unterkomponenten den State vom `<Menu>` verändern können.

Pros

1. `<Menu>` regelt den internen State selbst, ob es in geöffneten oder geschlossenen Zustand ist

Cons

1. Die Verantwortung für das Rendering aller Unterkomponenten ist in direkter Verantwortung vom `<Menu>`-Komponenten, welches die verschiedenen Komponenten sehr stark miteinender koppelt
2. Die `props` werden durch mehrere Komponenten durchgereicht. Dies wird als "prop drilling" bezeichnet und gilt allgemein als "code smell"
3. Wenn die Daten in einem anderen Format daherkommen, müssen diese zuerst auf die Struktur der Config gemappt werden TODO: Beispiel fürs mapping

Stateverwaltung selber übernehmen

```
function Example() {
  const [isOpen, setIsOpen] = useState(false);

  const open = () => {
    setIsOpen(true);
  }

  const close = () => {
    setIsOpen(false);
  }

  const download = () => {
    // Download stuff ...
    close();
  }
```

```
  return (
    <Menu open={isOpen}>
      <MenuButton onClick={open}>Menu</MenuButton>
      <MenuList>
        <MenuItem onClick={download}>Download</MenuItem>
        <MenuLink to="details" onClick={close}>Details anzeigen</MenuLink>
      </MenuList>
    </Menu>
  );
}
```

In diesem Beispiel müssen wir zwar kein kompliziertes Konfigurationsobjekt erstellen. Da die Komponenten jedoch nicht miteinander verknüpft sind, müssen wir den State des Menu selbst verwalten.

Pros

1. Die Komponenten sind nicht mehr so eng gekoppelt und das `<Menu>` muss sich nicht mehr um das Rendering der Unterkomponenten kümmern
2. Die Datenstruktur spielt keine Rolle mehr TODO: Beispiel für die Datenstruktur

Cons

1. State muss selbst verwaltet werden, je nach Komponenten wird dies sehr kompliziert

Die Lösung

Mit dem Compound Components Pattern können wir diese beiden Ansätze miteinander verbinden.
Der State welchen wir nicht selbst verwalten wollen, ist in dem `<Menu>` -Komponenten verwaltet. Die verschiedenen Komponenten kommunizieren per React Context API und somit sind die Implementationsdetails komplett abgekapselt. Wir können uns also voll und ganz um unsere Implementation kümmern.

```
function Example() {
  const download = () => {
    // Download stuff ...
  }

  return (
    <Menu closeOnItemClick>
      <Menu.Button>Menu</Menu.Button>
      <Menu.List>
        <Menu.Item onClick={download}>Download</Menu.Item>
        <Menu.Link to="details">Details anzeigen</Menu.Link>
      </Menu.List>
    </Menu>
  );
}
```

Sieht doch viel besser aus, oder? 😍

Pros

1. State muss nicht selbst verwaltet werden, je nach Implementation kann aber trotzdem auf diesen zugegriffen werden (per Renderprops)

2. Die Datenstruktur spielt keine Rolle mehr
3. Die Komponenten sind nicht eng miteinander gekoppelt

Hilfreiche Links

- Passing Data Deeply with Context

Beispiel: Accordion

Hier sind verschiedene Nutzungen von einem Accordion, welches als Compound Component erstellt wurde.

Accordion mit statischen Daten

Per Renderprops können wir auch an Daten rankommen, welche vom Accordion verwaltet werden. Dies ist natürlich optional, und man muss nicht auf die Renderprops zugreiffen. Für Infos, wie diese optionalen Renderprops genutzt werden können, könnt ihr die unten verlinkte Demo anschauen.

```
function App() {
  return (
    <Accordion multiple>
      <Accordion.Item id="1">
        {({ id, isActive }) => (
          <>
            <Accordion.Toggle>
              Section {id} {isActive ? "(active)" : null}
            </Accordion.Toggle>
            <Accordion.Panel>
              <p>
                Lorem ipsum dolor sit amet consectetur adipisicing elit.
                Magnam, voluptatum.
              </p>
            </Accordion.Panel>
          </>
        )}
      </Accordion.Item>
      <Accordion.Item id="2">
        <Accordion.Toggle>Section 2</Accordion.Toggle>
        <Accordion.Panel>
          <p>
            Lorem ipsum dolor sit amet consectetur adipisicing elit. Magnam,
            voluptatum.
          </p>
        </Accordion.Panel>
      </Accordion.Item>
      <Accordion.Item id="3">
        <Accordion.Toggle>Section 3</Accordion.Toggle>
        <Accordion.Panel>
          <p>
            Lorem ipsum dolor sit amet consectetur adipisicing elit. Magnam,
            voluptatum.
          </p>
        </Accordion.Panel>
      </Accordion.Item>
```

```
    </Accordion>
  );
}
```

Accordion mit Daten aus einem Array

```
function App() {
  // Die Daten können von überall bezogen werden und in jeglicher Struktur genutzt
werden
  const data = [{
    id: "1",
    title: "Section 1",
    description: (<p>Lorem ipsum 1</p>)
  }, {
    id: "2",
    title: "Section 2",
    description: (<p>Lorem ipsum 2</p>)
  }, {
    id: "3",
    title: "Section 3",
    description: (<p>Lorem ipsum 3</p>)
  }];

  return (
    <Accordion multiple>
      {data.map(entry => (
        <Accordion.Item id={entry.id}>
          <Accordion.Toggle>
            {entry.title}
          </Accordion.Toggle>
          <Accordion.Panel>
            {entry.description}
          </Accordion.Panel>
        </Accordion.Item>
      ))}
    </Accordion>
  );
}
```

Demo 🤯

- Accordion Compound Component (React)

Hilfreiche Links

- React Hooks: Compound Components
- Passing data with a render prop
- Advanced React Component Patterns : Render Props