20 - React: Interaktive Formulare & CRUD-Operationen mit der API

Einleitung

- **Themen**: In diesem Skript machen wir unsere React-Anwendung vollständig interaktiv. Wir lernen, wie man Daten über Formulare vom Benutzer entgegennimmt und diese an unsere Django-API sendet, um Datenbankeinträge zu erstellen (Create), zu aktualisieren (Update) und zu löschen (Delete).
- **Fokus**: Der komplette **CRUD-Zyklus** aus der Frontend-Perspektive. Wir behandeln das Senden von `POST`-, `PUT`/`PATCH`- und `DELETE`-Anfragen mit der `fetch`-API. Ein zentraler Punkt ist das anschließende Aktualisieren des Zustands in React, damit die Benutzeroberfläche die Änderungen sofort widerspiegelt.
- **Lernziele**:
  - Das "Controlled Components"-Muster zur Handhabung von Formulardaten sicher anwenden.
  - Formular-Absendungen mit dem `onSubmit`-Event verarbeiten und das Standard-Browser-Verhalten mit `event.preventDefault()` unterbinden.
  - `POST`-Anfragen mit `fetch` senden, um neue Daten in der Django-API zu erstellen.
  - `PUT`/`PATCH`- und `DELETE`-Anfragen senden, um bestehende Daten zu bearbeiten oder zu löschen.
  - Den `Authorization`-Header mit dem Authentifizierungs-Token korrekt bei Anfragen mitsenden.
  - Den lokalen React-Zustand (State) nach erfolgreichen API-Aktionen synchronisieren, ohne die Seite neu laden zu müssen.

1. Wiederholung: Controlled Components - Die Basis für Formulare

Wie in Skript 19 gelernt, ist ein "Controlled Component" ein Formularelement, dessen Wert von Reacts State kontrolliert wird. Dies ist unsere Grundlage.

```
// Kurze Wiederholung
import { useState } from 'react';

function MyInput() {
  const [text, setText] = useState('');

  return (
    <input
      type="text"
      value={text} // Wert wird aus dem State gelesen
      onChange={(e) => setText(e.target.value)} // State wird bei jeder Eingabe aktualisiert
    />
  );
}
```

2. Formular-Absendung verarbeiten (`onSubmit`)

Anstatt auf den `onClick`-Event eines Buttons zu hören, ist es die Best Practice, auf den `onSubmit`-Event des `<form>`-Elements zu lauschen. Dies fängt auch andere Submit-Aktionen ab, z.B. das Drücken der Enter-Taste in einem Textfeld.

- `event.preventDefault()`: Dieser Befehl ist **extrem wichtig**. Er verhindert das Standardverhalten des Browsers, bei einer Formular-Absendung die Seite komplett neu zu laden. Das ist das Herzstück einer Single-Page Application.

Beispiel für eine `handleSubmit`-Funktion:

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

function SimpleForm() {
  const [name, setName] = useState('');

  function handleSubmit(event) {
    event.preventDefault(); // Verhindert das Neuladen der Seite
    alert(`Ein Name wurde übermittelt: ${name}`);
    // Hier würde später unsere API-Anfrage stehen
```

```
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Name:</label>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
      />
      <button type="submit">Senden</button>
    </form>
  );
}
```

3. CREATE: Daten mit POST-Anfragen erstellen

Um ein neues Objekt (z.B. ein Produkt) in unserer Django-Datenbank zu erstellen, senden wir eine `POST`-Anfrage an den Listen-Endpunkt unserer API (z.B. `/api/products/`).

Wichtige Bestandteile der `fetch`-Anfrage:

- `method: 'POST'`
- `headers`:
  - `'Content-Type': 'application/json'`: Teilt dem Server mit, dass wir Daten im JSON-Format senden.
  - `'Authorization': 'Token <dein_token>'`: Der Authentifizierungs-Header, falls der Endpunkt geschützt ist.
- `body`: Die Daten, die wir senden wollen, umgewandelt in einen JSON-String mit `JSON.stringify()`.

Beispiel: Ein vollständiges Formular zum Hinzufügen von Produkten

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

function AddProductForm() {
  const [name, setName] = useState('');
  const [price, setPrice] = useState('');

  // Annahme: Der Token wurde nach dem Login gespeichert
  const authToken = 'HIER_DEINEN_AUTH_TOKEN_EINFÜGEN';

  async function handleSubmit(event) {
    event.preventDefault();

    const productData = {
      name: name,
      price: price
    };

    try {
      const response = await fetch('http://127.0.0.1:8000/api/products/', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          'Authorization': `Token ${authToken}`
        },
        body: JSON.stringify(productData)
      });

      if (!response.ok) {
        throw new Error('Fehler beim Erstellen des Produkts');
      }

      const newProduct = await response.json();
      console.log('Erfolgreich erstellt:', newProduct);
```

```
      // Hier müssen wir die UI aktualisieren! (Siehe Abschnitt 6)

    } catch (error) {
      console.error(error);
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      {/* ... Input-Felder für name und price ... */}
    </form>
  );
}
```

4. UPDATE: Daten mit PUT- oder PATCH-Anfragen bearbeiten

Um ein bestehendes Objekt zu aktualisieren, senden wir eine PUT- oder PATCH-Anfrage an den **Detail-Endpunkt** der API (z.B. `/api/products/123/`).

- **PUT vs. PATCH**:
  - PUT: Ersetzt das **gesamte** Objekt. Man muss alle Felder des Objekts mitsenden.
  - PATCH: Aktualisiert nur die **übergebenen** Felder. Dies ist oft flexibler und wird bevorzugt.

Beispiel für eine PATCH-Anfrage:

```
// Innerhalb einer Funktion, die z.B. nur den Preis aktualisiert
async function handlePriceUpdate(productId, newPrice) {
  const response = await fetch(`http://127.0.0.1:8000/api/products/${productId}/`, {
    method: 'PATCH',
    headers: {
      'Content-Type': 'application/json',
      'Authorization': `Token ${authToken}`
    },
    body: JSON.stringify({ price: newPrice })
  });
  // ... Fehlerbehandlung und UI-Update ...
}
```

5. DELETE: Daten mit DELETE-Anfragen löschen

Um ein Objekt zu löschen, senden wir eine DELETE-Anfrage an den **Detail-Endpunkt**. Diese Anfrage hat normalerweise keinen body.

Beispiel für eine DELETE-Anfrage:

```
// Innerhalb einer Funktion, die auf einen "Löschen"-Button reagiert
async function handleDelete(productId) {
  const response = await fetch(`http://127.0.0.1:8000/api/products/${productId}/`, {
    method: 'DELETE',
    headers: {
      'Authorization': `Token ${authToken}`
    }
  });

  if (response.status === 204) { // 204 No Content ist die typische Antwort bei erfolgreichem
DELETE
    console.log('Erfolgreich gelöscht!');
    // Hier müssen wir die UI aktualisieren! (Siehe Abschnitt 6)
  } else {
    console.error('Fehler beim Löschen.');
  }
}
```

6. Wichtig: Den UI-Zustand nach API-Aktionen aktualisieren

Nachdem eine `POST`-, `PUT`- oder `DELETE`-Anfrage erfolgreich war, ist unsere Datenbank auf dem neuesten Stand, aber der **Zustand in unserer React-Anwendung (z.B. die `products`-Liste) ist veraltet**. Wir müssen ihn manuell synchronisieren, ohne die Seite neu zu laden.

Angenommen, wir haben eine Komponente, die eine Produktliste im State hält: `const [products, setProducts] = useState([]);`

- **Nach einem CREATE (`POST`)**: Die API gibt uns das neu erstellte Produkt (inkl. `id`) zurück. Wir fügen dieses zum bestehenden `products`-Array hinzu.

```
// const newProductFromApi = await response.json();
setProducts([...products, newProductFromApi]); // Fügt das neue Produkt am Ende hinzu
```

- **Nach einem UPDATE (`PUT`/`PATCH`)**: Die API gibt das aktualisierte Produkt zurück. Wir suchen das alte Produkt im Array und ersetzen es.

```
// const updatedProductFromApi = await response.json();
setProducts(products.map(p =>
    p.id === updatedProductFromApi.id ? updatedProductFromApi : p
));
```

- **Nach einem DELETE**: Wir haben die `id` des gelöschten Produkts. Wir filtern das `products`-Array, um ein neues Array ohne dieses Produkt zu erstellen.

```
// const productIdToDelete = 123;
setProducts(products.filter(p => p.id !== productIdToDelete));
```

Fazit

- **`onSubmit` & `preventDefault`**: Das Standardmuster, um Formular-Absendungen in React zu handhaben, ohne die Seite neu zu laden.
- **`fetch` mit Optionen**: `fetch` kann durch die Angabe von `method`, `headers` und `body` für alle CRUD-Operationen konfiguriert werden.
- **`Authorization`-Header**: Unerlässlich für die Kommunikation mit geschützten API-Endpunkten.
- **State-Synchronisation**: Der entscheidende Schritt für eine gute User Experience. Nach jeder erfolgreichen schreibenden API-Aktion muss der lokale React-State manuell aktualisiert werden, damit die UI die Änderungen sofort widerspiegelt.

Cheat Sheet

`fetch` für CRUD-Operationen

- **CREATE (`POST`)**:

```
fetch(url, {
    method: 'POST',
    headers: { 'Content-Type': 'application/json', 'Authorization': `Token ...` },
    body: JSON.stringify(dataObject)
});
```

- **UPDATE (`PATCH`)**:

```
fetch(`${url}/${id}/`, {
    method: 'PATCH',
    headers: { 'Content-Type': 'application/json', 'Authorization': `Token ...` },
    body: JSON.stringify(partialDataObject)
});
```

- **DELETE**:

```
fetch(`${url}/${id}/`, {
    method: 'DELETE',
    headers: { 'Authorization': `Token ...` }
});
```

State-Updates für Listen

- **Item hinzufügen**: `setItems([...items, newItem]);`
- **Item aktualisieren**: `setItems(items.map(item => item.id === updatedItem.id ? updatedItem : item));`
- **Item entfernen**: `setItems(items.filter(item => item.id !== idToRemove));`

Übungsaufgaben

Erstelle eine komplette **To-Do-Listen-Komponente**, die mit einer fiktiven API unter `/api/todos/` kommuniziert.

1. **Anzeige & Laden**: Die Komponente soll beim ersten Rendern alle To-Dos von der API laden und anzeigen.
2. **Erstellen (`POST`)**:
   - Ein Formular mit einem Text-Input und einem "Hinzufügen"-Button erstellen.
   - Bei Absenden des Formulars soll ein `POST`-Request an `/api/todos/` gesendet werden.
   - Nach erfolgreicher Erstellung soll das neue To-Do **ohne Neuladen** in der Liste erscheinen.
3. **Löschen (`DELETE`)**:
   - Neben jedem To-Do-Eintrag einen "Löschen"-Button anzeigen.
   - Ein Klick darauf soll einen `DELETE`-Request an `/api/todos/{id}/` senden.
   - Nach erfolgreichem Löschen soll das To-Do **ohne Neuladen** aus der Liste verschwinden.
4. **(Fortgeschritten) Aktualisieren (`PATCH`)**:
   - Neben jedem To-Do eine Checkbox anzeigen.
   - Ein Klick darauf soll den `completed`-Status des To-Dos umschalten, indem ein `PATCH`-Request an `/api/todos/{id}/` mit `{"completed": true/false}` gesendet wird.
   - Der Zustand der Checkbox und ggf. das Styling des To-Dos (z.B. durchgestrichen) sollen sich sofort aktualisieren.

Schüler-Projekt (Eigenständig): Community Recipe Sharing Platform

Wir implementieren die Funktionalität, um neue Rezepte über das React-Frontend zu erstellen.

Aufgabe:

1. **`AddRecipeForm`-Komponente erstellen**:
   - Erstelle eine neue Komponente `src/components/AddRecipeForm.jsx`.
   - Baue ein "Controlled Component"-Formular mit Input-Feldern für `title` und `description` (als `<textarea>`). Jedes Feld sollte seinen eigenen State haben.
2. **`handleSubmit`-Funktion implementieren**:
   - Die Funktion soll das Standard-Formularverhalten mit `event.preventDefault()` unterbinden.
   - Sie soll ein `recipeData`-Objekt aus den State-Variablen zusammenbauen.
   - Sie soll eine `POST`-Anfrage mit `fetch` an deinen `/api/recipes/`-Endpunkt senden.
   - **Wichtig**: Füge den `Authorization: Token <dein_token>`-Header hinzu. Den Token kannst du fürs Erste hartcodiert in einer Variable speichern.
3. **State-Management in der Elternkomponente (`App.jsx` oder `RecipeList.jsx`)**:
   - Die `AddRecipeForm`-Komponente benötigt eine Möglichkeit, die Elternkomponente darüber zu informieren, dass ein neues Rezept hinzugefügt wurde. Übergebe eine Funktion als Prop an das Formular (z.B. `onRecipeCreated`).

    - In der `handleSubmit`-Funktion des Formulars, nachdem die API-Antwort erfolgreich war, rufe diese Prop-Funktion auf und übergib ihr das neue Rezept: `props.onRecipeCreated(newRecipeFromApi)`.
    - In der Elternkomponente (dort, wo die `recipes`-Liste im State gehalten wird), sorgt diese Funktion dann dafür, dass das neue Rezept zum State hinzugefügt wird: `setRecipes([...recipes, newRecipe])`.
4. **Komponente einbinden**:
    - Füge die `AddRecipeForm`-Komponente in `App.jsx` (oder einer passenden Seite) ein und übergib die Callback-Funktion als Prop.
5. **Testen**:
    - Lade die Seite, melde dich (im Geiste, d.h. hol dir einen Token mit Insomnia) an und füge den Token in deinen Code ein.
    - Erstelle ein neues Rezept über das Formular. Es sollte nach dem Absenden sofort und ohne Neuladen der Seite in deiner Rezeptliste erscheinen.