gistrieren neuer Abstimmungen (PUT). In beiden Fällen wollen wir aus der Antwort vom Server noch ein JSON-Objekt erzeugen, das wir per Callback-Funktion an den Initiator des Server Requests zurückgeben.

Um diesen (technischen) Code zentral vorzuhalten, erstellen wir sowohl für Lese- als auch für Schreibzugriffe zwei Hilfsfunktionen. Dafür lege bitte die neue Datei `src/backend.js` an, in der wir die Funktionen zur Serverkommunikation implementieren. Du kannst dort jetzt schon die URL zu unserem API-Server als Konstante hinterlegen:

```
const BACKEND_URL = "http://localhost:3000";
```

Nun implementieren wir die Hilfsfunktionen zum Lesen von Daten als HTTP-*GET*-Aufruf. Die Funktion soll `fetchJson` heißen und erwartet einen Parameter, nämlich den aufzurufenden Pfad (also den Teil der URL, der hinter dem Servernamen kommt). Als zweiten Parameter können (optional) weitere Eigenschaften für den zugrunde liegenden fetch-Aufruf übergeben werden. Die Funktion wird als async-Funktion implementiert und liefert ein Promise zurück, das aufgelöst wird, sobald die Antwort vom Server gekommen ist und in ein JSON-Objekt umgewandelt wurde. Damit die Funktion außerhalb der `backend.js`-Datei sichtbar ist, muss diese exportiert werden. Füge jetzt also bitte folgenden Code in die `backend.js`-Datei ein:

Die Funktion fetchJson (backend.js)

```
export async function fetchJson(path, options) {
  const url = `${BACKEND_URL}${path}`;

  const response = await fetch(url, options);
  if (!response.ok) {
    throw new Error(`Response not OK: ${response.status}`);
  }
  return await response.json();
}
```

Beim Ausführen der Funktion wird der HTTP-Aufruf an den übergebenen Pfad auf unserem API-Server durchgeführt. Das vom Server zurückgeschickte Ergebnis wird in ein JSON-Objekt umgewandelt. Falls der HTTP-Status der Antwort nicht Status OK (200) entspricht, werfen wir einen Fehler, sodass der Aufrufer der Funktion darauf reagieren kann.

Daten auf den Server schreiben

Implementieren wir nun die Funktion für die schreibenden Serverzugriffe, die wir `sendJson` nennen. Analog zur `fetchJson`-Funktion erwartet die Funktion als ersten Parameter den Pfad. Zusätzlich muss der Aufrufer die HTTP-Methode (beispielsweise *POST* oder *PUT*) übergeben und das Objekt, das als Body im JSON-Format an den Server geschickt werden soll. Die Funktion ist ebenfalls als asynchrone Funktion implementiert und liefert ein Promise zurück, das aufgelöst

wird, sobald eine Antwort vom Server zurückgekommen ist und in ein JSON-Objekt umgewandelt wurde. Auch diese Funktion wird exportiert, um sie außerhalb der `backend.js`-Datei sichtbar zu machen. Die vollständige Funktion sieht dann so aus:

Die Funktion sendJson (backend.js)

```
export async function sendJson(method, path, payload = {}) {
  const url = `${BACKEND_URL}${path}`;

  const response = await fetch(url, {
    method: method,
    body: JSON.stringify(payload),
    headers: {
      Accept: "application/json",
      "Content-Type": "application/json"
    }
  });

  if (!response.ok) {
    throw new Error(`Response not OK: ${response.status}`);
  }

  return await response.json();
}
```

Die Funktion konvertiert das mit dem Parameter `payload` übergebene Objekt in einen JSON-String und setzt sie als Body für die Serveranfrage. Außerdem werden für den Serverzugriff zwei HTTP-Header gesetzt: `Accept` und `Content-Type`.

Schritt 2: Votes vom Server abfragen

Um die Kommunikation mit dem Server durchzuführen, legen wir nun die neue Komponente `VoteListPage` an. Dazu gehen wir in mehreren Schritten vor. Sehen wir uns als Erstes an, was nötig ist, um die `vote`-Objekte von unserem Server zu lesen. Dazu schreiben wir zunächst eine Funktion, die die Daten lädt und dann in den Zustand der Komponente setzt. Demnach wird die Komponente neu gerendert, sobald die Daten vorhanden sind, und die Komponente kann die Daten – mithilfe von `VoteController` – anzeigen. Solange noch keine Daten vorhanden sind, wird dem Benutzer eine entsprechende Meldung angezeigt (`VoteLoadingIndicator`).

Die initiale Version von unserer `VoteListPage` sieht wie folgt aus:

Die Komponenten VoteLoadingIndicator und VoteListPage

```
import React from "react";
import { fetchJson, sendJson } from "../backend";

function VoteLoadingIndicator() {
  return (
    <div className="Row VotingRow Spacer">
```

```
      <h1 className="Title">Votes are loading...</h1>
    </div>
  );
}

export default function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }


  if (!allVotes) {
    return <VoteLoadingIndicator />;
  }

  return (
    <VoteController
      allVotes={allVotes} ... />
  );
}
```

Die Komponente wird beim initialen Rendern also den `VoteLoadingIndicator` anzeigen, da noch keine Daten geladen wurden; dazu müsste die `loadVotes`-Funktion aufgerufen werden. Da die Daten automatisch geladen werden sollen, sobald die Komponente das erste Mal verwendet wird, wäre ein naiver Ansatz, die `loadVotes`-Funktion direkt vor dem `return` aufzurufen. Dann würden darin die Daten geladen und sobald sie vom Server kommen, würden sie in den Zustand gesetzt und die Komponente erneut gerendert werden. *Das ist in React allerdings verboten! Funktionskomponenten müssen seiteneffektfrei sein.* Zu Seiteneffekten gehören beispielsweise das Manipulieren des DOM, das Öffnen (oder Schließen) einer Websocket-Verbindung oder eben Serveraufrufe.

Seiteneffekte mit useEffect

Aus diesem Grund gibt es den Hook `useEffect`, mit dem wir uns in den Lebenszyklus der Komponente einhängen können. Mit diesem Hook können wir eine Callback-Funktion angeben, die von React aufgerufen wird, sobald das Rendering der Komponente abgeschlossen ist. Zu diesem Zeitpunkt, *nach* dem Rendering, sind Seiteneffekte erlaubt, sodass wir innerhalb der Callback-Funktion unseren Server-Call durchführen können. Wichtig dabei ist, dass die übergebene Callback-Funktion keine asynchrone Funktion sein darf. Auch aus diesem Grund haben wir das Laden der Votes in eine eigene Funktion ausgelagert (der andere Grund ist, dass wir diese Funktion später noch verwenden werden, um die Daten zu aktualisieren).

Die Funktion zum Laden der Votes (`loadVotes`) wartet, bis die Daten vom Server gekommen sind, und setzt diese mittels `setVotes` in den Zustand, woraufhin sich die Komponente neu rendert.

Mit dem `useEffect`-Hook könnte die Komponente nun wie folgt aussehen (führt allerdings so zu einer Endlosschleife, die wir gleich auflösen):

Fehlerhafte Verwendung von useEffect

```
function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }

  React.useEffect(() => {
    loadVotes();
  });

  if (!allVotes) {
    return <VoteLoadingIndicator />;
  }

  return <VoteController ... />;
}
```

Der Lebenszyklus der Komponente sieht jetzt wie folgt aus:

1. Die Funktion `VoteListPage` wird von React erstmalig aufgerufen (Render-Phase). Dabei wird der `allVotes`-State mit `null` vorbelegt und mit `useEffect` wird die `loadVotes`-Callback-Funktion als Effekt, der nach dem Rendern ausgeführt werden soll, registriert. Da noch keine Daten geladen wurden, wird die `VoteLoadingIndicator`-Komponente zurückgeliefert.
2. React schließt das Rendern ab, schreibt also die benötigten Änderungen in den nativen DOM und führt die mit `useEffect` hinterlegte Callback-Funktion aus (Commit Phase). In unserem Beispiel beginnt also das Laden der Daten in `loadVotes`.
3. Der Server antwortet auf den Request und `loadVotes` setzt die geladenen Umfragen in den State.
4. Die Funktion `VoteListPage` wird erneut von React aufgerufen, da sich der State verändert hat (Render-Phase). Der State enthält nun die geladenen Umfragen und die Funktion gibt die `VoteController`-Komponente zurück.
5. React schließt das Rendern ab und aktualisiert den DOM entsprechend (Commit-Phase).

6. Achtung: Der `useEffect`-Hook wird erneut ausgeführt! Die Daten werden erneut geladen, der Zustand neu gesetzt etc. Wir befinden uns also in einer Endlosschleife. Dieses Problem lösen wir gleich.

Schritt 3: Laden der Daten abschließen

Abhängigkeiten für useEffect angeben

Wie in der Zusammenfassung gesehen, funktioniert unsere Komponente nur scheinbar, da wir uns in einer Endlosschleife befinden: die Komponente wird gerendert, Hook wird ausgeführt, Daten werden geladen und in den Zustand gesetzt, Komponente wird erneut gerendert, Hook wird ausgeführt, Daten werden geladen und in den Zustand gesetzt ... Wir müssen React also noch angeben, unter welchen Bedingungen unsere Callback-Funktion ausgeführt werden soll. Das machen wir mit dem zweiten Parameter von `useEffect`. Darin geben wir in einem Array Werte als *Abhängigkeiten* an. Nur wenn sich eine der Abhängigkeiten, also einer der Werte, zwischen zwei Renderzyklen verändert, wird der Hook erneut ausgeführt. In unserem Beispiel möchten wir, dass der Hook in jedem Fall nur ein einziges Mal nach dem ersten Rendern ausgeführt wird, deswegen geben wir ein leeres Array ein (ein Beispiel für die Verwendung von Werten in dem Array findest du weiter unten, in Abschnitt 7.2):

Beispiel: useEffect mit Abhängigkeiten

```
function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() {
    const votes = await fetchJson("/api/votes");
    setAllVotes(votes);
  }

  React.useEffect(
    () => { loadVotes() },
    [] // nur einmaliges Rendern erzwingen
  );

  if (!allVotes) { return ... }

  return <VoteController votes={allVotes} />;
}
```

Um die neue Komponente `VoteListPage` auch zu verwenden, ersetzen wir in der `index.js` den `VoteController` durch die `VoteListPage`-Komponente. Auch die hartcodierten Beispieldaten können wir dort nun löschen:

```
import VoteListPage from "./components/VoteListPage";
...
```

```
ReactDOM.render(
  <App>
    <VoteListPage />
  </App>,
  document.getElementById("root")
);
```

Wenn du den Client zu diesem Zeitpunkt im Browser öffnest, werden die Votes vom Server geladen. Die entsprechende Anfrage an den Server kannst du dir zum Beispiel in den Chrome Developer Tools ansehen (den aktuellen Stand findest du in `schritte/ 07a_loadvotes`).

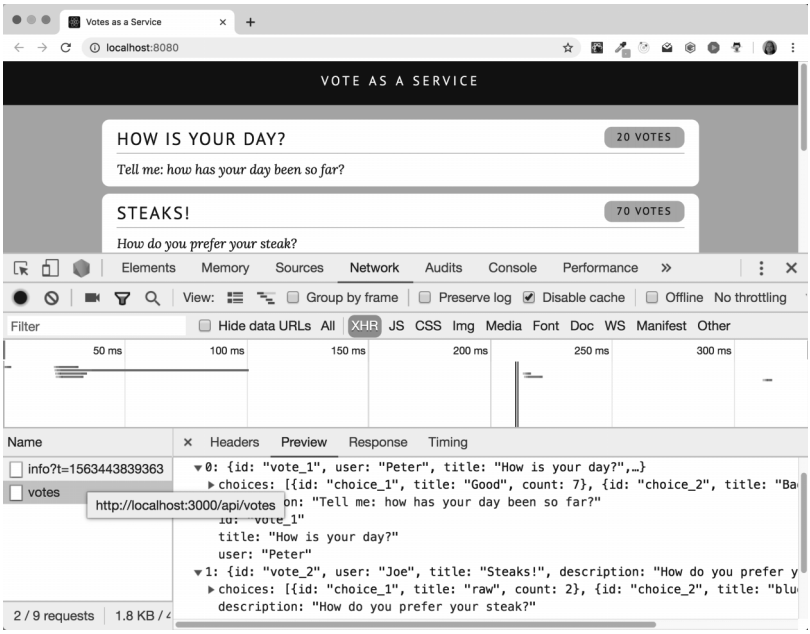

Abb. 7–4
Aufruf des API-Servers in den Chrome Dev Tools dargestellt

Im nächsten Schritt sorgen wir dafür, dass auch die abgegebenen Stimmen für eine Umfrage auf dem Server gespeichert werden.

Schritt 4: Speichern von neuen Umfragen und Stimmabgaben

Das Speichern von Daten passiert infolge eines Events (z.B. Anklicken einer Choice). Daraufhin haben wir bislang die im Client gespeicherte Liste aktualisiert. Nun soll aufgrund des Events ein Serveraufruf erfolgen, der die neue Umfrage speichert bzw. die Stimmabgabe registriert. Da wir uns in einem Event Handler nicht im Renderzyklus befinden, dürfen wir hier Seiteneffekte verwenden, und somit über die API auf den Server zugreifen. Die beiden Event Handler in der `VoteListPage` heißen `addVote` (neue Umfrage) bzw. `registerVote` (Stimmabgabe für

eine Umfrage). Beim Anlegen einer neuen Umfrage kommt die Umfrage als Antwort auf den HTTP-Aufruf vom Server zurück, sodass wir diese direkt in unseren Zustand einfügen können. Bei der Registrierung einer Stimmabgabe gibt uns der Server nur zurück, ob die Registrierung geklappt hat. Aus diesem Grund laden wir hier die Umfragen einfach komplett neu.

Die beiden Event Handler übergeben wir dann dem `VoteController`, den wir im nächsten Schritt leicht anpassen müssen.

Speichern von Daten in der VoteListPage

```
export default function VoteListPage() {
  const [allVotes, setAllVotes] = React.useState(null);

  async function loadVotes() { ... }

  React.useEffect( ... }

  async function registerVote(vote, choice) {
    await sendJson("PUT",
      `/api/votes/${vote.id}/choices/${choice.id}/vote`
    );

    loadVotes();
  }

  async function addVote(vote) {
    const newVote = await sendJson("POST", "/api/votes", vote);
    setAllVotes( currentVotes => [...currentVotes, newVote]);
  }

  if (!allVotes) { ... }

  return (
    <VoteController
      votes={allVotes}
      onRegisterVote={registerVote}
      onSaveVote={addVote}
    />
  );
}
```

Wenn du dir die `addVote`-Funktion ansiehst, erkennst du, dass dort mittels eines POST-Requests die übergebene Umfrage auf dem Server gespeichert und als Ergebnis auch zurückgeliefert wird. Das Ergebnis enthält dann auch die fehlenden Ids, die wir im vorherigen Kapitel noch improvisiert vergeben haben. Die neue Vote wird dann einfach im `allVotes`-State hinzugefügt, so wie das zuvor im `VoteController` passiert ist. Eine Änderung haben wir hier allerdings vorgenommen: Der `setAllVotes`-Funktion wird nicht die Liste der neuen Umfragen übergeben, sondern eine Callback-Funktion. Diese Callback-Funktion ruft

React mit dem jeweils aktuellen Wert des Zustands auf. Der Rückgabewert der Callback-Funktion wird dann von React als neuer Zustand gesetzt.

Nebenläufigkeiten und Zustand

Hintergrund ist, dass der Event Handler (`addVote`) immer nur Zugriff auf den State hat, so wie dieser zu dem Zeitpunkt aussah, als der Event Handler aufgerufen wurde. Im `VoteController` war das auch kein Problem, weil der Event Handler aufgerufen wurde und die Liste im State unmittelbar angepasst hat. In der neuen Variante ist `addVote` aber asynchron. Zwischen dem Senden und Empfangen der Daten vom Server und dem Aufruf von `setAllVotes` kann Zeit vergehen und der State kann sich somit in der Zwischenzeit verändert haben:

Nebenläufigkeit beim Laden von Daten

1. Eine neue Umfrage *U1* wird angelegt und `addVote` wird ausgeführt.
2. Server-Request *S1* startet, um die Umfrage zu speichern; `addVote` »pausiert« so lange.
3. In der Zwischenzeit wird von der Benutzerin eine neue Umfrage *U2* angelegt (die UI ist durch das Warten auf den Server-Request *S1* nicht blockiert!) und `addVote` wird ausgeführt.
4. Server-Request *S2* startet, um die Umfrage *U2* zu speichern.
5. Server-Request *S2* kommt zurück (Server-Request *S1* läuft immer noch) und in `addVote` wird die Liste der Umfragen um die neue Umfrage *U2* erweitert und in den Zustand gesetzt.
6. Nun kommt Server-Request *S1* zurück mit der Umfrage *U1* vom Server. Der Event Handler sieht allerdings noch den `allVotes`-Zustand vom Zeitpunkt seines Aufrufs (Schritt 1) und nicht die in Schritt 5 um *U2* erweiterte Liste. Wenn der Event Handler nun ebenfalls die Liste modifiziert und in den Zustand setzt, ist die Änderung aus Schritt 5 überschrieben.

Durch die Verwendung der Callback-Funktion kann dieses Szenario nicht eintreten, da der Callback-Funktion in Schritt 6 der aktuelle Wert aus dem Zustand übergeben wird (also Stand nach Schritt 5), sodass das Hinzufügen der Umfrage *U1* nicht dazu führt, dass die Umfrage *U2* »gelöscht« wird.

Ob diese Konstellation in deiner Anwendung vorkommt, hängt von mehreren Faktoren ab. Wäre zum Beispiel die UI während des Serverzugriffs blockiert, z.B. mit einer »Bitte warten«-Meldung, kann das Problem so nicht auftreten. Aber du solltest dir zumindest im Klaren darüber sein, dass durch das Arbeiten mit asynchronen APIs grundsätzlich solche Probleme entstehen können.

Den `VoteController` müssen wir noch anpassen, damit er unsere beiden Event Handler auch aufruft. Dazu fügen wir im `VoteController` die Properties `onRegisterVote` und `onSaveVote` hinzu, die Callback-Funktionen mit Event Handlern erwarten. Über diese beiden Properties können wir dann die Event Handler aus der `VoteLoadPage` (`registerVote` und `addVote`) übergeben. Bei den entsprechenden Ereignissen führt der `VoteController` diese dann aus. Da der `VoteController` die Votes nun selbst nicht mehr verändert, sondern nur noch darstellt, können wir dort auch den `allVotes`-State entfernen.

```
function VoteController({ votes, onSaveVote, onRegisterVote }) {

  // allVotes-State entfällt hier
  ...

  function saveVote(vote) {
    closeVoteComposer();

    // onSaveVote ist jetzt Event Handler aus VoteListPage
    onSaveVote(vote);
  }

  return (
    <div>
      { /* onRegisterVote ist Event Handler aus VoteListPage */ }
      <VoteList
        allVotes={votes}
        currentVoteId={currentVoteId}
        onSelectVote={setCurrentVote}
        onDismissVote={unsetCurrentVote}
        onRegisterVote={onRegisterVote}
      />
      ...
    </div>
  );
}
```

Diesen Stand findest du im Verzeichnis `schritte/07b_load_and_save_votes`.

7.2 Seiteneffekte mit useEffect

Das Rendern einer Komponente muss seiteneffektfrei sein. Das bedeutet, dass innerhalb der Komponentenfunktion beispielsweise nicht mit dem nativen DOM gearbeitet werden darf und auch keine Serverzugriffe erfolgen dürfen. Für Seiteneffekte kommt der `useEffect`-Hook zum Einsatz.