React Data Fetching

Lernziele

- [ ] Verständnis der Vorteile einer Fetching-Bibliothek im Allgemeinen
- [ ] Wissen, wie man mit `SWR` fetcht:
  - `fetcher`-Funktion
  - Lade- und Validierungszustand
  - Fehlerzustand
  - Fetching im Intervall
  - `mutate()`
- [ ] Wissen, wie man lokalen Zustand mit abgerufenen Daten kombiniert

---

Warum eine Data-Fetching-Bibliothek statt `useEffect` und `fetch`?

Bisher konntest du Daten mit dem `useEffect`-Hook abrufen. Dabei musst du viele Dinge selbst handhaben:

- Caching der abgerufenen Daten
- Programmatisches Neuladen (Refetching)
- Implementierung eines Fehler- und Ladezustands
- Fetching im Intervall
- und vieles mehr.

Eine Fetching-Bibliothek wie `SWR` bietet dir Abkürzungen für all diese Aufgaben.

> 📙 Lies mehr über die Features von SWR.

Wie man SWR verwendet

Basis-Datenabruf

Um den `useSWR`-Hook zu verwenden, musst du zuerst eine `fetcher`-Funktion erstellen, die nur ein Wrapper des nativen Fetch ist. Ein einfaches Beispiel empfohlen in den Docs sieht so aus:

```
const fetcher = (...args) => fetch(...args).then((res) => res.json());
```

Dann kannst du den `useSWR`-Hook importieren und ihm zwei Argumente übergeben: die `url`, die du abrufen möchtest, und die `fetcher`-Funktion. `useSWR` gibt ein `data`-Objekt zurück, das du in deinem JSX verwenden kannst.

```
import useSWR from "swr";

const fetcher = (...args) => fetch(...args).then((res) => res.json());
```

```
function Character() {
  const { data } = useSWR("https://swapi.dev/api/people/1", fetcher);

  // Daten rendern
  return <div>Hello {data.name}!</div>; // Hello Luke Skywalker!
}
```

> 💡 Beachte, dass `useSWR` ein Objekt zurückgibt, aus dem du `data` destrukturierst. Aus diesem Grund kannst du das data-Objekt nicht einfach nach Belieben aufrufen, sondern musst es gemäß den Destrukturierungsregeln umbenennen: `{ data: person }`. 📙 Lies mehr über Getting Started in den Docs.

Konfigurieren von SWR

Es kann nützlich sein, eine Anwendungs-weite Konfiguration für `SWR` festzulegen. Dies kannst du tun, indem du ein Konfigurationsobjekt an die `SWRConfig`-Komponente in deiner `App` übergibst (in Next.js ist das `pages/_app.js`). Das folgende Beispiel setzt eine Anwendungs-weite `fetcher`-Funktion und ein Anwendungs-weites `refreshInterval`:

```
import { SWRConfig } from "swr";

const fetcher = (...args) => fetch(...args).then((res) => res.json());

function App() {
  return (
    <SWRConfig
      value={{
        fetcher,
        refreshInterval: 1000,
      }}
    >
      {/* … deine App */}
    </SWRConfig>
  );
}
```

Das Festlegen einer Anwendungs-weiten `fetcher`-Funktion ist sehr praktisch, wenn du dieselbe Fetcher-Funktion an vielen Stellen verwenden möchtest.

> 💡 Alle folgenden Beispiele gehen davon aus, dass eine Anwendungs-weite `fetcher`-Funktion über `SWRConfig` konfiguriert ist.

Lade- und Fehlerzustand

Der `useSWR`-Hook bietet einen `error`, `isLoading` (Laden der Daten zum ersten Mal) und `isValidating` (jedes Mal, wenn Daten (neu) geladen werden) Zustand, die du verwenden kannst, um die entsprechende UI-Ausgabe zu erstellen.

Du kannst sie wie das `data`-Objekt destrukturieren und verwenden, um bedingt JSX zurückzugeben:

```
function Character() {
  const { data, error, isLoading, isValidating } = useSWR(
    "https://swapi.dev/api/people/1"
  );

  if (error) return <div>failed to load</div>;
  if (isLoading) return <div>loading...</div>;

  // Daten rendern
  return (
    <div>
      <span role="img" aria-label={isValidating ? "Validating" : "Ready"}>
        {isValidating ? "🔄" : "✅"}
      </span>
      Hello {data.name}!
    </div>
  );
}
```

Die oben genannte `fetcher`-Funktion `wirft` kein `Error`-Objekt für nicht-`ok`-Antworten. Das Werfen ist erforderlich, damit SWR einen `Fehler` erkennt und ihn in die error-Eigenschaft des vom Hook zurückgegebenen Objekts einfügt.

Du kannst den `fetcher` anpassen, um einen `Error` mit zusätzlichen Informationen zu `throw` (das folgende Beispiel stammt aus den Docs:

```
const fetcher = async (url) => {
  const res = await fetch(url);

  // Wenn der Statuscode nicht im Bereich 200–299 liegt,
  // versuchen wir ihn trotzdem zu parsen und werfen ihn.
  if (!res.ok) {
    const error = new Error("An error occurred while fetching the data.");
    // Füge dem Error-Objekt zusätzliche Informationen hinzu.
    error.info = await res.json();
    error.status = res.status;
    throw error;
  }

  return res.json();
};
```

Diese Funktion wirft einen Fehler mit den Schlüsseln `info` und `status`, wenn der Statuscode der Antwort nicht im Bereich von 200-299 liegt.

> 💡 Der erweiterte `fetcher` oben verwendet zwei Konzepte, die wir noch nicht behandelt haben: den `new`-Operator und das `throw`-Statement. Dies sind fortgeschrittene JS-Features, auf die wir jetzt

> nicht im Detail eingehen müssen, aber das Eintauchen wird dir ein besseres Verständnis von JS als Programmiersprache geben. 📙 Lies mehr über Status Code und Error Object.

Du kannst das `error`-Objekt verwenden, um eine detailliertere Fehlermeldung anzuzeigen (`message` ist der String von `new Error()`):

```
function Character() {
  const { data, error, isLoading } =
useSWR("https://swapi.dev/api/people/1");

  if (error) return <div>{error.message}</div>;
  if (isLoading) return <div>loading...</div>;

  // Daten rendern
  return <div>Hello {data.name}!</div>;
}
```

Fetch im Intervall und Button-Klick

Um die API im Intervall neu zu laden, übergebe einen `refreshInterval`-Wert innerhalb eines Options-Objekts als zusätzliches Argument an den `useSWR`-Hook:. Im folgenden Beispiel lädt `SWR` die API jede Sekunde neu:

```
useSWR("https://swapi.dev/api/people/1", { refreshInterval: 1000 });
```

> 📙 Lies mehr über Revalidate on Interval.

Um Daten programmatisch (z.B. durch Klicken eines Buttons) zu laden, kannst du die `mutate`-Funktion verwenden, die vom `useSWR`-Hook bereitgestellt wird.

```
function Character() {
  const { data, mutate } = useSWR("https://swapi.dev/api/people/1");

  return <RefetchButton onRefetch={() => mutate()}>Refetch
data</RefetchButton>;
}

function RefetchButton({ children, onRefetch }) {
  return (
    <button type="button" onClick={onRefetch}>
      {children}
    </button>
  );
}
```

Daten werden zwischengespeichert

SWR wird die abgerufenen Daten im Speicher des Browsers zwischenspeichern. Das bedeutet, dass, wenn du dieselben Daten zweimal abrufst, beim zweiten Mal die Daten aus dem Cache statt aus dem Netzwerk geladen werden. Dies bedeutet, dass du den useSWR-Hook mehrfach in deiner App verwenden kannst, ohne dir Sorgen machen zu müssen, dass die gleichen Daten mehrfach abgerufen werden.

```
function CharacterName() {
  const { data } = useSWR("https://swapi.dev/api/people/1");
  return <div>Hallo {data.name}!</div>; // Hallo Luke Skywalker!
}

function CharacterHairColor() {
  const { data } = useSWR("https://swapi.dev/api/people/1");
  return <div>Seine Haarfarbe ist {data.hair_color}.</div>; // Seine
Haarfarbe ist blond.
}

function CharacterHeight() {
  const { data } = useSWR("https://swapi.dev/api/people/1");
  return <div>Er ist {data.height} cm groß.</div>; // Er ist 172 cm groß.
}

function App() {
  return (
    <>
      <CharacterName />
      <CharacterHairColor />
      <CharacterHeight />
    </>
  );
}
```

Diese Anwendung wird die Daten nur einmal abrufen, obwohl der useSWR-Hook dreimal verwendet wird.

Zusätzlich, wenn du die Daten manuell mutate (was eine Neubewertung auslöst), wird der Cache aktualisiert und die Daten stehen allen Komponenten zur Verfügung, die den useSWR-Hook mit demselben Schlüssel (URL) verwenden.

Dies gilt sogar, wenn du mutate von einer anderen Komponente aus aufrufst, solange sie denselben Schlüssel (URL) hat:

```
function RevalidateButton() {
  const { mutate } = useSWR("https://swapi.dev/api/people/1");
  return (
    <button type="button" onClick={() => mutate()}>
      Neu bewerten
    </button>
  );
}

// … andere Komponenten
```

```
function App() {
  return (
    <>
      <CharacterName />
      <CharacterHairColor />
      <CharacterHeight />
      <RevalidateButton />
    </>
  );
}
```

SWR Antwort-API

Der useSWR-Hook gibt ein SWR-Antwortobjekt mit den folgenden Eigenschaften zurück:

The `useSWR` hook returns an SWR response object with the following properties:

response property	description
`data`	The data fetched for the given key (URL)
`error`	An error object if the fetcher function threw an error
`isLoading`	`true` if the data is being loaded for the first time
`isValidating`	`true` if there is any request or revalidation loading
`mutate()`	A function to mutate the data

Abgerufene Daten mit lokalem Zustand kombinieren

Mit SWR kontrollierst du den Zustand, der die abgerufenen Daten enthält, nicht selbst. Daher kannst du den Zustand nicht direkt ändern. Das ist eine **gute Sache**, da das Modifizieren von Zustand, der vom Server abgerufen wurde, ein Anti-Pattern ist. Wenn dein Server dir Daten gibt, müssen diese die einzige Quelle der Wahrheit sein.

Wenn du Serverdaten mit lokalem Zustand anreichern möchtest (z.B. indem du eine `isFavorite`-Eigenschaft zu einem Film hinzufügst), kannst du den `useSWR`-Hook verwenden, um die Daten abzurufen, und den `useState`-Hook, um den lokalen Zustand zu verwalten. Der lokale Zustand sollte über eine eindeutige Kennung (wie `id` oder `slug`) mit den Serverdaten verbunden sein.

```
function Movies() {
  /* nehmen wir an, die API gibt eine Liste von Filmen wie folgt zurück:
    [
      {
        id: 1,
        title: "Star Wars",
        year: 1977,
      },
      {
```

```
        id: 2,
        title: "The Empire Strikes Back",
        year: 1980,
      }
    ]
  */
  const { data: moviesData } = useSWR("/api/movies");

  // initialisiere den lokalen Zustand mit einem leeren Array
  const [moviesInfo, setMoviesInfo] = useState([]);

  function handleToggleFavorite(id) {
    setMoviesInfo((moviesInfo) => {
      // finde den Film im Zustand
      const info = moviesInfo.find((info) => info.id === id);

      // wenn der Film bereits im Zustand ist, toggle die isFavorite-
Eigenschaft
      if (info) {
        return moviesInfo.map((info) =>
          info.id === id ? { ...info, isFavorite: !info.isFavorite } :
info
        );
      }

      // wenn der Film nicht im Zustand ist, füge ihn hinzu und setze
isFavorite auf true
      return [...moviesInfo, { id, isFavorite: true }];
    });
  }

  return (
    <ul>
      {moviesData.map(({ id, title, year }) => {
        // finde den Film im Zustand und destrukturiere die isFavorite-
Eigenschaft
        // falls er nicht im Zustand ist, setze isFavorite auf false
        const { isFavorite } = moviesInfo.find((info) => info.id === id)
?? {
          isFavorite: false,
        };

        return (
          <li key={id}>
            {title} ({year})
            <button type="button" onClick={() =>
handleToggleFavorite(id)}>
              {isFavorite
                ? "Von Favoriten entfernen"
                : "Zu Favoriten hinzufügen"}
            </button>
          </li>
        );
      })}
```

```
      </ul>
    );
  }
```

> 💡 Wenn du dieses Muster verwendest, verlässt du dich darauf, dass dein lokaler Zustand ad-hoc erstellt wird. Daher wird dein lokaler Zustands-Array `undefined` bei `find` zurückgeben, wenn der Film nicht im Zustand ist. Deshalb verwenden wir den `??`-Operator, um auf `{ isFavorite: false }` zurückzugreifen, wenn der Film nicht im Zustand ist.

Wenn du `Immer` und `useImmer` verwendest, kannst du den Update-Code ein wenig vereinfachen:

```
function handleToggleFavorite(id) {
  updateMoviesInfo((draft) => {
    // finde den Film im Zustand
    const info = draft.find((info) => info.id === id);

    // wenn der Film bereits im Zustand ist, toggle die isFavorite-
Eigenschaft
    if (info) {
      info.isFavorite = !info.isFavorite;
    } else {
      // wenn der Film nicht im Zustand ist, füge ihn hinzu und setze
isFavorite auf true
      draft.push({ id, isFavorite: true });
    }
  });
}
```

Resources

- SWR Docs