19 - React: Interaktivität mit State, Events & API-Daten

Einleitung

- **Themen**: In diesem Skript erwecken wir unsere statischen React-Komponenten zum Leben. Wir führen das fundamentale Konzept des **State** ein, das es Komponenten ermöglicht, sich Daten zu merken und auf Änderungen zu reagieren.
- **Fokus**: Die drei Säulen der interaktiven UI: **State** zur Datenverwaltung (`useState`), **Event-Handling** zur Reaktion auf Benutzerinteraktionen und **Side Effects** (`useEffect`) zum Abrufen von externen Daten, wie z.B. von unserer Django-API.
- **Lernziele**:
  - Verstehen, warum `props` alleine nicht ausreichen und wann `state` benötigt wird.
  - Den `useState`-Hook sicher anwenden, um den Zustand einer Komponente zu verwalten.
  - Auf Benutzer-Events wie Klicks und Eingaben reagieren.
  - Das "Controlled Components"-Muster für Formularfelder verstehen und implementieren.
  - Den `useEffect`-Hook verstehen, um Code nach dem Rendern auszuführen (z.B. für API-Anfragen).
  - Daten von der eigenen Django-API abrufen und im React-Frontend dynamisch anzeigen.

1. Die Grenzen von Props und die Notwendigkeit von State

Im letzten Skript haben wir gelernt, dass **Props** Daten sind, die von einer Elternkomponente an eine Kindkomponente "von oben nach unten" weitergegeben werden. Wichtig dabei ist: **Props sind für die Kindkomponente schreibgeschützt (read-only)**.

Das Problem: Was passiert, wenn eine Komponente Daten benötigt, die sich im Laufe der Zeit durch Benutzerinteraktion ändern?

- Der Text, den ein Benutzer in ein Suchfeld eingibt.
- Ob eine "Mehr erfahren"-Box ein- oder ausgeklappt ist.
- Die Liste der Produkte, die von einer API geladen wurde.

Diese Daten gehören der Komponente selbst. Sie sind ihr interner, privater, veränderbarer Zustand. Dafür brauchen wir **State**.

Definition: **State** ist das "Gedächtnis" einer React-Komponente. Es ist ein Datensatz, den die Komponente besitzt und verändern kann. Die entscheidende Regel in React lautet: **Wenn sich der State einer Komponente ändert, rendert React diese Komponente (und ihre Kinder) automatisch neu**, um die Änderungen auf dem Bildschirm darzustellen.

2. Der `useState` Hook: Das Gedächtnis einer Komponente

Um State in einer Function Component zu verwenden, nutzen wir **React Hooks**. Hooks sind spezielle JavaScript-Funktionen, die mit `use` beginnen und es uns erlauben, uns in React-Features "einzuhaken". Der grundlegendste Hook ist `useState`.

Syntax und Funktionsweisege

1. **Importieren**: Zuerst muss `useState` aus React importiert werden.

   ```
   import { useState } from 'react';
   ```

2. **Aufrufen**: Innerhalb der Komponentenfunktion rufen wir `useState` auf und übergeben ihm den **Anfangswert** (initial value) des Zustands.

   ```
   const [count, setCount] = useState(0);
   ```

Was passiert hier genau?

- `useState(0)`: Wir sagen React: "Erstelle eine neue Zustandsvariable für diese Komponente. Ihr Anfangswert soll `0` sein."
- `useState` gibt ein Array mit genau zwei Elementen zurück:
  1. Die **aktuelle Zustandsvariable** (`count`): Sie enthält den aktuellen Wert des Zustands für den *aktuellen Render-Vorgang*.
  2. Die **Setter-Funktion** (`setCount`): Dies ist die einzige Funktion, mit der wir den Zustand aktualisieren dürfen. Wenn wir sie aufrufen, teilen wir React mit: "Der Zustand hat sich geändert, bitte plane ein neues Rendering der Komponente."

- `const [count, setCount] = ...`: Wir verwenden "Array Destructuring" aus JavaScript, um diesen beiden Elementen direkt Namen zu geben.

Beispiel von einfach bis komplex

- **Einfaches Beispiel: Ein Klick-Zähler**

```jsx
// src/components/Counter.jsx
import { useState } from 'react';

function Counter() {
  // 1. State initialisieren
  const [count, setCount] = useState(0);

  // 2. Eine Funktion, die den State ändert
  function handleIncrement() {
    setCount(count + 1);
  }

  // 3. Im JSX den State anzeigen und die Funktion aufrufen
  return (
    <div>
      <p>Aktueller Zählerstand: {count}</p>
      <button onClick={handleIncrement}>Erhöhen</button>
    </div>
  );
}

export default Counter;
```

Ablauf: Klick -> `handleIncrement` wird aufgerufen -> `setCount(count + 1)` wird aufgerufen -> React rendert die Komponente neu -> `useState` gibt nun den neuen Wert (`1`) für `count` zurück -> die UI zeigt "1" an.

- **Komplexeres Beispiel: Ein- und Ausblenden (Toggle)**

```jsx
// src/components/Accordion.jsx
import { useState } from 'react';

function Accordion({ title, children }) {
  const [isOpen, setIsOpen] = useState(false);

  return (
    <div className="accordion">
      <h3 onClick={() => setIsOpen(!isOpen)} style={{ cursor: 'pointer' }}>
        {title} {isOpen ? '-' : '+'}
      </h3>
      {isOpen && <div>{children}</div>}
    </div>
  );
}

export default Accordion;
```

Hier wird ein Boolean-Zustand verwendet, um bedingt (`{isOpen && ...}`) den Inhalt anzuzeigen.

3. Event-Handling: Auf Benutzer reagieren

Um State-Änderungen auszulösen, reagieren wir auf Events. In JSX werden Event-Handler direkt als Attribute an die Elemente übergeben.

- **Schreibweise**: `camelCase`, z.B. `onClick`, `onChange`, `onSubmit`.
- **Handler**: Man übergibt eine Funktion als Event-Handler.

```
// Direkt eine Arrow-Funktion übergeben
<button onClick={() => console.log('Button geklickt!')}>Klick mich</button>

// Eine Referenz auf eine zuvor definierte Funktion übergeben
function handleClick() {
    console.log('Button geklickt!');
}
<button onClick={handleClick}>Klick mich</button>
```

4. Controlled Components: Das Formular-Muster in React

Ein "Controlled Component" ist ein Formularelement (wie `<input>`), dessen Wert von Reacts State kontrolliert wird. Dies ist das Standardmuster für Formulare in React.

Der Kreislauf:

1. Wir erstellen eine State-Variable, die den Wert des Inputs speichert.
2. Wir binden das `value`-Attribut des Inputs an diese State-Variable.
3. Wir verwenden das `onChange`-Event, um die State-Variable bei jeder Benutzereingabe zu aktualisieren.

Beispiel: Ein synchronisiertes Eingabefeld

```
// src/components/SearchBar.jsx
import { useState } from 'react';

function SearchBar() {
  const [searchTerm, setSearchTerm] = useState('');

  function handleChange(event) {
    // e.target.value enthält den aktuellen Text im Input-Feld
    setSearchTerm(event.target.value);
  }

  return (
    <div>
      <label htmlFor="search">Suche: </label>
      <input
        type="text"
        id="search"
        value={searchTerm}      // 2. Wert an State binden
        onChange={handleChange}   // 3. State bei Änderung aktualisieren
      />
      <p>Aktueller Suchbegriff: <strong>{searchTerm}</strong></p>
    </div>
  );
}

export default SearchBar;
```

Der angezeigte Text unter dem Input ist **immer** synchron mit dem Inhalt des Input-Feldes, weil beide aus derselben "Quelle der Wahrheit" (der `searchTerm`-State-Variable) lesen.

5. Der `useEffect` Hook: Aktionen außerhalb des Renderings

Was ist, wenn wir Code ausführen wollen, der nicht direkt mit dem Rendern der UI zu tun hat? Zum Beispiel Daten von einer API laden, nachdem die Komponente zum ersten Mal angezeigt wird. Solche Aktionen nennt man **Side Effects (Seiteneffekte)**.

Der `useEffect`-Hook ist das Werkzeug dafür. Er führt eine Funktion aus, *nachdem* React die Änderungen an das DOM übergeben hat.

Syntax und Funktionsweise

```
import { useEffect } from 'react';

useEffect(() => {
  // Das ist der "Effekt" - dieser Code wird ausgeführt.
  // z.B. API-Anfrage, Titel des Dokuments ändern
  console.log('Komponente wurde gerendert.');

}, [dependencyArray]); // Das ist das "Abhängigkeits-Array"
```

- **Das Abhängigkeits-Array `[]` ist entscheidend**: Es steuert, *wann* der Effekt erneut ausgeführt wird.
  - `[]` (leeres Array): Der Effekt wird **nur einmal** ausgeführt, direkt nach dem ersten Rendern der Komponente. **Perfekt für initiale Datenabrufe!**
  - `[stateVariable, prop]` (Array mit Werten): Der Effekt wird nach dem ersten Rendern ausgeführt **und immer dann, wenn sich einer der Werte im Array ändert**.
  - **Kein Array** (omitted): Der Effekt wird **nach jedem einzelnen Rendering** ausgeführt. Dies führt oft zu Endlosschleifen und sollte fast immer vermieden werden.

6. Daten von der Django API abrufen und anzeigen

Jetzt kombinieren wir `useState` und `useEffect`, um dynamisch Daten von unserer API zu laden.

Beispiel: Eine Komponente, die Produkte von `/api/products/` lädt

```
// src/components/ProductList.jsx
import { useState, useEffect } from 'react';

function ProductList() {
  // 1. State für die Produkte (anfangs ein leeres Array)
  const [products, setProducts] = useState([]);
  // Optional: State für den Ladezustand
  const [isLoading, setIsLoading] = useState(true);
  // Optional: State für Fehlermeldungen
  const [error, setError] = useState(null);

  // 2. useEffect, um die Daten nach dem ersten Rendern zu laden
  useEffect(() => {
    // Wir definieren eine asynchrone Funktion innerhalb des Effekts
    async function fetchProducts() {
      try {
        const response = await fetch('http://127.0.0.1:8000/api/products/');
        if (!response.ok) {
          throw new Error('Daten konnten nicht geladen werden.');
        }
        const data = await response.json();
        setProducts(data); // 4. State mit den API-Daten aktualisieren
      } catch (error) {
        setError(error.message); // Fehler im State speichern
      } finally {
        setIsLoading(false); // Ladevorgang beenden (egal ob erfolgreich oder nicht)
      }
    }

    fetchProducts(); // Funktion aufrufen
  }, []); // 3. Leeres Array, damit der Effekt nur einmal läuft

  // Bedingtes Rendern basierend auf dem State
  if (isLoading) {
    return <p>Lade Produkte...</p>;
  }

  if (error) {
    return <p>Fehler: {error}</p>;
  }
```

```
  return (
    <div>
      <h2>Unsere Produkte</h2>
      <ul>
        {products.map(product => (
          <li key={product.id}>
            {product.name} - {product.price} €
          </li>
        ))}
      </ul>
    </div>
  );
}

export default ProductList;
```

*`.map()` ist eine Standard-JavaScript-Array-Methode, um aus einem Daten-Array ein Array von JSX-Elementen zu erstellen. *`key={product.id}` ist wichtig. React benötigt einen einzigartigen `key` für jedes Element in einer Liste, um die Performance bei Änderungen zu optimieren. Die `id` aus der Datenbank ist perfekt dafür.

Fazit

- **State ist das Gedächtnis**: Der `useState`-Hook erlaubt es Komponenten, sich veränderliche Daten zu merken. Eine Änderung des States führt zu einem neuen Rendering.
- **Events sind die Auslöser**: Event-Handler wie `onClick` und `onChange` rufen Funktionen auf, die den State aktualisieren und so die Anwendung interaktiv machen.
- **Controlled Components**: Sind das Standardmuster für Formulare in React, bei dem der Wert eines Input-Feldes immer an eine State-Variable gebunden ist.
- **Side Effects**: Der `useEffect`-Hook ist für Aktionen zuständig, die außerhalb des reinen Renderings stattfinden. Mit einem leeren Abhängigkeits-Array `[]` ist er das perfekte Werkzeug, um initiale Daten von einer API zu laden.

Cheat Sheet

- **State hinzufügen**:

```
import { useState } from 'react';
const [value, setValue] = useState(initialValue);
```

- **Event-Handling**:

```
<button onClick={() => setValue(newValue)}>Ändern</button>
<input value={text} onChange={(e) => setText(e.target.value)} />
```

- **Datenabruf nach dem ersten Rendern**:

```
import { useEffect, useState } from 'react';

useEffect(() => {
  // API-Anfrage hier...
  fetch('url...')
    .then(res => res.json())
    .then(data => setState(data));
}, []); // <-- Leeres Array nicht vergessen!
```

Übungsaufgaben

1. **Ein- und Ausklappbarer Text**:
    - Erstelle eine `Collapsible`-Komponente.
    - Sie soll eine State-Variable `isVisible` (Boolean, anfangs `false`) haben.
    - Sie soll einen Button anzeigen (z.B. "Mehr anzeigen" / "Weniger anzeigen"). Ein Klick auf den Button soll den `isVisible`-State umschalten (`!isVisible`).
    - Unter dem Button soll ein Text nur dann angezeigt werden, wenn `isVisible` `true` ist.
2. **Einfacher Taschenrechner**:
    - Erstelle eine Komponente `SimpleCalculator`.
    - Sie soll zwei `<input type="number">`-Felder haben. Verwende "Controlled Components", d.h. erstelle für jedes Input-Feld eine eigene State-Variable (`num1`, `num2`).
    - Zeige unter den Inputs das Ergebnis der Addition an (`num1 + num2`). Beachte, dass die Werte aus den Inputs als Strings kommen können und eventuell mit `parseInt()` oder `Number()` umgewandelt werden müssen.
3. **Live-Suche (simuliert)**:
    - Erstelle eine Liste von Strings (z.B. `['Apfel', 'Banane', 'Ananas', 'Kirsche']`).
    - Erstelle eine Komponente `LiveSearch` mit einem Input-Feld (State: `searchTerm`).
    - Zeige unter dem Input-Feld nur die Früchte aus der Liste an, deren Name den `searchTerm` enthält (case-insensitive).
    - *Tipp*: Filtere das Array bei jedem Rendering: `fruits.filter(fruit => fruit.toLowerCase().includes(searchTerm.toLowerCase()))`.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Wir machen das Frontend unserer Rezept-Plattform dynamisch, indem wir die echten Daten von unserer Django-API laden.

Aufgabe:

1. **Eine `RecipeList`-Komponente erstellen**:
    - Erstelle eine neue Komponente in `src/components/RecipeList.jsx`.
    - Diese Komponente soll, wie im Beispiel `ProductList` oben gezeigt, den kompletten Lebenszyklus für den Datenabruf implementieren:
        - Einen State für die Rezepte (`recipes`, anfangs `[]`).
        - Einen State für den Ladezustand (`isLoading`, anfangs `true`).
        - Einen State für Fehler (`error`, anfangs `null`).
2. **API-Daten abrufen mit `useEffect`**:
    - Innerhalb eines `useEffect`-Hooks (der nur einmal läuft!) eine `fetch`-Anfrage an deinen lokalen Django-API-Endpunkt für Rezepte senden (z.B. `http://127.0.0.1:8000/api/recipes/`).
    - Bei Erfolg die `recipes`-State-Variable mit den erhaltenen Daten füllen und `isLoading` auf `false` setzen.
    - Bei einem Fehler die `error`-State-Variable mit einer Fehlermeldung füllen und `isLoading` auf `false` setzen.
3. **Bedingtes Rendern**:
    - Wenn `isLoading` `true` ist, eine "Lade..."-Nachricht anzeigen.
    - Wenn `error` einen Wert hat, die Fehlermeldung anzeigen.
    - Wenn die Daten geladen sind, eine `<ul>`-Liste mit den Rezepten rendern.
4. **Daten anzeigen**:
    - Verwende die `.map()`-Methode, um über das `recipes`-Array zu iterieren.
    - Gib für jedes Rezept den Titel und Autor aus. Nutze dafür deine `RecipeListItem`-Komponente aus dem letzten Skript, an die du die Daten als Props (`title`, `author`, etc.) weitergibst. Vergiss den `key`-Prop nicht (`key={recipe.id}`).
5. **`RecipeList` in `App.jsx` einbinden**:
    - Ersetze die statische Liste von Rezepten in `App.jsx` durch die neue, dynamische `RecipeList`-Komponente.
    - Starte deinen Django-Server UND deinen React-Dev-Server und stelle sicher, dass die Rezepte aus deiner Datenbank korrekt im Frontend angezeigt werden. (Möglicherweise tritt ein CORS-Fehler auf – das ist normal und wird im letzten Skript behandelt. Fürs Erste können wir ihn ignorieren oder eine Browser-Erweiterung zum Deaktivieren von CORS für die Entwicklung nutzen).