React Immutable State

Lernziele

- [ ] Verstehen, warum du den Zustand niemals direkt verändern solltest
- [ ] Arbeiten mit verschachtelten Arrays und Objekten im Zustand
- [ ] Den `useImmer` Hook kennen

Zustand niemals mutieren

In der Sitzung **React State 3** haben wir besprochen, wie man mit Objekten und Arrays umgeht, die im Zustand gespeichert sind.

Wie du gelernt hast, kannst du Daten, die im Zustand gespeichert sind, nicht direkt ändern (mutieren). Du musst den Zustand als **nur lesbar** behandeln. Um den Zustand zu ändern, rufst du die Setter-Funktion auf und übergibst den vollständigen nächsten Zustand.

Betrachte ein solches Objekt im Zustand:

```
const [user, setUser] = useState({
  name: "John Doe",
  email: "john@doe.com",
});
```

Du könntest versucht sein, einen Wert im Objekt zu ändern und ihn an die Setter-Funktion zu übergeben.

```
user.email = "john_doe@example.com"; // ❌ direkte Zustandsmutation: Das solltest du nicht tun!
setUser(user);
```

Dieser Code wird nicht wie erwartet funktionieren: Er mutiert das im Zustand gespeicherte Objekt direkt!

Wenn du die Setter-Funktion aufrufst, prüft React, ob sich das Objekt im Zustand geändert hat und ob die Benutzeroberfläche aktualisiert werden muss. Da du das vorherige Zustandsobjekt mutiert hast, ist es gleich dem neuen Zustand, den du an die Setter-Funktion übergeben hast. React erkennt keinen Unterschied und wird die Benutzeroberfläche nicht aktualisieren.

Daher musst du eine Kopie der Daten mit der Spread-Syntax erstellen und die Änderungen an der Kopie vornehmen. So mutierst du das vorherige Zustandsobjekt nicht.

```
setUser({
  ...user,
```

```
  email: "john_doe@example.com",
});
```

Aktualisierung verschachtelter Zustände

Es kann etwas komplizierter werden, wenn du Daten in einem tiefer verschachtelten Zustand ändern musst.

```
const [user, setUser] = useState({
  name: "John Doe",
  contact: {
    email: "john@doe.com",
    phone: {
      mobile: "+001111111111",
      work: "+001234567890",
    },
  },
});
```

Wenn `user.contact.phone.mobile` geändert werden soll, musst du eine Kopie jeder Ebene erstellen.

```
setUser({
  ...user,
  contact: {
    ...user.contact,
    phone: {
      ...user.contact.phone,
      mobile: "+009999999999",
    },
  },
});
```

Dieser Code funktioniert einwandfrei! Allerdings musst du viel Code schreiben, um nur einen einzigen Wert zu ändern.

Die `immer` Bibliothek hilft dir dabei, Werte in tiefer verschachtelten Zuständen zu aktualisieren.

Sie erstellt für dich eine vollständige Kopie des vorherigen Zustands. Diese Kopie ist der `draft` für den nächsten Zustand. Da es sich um eine Kopie handelt, kannst du Mutationen auf jede beliebige Weise vornehmen. Die `immer` Bibliothek sorgt dafür, dass der Zustand entsprechend aktualisiert wird.

Verwendung von `immer` in React: `useImmer` Hook

Der `useImmer` Hook ermöglicht es dir, `immer` einfach in React-Komponenten hinzuzufügen.

- Anstelle von `useState` zum Deklarieren eines Zustands rufst du `useImmer` auf
- Die zurückgegebene Funktion sollte mit `update` anstelle von `set` beginnen.

Das vorherige Beispiel sieht mit dem `useImmer` Hook so aus:

```
// useState → useImmer
// setUser → updateUser
const [user, updateUser] = useImmer({
  name: "John Doe",
  contact: {
    email: "john@doe.com",
    phone: {
      mobile: "+001111111111",
      work: "+001234567890",
    },
  },
});
```

Wenn du die `update` Funktion aufrufst, übergibst du einen Callback. Der Callback erhält einen `draft` für den nächsten Zustand als Parameter. Du kannst Mutationen direkt auf den `draft` anwenden.

```
updateUser((draft) => {
  // Mutiere den draft direkt
  draft.contact.phone.mobile = "+009999999999";
});
```

> 💡 Du findest eine gute Anleitung zu Update-Mustern in den `immer` Dokumentationen.

Arbeiten mit Objekten in Arrays

Die obigen Beispiele konzentrieren sich auf Mutationen in einem Objekt. In vielen Anwendungen wirst du jedoch wahrscheinlich mit Objekten arbeiten, die in Arrays verschachtelt sind.

Dein Zustand könnte eine solche Struktur haben:

```
const [users, setUsers] = useState([
  {
    id: 1,
    name: "John Doe",
    email: "john@doe.com",
  },
  {
    id: 2,
    name: "Jane Doe",
    email: "jane@doe.com",
  },
  {
    id: 3,
    name: "James Doe",
    email: "james@doe.com",
```

```
  },
]);
```

Du kannst ein Update durchführen, um die `email` eines Benutzers mit der `id` von `1` zu ändern:

```
setUsers(
  users.map((user) =>
    user.id === 1
      ? {
          ...user,
          email: "john_doe@example.com",
        }
      : user
  )
);
```

Die gleiche Operation mit der `update` Funktion des `useImmer` Hooks sieht so aus:

```
updateUsers((draft) => {
  const user = draft.find((user) => user.id === 1);
  user.email = "john_doe@example.com";
});
```

Der genaue Code, den du schreiben musst, hängt stark von der Art der Operation (Update, Einfügen, Löschen) und der Struktur der im Zustand gespeicherten Daten ab.

Ob du `immer` verwenden möchtest oder nicht, hängt von deiner persönlichen Präferenz und der Komplexität der Datenstruktur ab. Bei tiefer verschachtelten Strukturen kann die Verwendung von `immer` es dir ermöglichen, einfacheren Code zu schreiben.

Resources

- React docs: Updating Objects in State
- useImmer hook
- Immer: update patterns