DIGA4015A:
Interactive Media 4A | React Exercises Solutions (Extra)

Note:

This is **one** method of completing the following exercises, not the only method. How you choose to answer will depend on how you interpret the question, but the logic should remain roughly the same.

*I have left out all the App.js code needed to render these components.

Exercise 1: Fetching Data with Pagination

You are building a table component that fetches data from an API. The API supports pagination, and you want to **fetch new data whenever the user changes the page** (this is a standard feature in applications that display large datasets, like dashboards, e-commerce product lists, or admin panels).

Task:

1. Create a table component with pagination controls (e.g., "*Next*" and "*Previous*" buttons).
2. Use `useEffect` to fetch data from an API whenever the page changes.
3. Display the fetched data in the table.
4. Handle loading and error states.

Hints:

- Use `useState` to manage the current page and the fetched data.
- Use `useEffect` to fetch data when the page changes.
- You can use the API URL (*https://jsonplaceholder.typicode.com/posts?_page=1&_limit=10*) which limits the data response to 10 entries. *consider the 'page' variable in the API URL.

```
import React, { useState, useEffect } from "react";


function PaginatedTable() {
  const [data, setData] = useState([]);
  const [page, setPage] = useState(1);
```

```
const [loading, setLoading] = useState(false);
const [error, setError] = useState(null);

useEffect(() => {
  const fetchData = async () => {
    setLoading(true);
    setError(null);
    try {
      const response = await fetch(
        `https://jsonplaceholder.typicode.com/posts?_page=${page}&_limit=10`
      );
      if (!response.ok) throw new Error("Failed to fetch data");
      const result = await response.json();
      setData(result);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  };

  fetchData();
}, [page]);

return (
  <div>
    <table>
      <thead>
        <tr>
          <th>ID</th>
          <th>Title</th>
        </tr>
      </thead>
      <tbody>
        {data.map((item) => (
          <tr key={item.id}>
            <td>{item.id}</td>
            <td>{item.title}</td>
          </tr>
        ))}
```

```
      </tbody>
    </table>
    <div>
      <button
        onClick={() => setPage((prev) => Math.max(prev - 1, 1))}
        disabled={page === 1}
      >
        Previous
      </button>
      <span>Page {page}</span>
      <button onClick={() => setPage((prev) => prev + 1)}>Next</button>
    </div>
    {loading && <p>Loading...</p>}
    {error && <p style={{ color: "red" }}>{error}</p>}
  </div>
 );
}

export default PaginatedTable;
```

Exercise 2: Debounced Search Input

You are building a search feature for a large dataset (e.g. Netflix). To optimise performance, you want to debounce (*Debouncing a function ensures that it doesn't get called too frequently*.) the search input so that the API call is only made after the user has stopped typing for 500ms.

Task:

1. Create a search input field.
2. Use `useEffect` to debounce the input value.
3. Log the debounced value to the console (simulating an API call).

Hints:

- Use `useState` to manage the input value.
- Use `useEffect` to set up a debounce mechanism with `setTimeout`.
- Clear the timeout on clean-up to avoid memory leaks.

```
import React, { useState, useEffect } from "react";

function DebouncedSearch() {
  const [inputValue, setInputValue] = useState("");
  const [debouncedValue, setDebouncedValue] = useState("");

  useEffect(() => {
    const handler = setTimeout(() => {
```

```
      setDebouncedValue(inputValue);
    }, 500);

    return () => {
      clearTimeout(handler);
    };
  }, [inputValue]);

  useEffect(() => {
    console.log("API call with:", debouncedValue);
  }, [debouncedValue]);

  return (
    <div>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
        placeholder="Search..."
      />
    </div>
  );
}

export default DebouncedSearch;
```

Exercise 3: Auto-Save Form Data

You are building a form where users can input data. To improve the user experience, you want to implement an auto-save feature that saves the form data to local storage every 2 seconds after the user stops typing.

Task:

1. Create a form with a text input field.
2. Use `useEffect` to implement an auto-save feature that saves the input value and displays it in the console (if you want, you can learn about the use of **local storage** and save the values there).
3. Use debouncing to ensure the save operation only occurs after the user has stopped typing for 2 seconds.
4. Optional: Load the saved data from local storage when the component mounts.

Hints:

- Use `useState` to manage the input value.
- Use `useEffect` to implement the debounced auto-save functionality.

- Use `useEffect` to load the saved data from local storage when the component mounts.
- Clean up the debounce timer in the `useEffect` clean-up function.

```
import React, { useState, useEffect } from "react";

function AutoSaveForm() {
  const [inputValue, setInputValue] = useState("");

  // This loads saved data from local storage when the component mounts
  useEffect(() => {
    const savedValue = localStorage.getItem("autoSaveData");
    if (savedValue) {
      setInputValue(savedValue);
    }
  }, []);

  // Autosave the input value to local storage after debouncing
  useEffect(() => {
    const handler = setTimeout(() => {
      localStorage.setItem("autoSaveData", inputValue);
      console.log("Data saved:", inputValue);
    }, 2000);

    return () => {
      clearTimeout(handler);
    };
  }, [inputValue]);

  return (
    <div>
      <h2>Auto-Save Form</h2>
      <input
        type="text"
        value={inputValue}
        onChange={(e) => setInputValue(e.target.value)}
        placeholder="Type something..."
      />
      <p>Current Input: {inputValue}</p>
```

```
    </div>
  );
}

export default AutoSaveForm;
```