

Bài Thực Hành: Xây Dựng Ứng Dụng Quản Lý Học Sinh (React, Express, MongoDB)

Thời lượng: 3 giờ.

Mô tả: Trong bài thực hành này, sinh viên sẽ xây dựng một ứng dụng web quản lý học sinh trong một lớp học, với các chức năng cơ bản: thêm học sinh, sửa thông tin, xóa và hiển thị danh sách học sinh. Ứng dụng được phát triển theo mô hình **MERN stack** gồm **React** (frontend), **Express.js** (backend RESTful API), **MongoDB** (database, sử dụng Docker) và **Axios** (giao tiếp HTTP giữa frontend-backend). Sau mỗi bài tập, ứng dụng sẽ hoàn thiện thêm một chức năng mới.

Mục tiêu

- Củng cố kiến thức về xây dựng **RESTful API** với Express và MongoDB.
- Luyện tập sử dụng **Axios** để trao đổi dữ liệu giữa React frontend và Express backend.
- Hiểu cách tổ chức mã nguồn React cho ứng dụng CRUD (Create, Read, Update, Delete).
- Thực hành cài đặt **MongoDB** bằng Docker Compose và kết nối cơ sở dữ liệu với ứng dụng Node.js.
- Hoàn thiện một ứng dụng web đầy đủ tính năng CRUD (quản lý học sinh) không yêu cầu đăng nhập.

Dưới đây là các bài tập cụ thể cần hoàn thành:

Bài 1: Thiết lập Dự án & Hiển thị Danh sách Học sinh

Yêu cầu

Sinh viên tiến hành khởi tạo dự án **React** và **Express**, thiết lập cơ sở dữ liệu **MongoDB** bằng Docker, và xây dựng chức năng **hiển thị danh sách học sinh**. Kết thúc bài 1, ứng dụng có thể kết nối đến cơ sở dữ liệu (ban đầu có thể chưa có dữ liệu) và hiển thị được danh sách học sinh (danh sách có thể trống lúc đầu) trên giao diện web.

Hướng dẫn thực hiện

- **Bước 1: Cài đặt dự án Frontend (React)** – Tạo ứng dụng React mới (sử dụng create-react-app hoặc Vite). Ví dụ với Create React App:

```
npx create-react-app student-management
cd student-management
npm start
```

Xác nhận ứng dụng React chạy tại <http://localhost:3000> (hiển thị trang mặc định của React).

- **Bước 2: Cài đặt dự án Backend (Express)** – Tạo một thư mục cho backend, khởi tạo Node.js project và cài đặt Express:

```
mkdir backend && cd backend
npm init -y
npm install express mongoose cors body-parser
```

Tạo tệp index.js (hoặc server.js) và thiết lập một máy chủ Express cơ bản chạy trên một cổng (ví dụ: 5000). Sử dụng middleware cors cho phép frontend truy cập API và express.json() (hoặc body-parser) để parse JSON request.

- **Bước 3: Thiết lập MongoDB với Docker Compose** – Tạo tệp docker-compose.yml trong thư mục backend để chạy MongoDB. Cấu hình Docker Compose để chạy container MongoDB, map cổng **27017** và mount volume để lưu dữ liệu bên ngoài. Ví dụ nội dung:

```
version: '3'
services:
  mongodb:
    image: mongo:latest
    container_name: student-mongo
    ports:
      - "27017:27017"
    volumes:
      - ./mongodbddata:/data/db
```

Chạy lệnh docker-compose up -d để khởi động MongoDB. (Thư mục mongodbddata sẽ lưu trữ dữ liệu của MongoDB trên máy host[1]).

- **Bước 4: Kết nối Express với MongoDB** – Sử dụng **Mongoose** để kết nối đến cơ sở dữ liệu. Trong tệp index.js của backend, import mongoose và gọi mongoose.connect() với URL kết nối MongoDB. Ví dụ:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/student_db')
  .then(() => console.log("Đã kết nối MongoDB thành công"))
  .catch(err => console.error("Lỗi kết nối MongoDB:", err));
```

Lưu ý: Sử dụng mongodb://localhost:27017/<tên_database> để kết nối đến MongoDB cục bộ (27017 là cổng mặc định)[2]. Sau khi gọi mongoose.connect(...), kiểm tra console để chắc chắn kết nối thành công.

- **Bước 5: Tạo Mongoose Schema và Model cho Student – Định nghĩa một model "Student" (Học sinh) với các trường thông tin, ví dụ: tên, tuổi, lớp. Tạo một tệp Student.js trong thư mục backend với nội dung:**

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;
const studentSchema = new Schema({
  name: { type: String, required: true },
  age: { type: Number, required: true },
  class: { type: String, required: true }
}, { collection: 'students' });
module.exports = mongoose.model('Student', studentSchema);
```

(Có thể tùy chỉnh tên trường hoặc bổ sung trường khác nếu muốn, nhưng name, age, class** là tối thiểu cho mỗi học sinh).

- **Bước 6: Tạo API GET danh sách học sinh** – Trong tệp index.js (hoặc tạo file routes riêng), thiết lập endpoint GET để lấy danh sách học sinh. Ví dụ:

```
const Student = require('./Student');
app.get('/api/students', async (req, res) => {
  try {
    const students = await Student.find();
    res.json(students);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

Đảm bảo prefix route API (ví dụ dùng tiền tố /api cho các route). Kiểm tra nhanh bằng cách chạy server (node index.js) và dùng trình duyệt hoặc cURL truy cập <http://localhost:5000/api/students> (nếu trả về mảng JSON rỗng là đúng mong đợi).

- **Bước 7: Hiển thị danh sách học sinh trên React** – Chỉnh sửa giao diện React để gọi API và hiển thị dữ liệu. Sử dụng axios trên frontend: cài đặt axios (npm install axios) và import vào component chính (ví dụ App.js). Trong App.js, dùng hook useEffect để fetch dữ liệu từ API khi component load:

```
import axios from 'axios';
import { useState, useEffect } from 'react';
// ...
const [students, setStudents] = useState([]);
```

```
useEffect(() => {
  axios.get('http://localhost:5000/api/students')
    .then(response => setStudents(response.data))
    .catch(error => console.error("Lỗi khi fetch danh sách:", error));
}, []);
```

Sau đó, hiển thị danh sách students trong JSX (ví dụ sử dụng <table> hoặc). Mỗi học sinh hiển thị các cột: Họ tên, Tuổi, Lớp.

- **Bước 8: Chạy thử và kiểm tra** – Khởi động backend (chạy node index.js hoặc npm start nếu có script) và đảm bảo **MongoDB container** đang chạy. Sau đó trong thư mục React, chạy npm start. Mở <http://localhost:3000> trên trình duyệt. Kết quả mong đợi: Trang web hiển thị tiêu đề (ví dụ "Danh sách học sinh") và một bảng rỗng (hoặc thông báo "Chưa có học sinh nào") vì chưa có dữ liệu. Nếu không có lỗi CORS hay kết nối, coi như hoàn thành bài 1.

Bài 2: Thêm Chức năng Thêm Học sinh Mới

Yêu cầu

Bổ sung chức năng **thêm học sinh** vào hệ thống. Cụ thể, cần có một **form nhập liệu trên frontend** để người dùng nhập thông tin học sinh mới (tên, tuổi, lớp) và khi bấm nút "Thêm", dữ liệu sẽ gửi lên **API backend** để tạo mới một học sinh trong cơ sở dữ liệu. Sau khi thêm thành công, danh sách học sinh trên giao diện sẽ cập nhật hiển thị thêm học sinh vừa tạo.

Hướng dẫn thực hiện

- **Bước 1: Tạo API thêm học sinh (HTTP POST)** – Trên backend Express, tạo một route mới để xử lý yêu cầu thêm học sinh. Ví dụ trong index.js:

```
app.post('/api/students', async (req, res) => {
  try {
    const newStudent = await Student.create(req.body); // tạo document mới từ dữ
    liệu gửi lên
    res.status(201).json(newStudent);
  } catch (e) {
    res.status(400).json({ error: e.message });
  }
});
```

Route này nhận JSON (body) chứa thông tin học sinh và sử dụng Student.create (hoặc new Student(req.body).save()) để lưu vào MongoDB. Trả về dữ liệu học sinh vừa tạo kèm mã trạng thái 201 (Created).

- **Bước 2: Tạo giao diện Form thêm học sinh** – Trên frontend (React), thêm một form cho phép nhập thông tin học sinh. Bạn có thể tạo một component riêng (ví dụ AddStudentForm) hoặc viết trực tiếp trong App.js. Form gồm các trường: **Họ tên**, **Tuổi**, **Lớp** và nút "Thêm học sinh". Sử dụng state để quản lý dữ liệu nhập vào (ví dụ dùng useState cho mỗi trường hoặc dùng useReducer để quản lý form). Ví dụ đơn giản sử dụng hooks trong cùng App.js:

```
const [name, setName] = useState("");
const [age, setAge] = useState("");
const [stuClass, setStuClass] = useState("");
...
<form onSubmit={handleAddStudent}>
  <input type="text" placeholder="Họ tên" value={name} onChange={e =>
setName(e.target.value)} required />
  <input type="number" placeholder="Tuổi" value={age} onChange={e =>
setAge(e.target.value)} required />
  <input type="text" placeholder="Lớp" value={stuClass} onChange={e =>
setStuClass(e.target.value)} required />
  <button type="submit">Thêm học sinh</button>
</form>
```

(**Mẹo:** Có thể dùng thư viện UI như **Bootstrap**, **Ant Design**... để form đẹp hơn, nhưng không bắt buộc).

- **Bước 3: Gửi yêu cầu thêm học sinh từ Frontend** – Cài đặt hàm xử lý sự kiện khi submit form (ví dụ handleAddStudent). Bên trong, sử dụng axios.post để gửi dữ liệu đến API /api/students.

```
const handleAddStudent = (e) => {
  e.preventDefault();
  const newStu = { name, age: Number(age), class: stuClass };
  axios.post('http://localhost:5000/api/students', newStu)
    .then(res => {
      console.log("Đã thêm:", res.data);
      // Cập nhật state students để hiển thị luôn học sinh mới:
      setStudents(prev => [...prev, res.data]);
      // Xóa nội dung form sau khi thêm thành công
      setName(""); setAge(""); setStuClass("");
    })
    .catch(err => console.error("Lỗi khi thêm:", err));
};
```

Sau khi nhận phản hồi từ server (res.data là object học sinh vừa được lưu, bao gồm `_id` sinh ra từ MongoDB), ta thêm nó vào danh sách students hiện tại để cập nhật giao diện. Xóa trắng form để sẵn sàng cho lần nhập tiếp.

- **Bước 4: Hiện thị thông báo (tùy chọn)** – Có thể bổ sung thông báo đơn giản cho người dùng biết khi thêm thành công. Ví dụ: hiện dòng chữ "Thêm học sinh thành công!" trong vài giây, hoặc highlight hàng vừa thêm trong bảng. (Phần này không bắt buộc, tùy thời gian cho phép).
- **Bước 5: Kiểm thử chức năng thêm** – Chạy server backend và frontend. Trên trang web, thử nhập thông tin vào form "Thêm học sinh" và bấm nút. Kiểm tra:
 - Dữ liệu gửi lên API thành công (xem log backend hoặc network request).
 - Bản ghi mới được lưu trong MongoDB (có thể vào container Mongo, dùng Compass hoặc Mongo shell để xác minh).
 - Giao diện React cập nhật danh sách hiển thị thêm học sinh mới vừa nhập.

Bài 3: Thêm Chức năng Chỉnh Sửa Thông Tin Học sinh

Yêu cầu

Bổ sung chức năng **sửa thông tin học sinh**. Cho phép người dùng cập nhật các thông tin của một học sinh có sẵn (ví dụ chỉnh sửa tên hoặc tuổi). Thực hiện bằng cách cung cấp giao diện để chỉnh sửa (có thể là một trang/chế độ chỉnh sửa riêng khi nhấn nút "Edit" cạnh mỗi học sinh) và tương ứng gọi **API cập nhật (PUT)** trên backend để lưu thay đổi vào cơ sở dữ liệu. Sau khi cập nhật, danh sách trên giao diện phải hiển thị thông tin mới của học sinh.

Hướng dẫn thực hiện

- **Bước 1: Tạo API cập nhật học sinh (HTTP PUT)** – Trên backend, thêm route để cập nhật học sinh theo ID. Ví dụ:

```
app.put('/api/students/:id', async (req, res) => {
  try {
    const updatedStu = await Student.findByIdAndUpdate(
      req.params.id,
      req.body,
      { new: true }
    );
    if (!updatedStu) {
      return res.status(404).json({ error: "Student not found" });
    }
    res.json(updatedStu);
  } catch (err) {
```

```

    res.status(400).json({ error: err.message });
  }
});

```

Route này tìm document theo req.params.id và cập nhật bằng dữ liệu mới (req.body). Tùy chọn { new: true } để kết quả trả về là document sau khi update. Nếu ID không tồn tại, trả về 404.

- **Bước 2: Giao diện cho phép sửa thông tin** – Trên frontend, quyết định cách triển khai giao diện chỉnh sửa:
- **Cách 1: Tạo một trang hoặc component EditStudent riêng.** Khi người dùng bấm "Sửa" trên một dòng học sinh, chuyển hướng đến trang chỉnh sửa (ví dụ URL /edit/<id>). Trang này sẽ lấy thông tin hiện có của học sinh (có thể gọi API GET /api/students/:id để lấy chi tiết) và hiển thị vào một form cho phép sửa. Khi submit sẽ gọi API PUT để lưu.
- **Cách 2: Sử dụng cùng một form trên trang chính** nhưng ở chế độ "chỉnh sửa". Ví dụ: khi bấm "Sửa", điền các thông tin của học sinh đó vào form (thay vì form trống như khi thêm), và nút submit sẽ thực hiện cập nhật thay vì tạo mới. Cách này phức tạp hơn một chút trong quản lý trạng thái, nhưng tránh phải tạo trang mới.

Sinh viên có thể chọn một trong hai cách trên tùy kiến thức đã học. Ở đây hướng dẫn theo **Cách 1 (tách trang Edit)** để dễ quản lý: - Sử dụng **React Router**: Đảm bảo đã có react-router-dom trong dự án. Cấu hình route cho component EditStudent, ví dụ:

```

<Routes>
  <Route path="/" element={<HomePage />} />
  <Route path="/edit/:id" element={<EditStudent />} />
</Routes>

```

Mỗi hàng trong bảng danh sách học sinh, thêm nút hoặc link "Sửa". Khi bấm, dùng navigate("/edit/"+id) (với hook useNavigate) hoặc dùng Link để chuyển đến route chỉnh sửa.

- Tạo component EditStudent (hoặc sử dụng một state trong cùng component để hiển thị form chỉnh sửa). Trong component này, dùng useEffect để fetch thông tin hiện tại của học sinh từ API (gọi GET /api/students/:id) và hiển thị lên form giống như form thêm nhưng điền sẵn giá trị.

```

// Giả sử nhận id qua props hoặc useParams
const { id } = useParams();
useEffect(() => {
  axios.get(`http://localhost:5000/api/students/${id}`)

```

```

.then(res => {
  setName(res.data.name);
  setAge(res.data.age);
  setStuClass(res.data.class);
})
.catch(err => console.error(err));
}, [id]);

```

- **Bước 3: Gửi yêu cầu cập nhật từ Frontend** – Tương tự chức năng thêm, ta viết hàm xử lý submit cho form chỉnh sửa. Sử dụng `axios.put` gửi dữ liệu mới đến API `/api/students/:id`. Ví dụ:

```

const handleUpdate = (e) => {
  e.preventDefault();
  axios.put(`http://localhost:5000/api/students/${id}`, {
    name, age: Number(age), class: stuClass
  })
  .then(res => {
    console.log("Đã cập nhật:", res.data);
    // Có thể điều hướng về trang chủ hoặc cập nhật state global
    navigate("/");
  })
  .catch(err => console.error("Lỗi khi cập nhật:", err));
};

```

Ở đây sau khi cập nhật thành công, gọi `navigate("/")` để quay về trang danh sách. Nếu không dùng router, bạn có thể cập nhật trực tiếp mảng `students` trong state tương ứng (tìm đúng đối tượng và thay đổi).

- **Bước 4: Cập nhật giao diện danh sách sau khi sửa** – Nếu quay lại trang danh sách, cần thấy thông tin học sinh đã được thay đổi. Cách đơn giản: sau khi `navigate` về trang chủ, có thể gọi lại API lấy danh sách (như đã làm trong `useEffect` của `HomePage`) để đảm bảo dữ liệu mới nhất được hiển thị. Hoặc nếu bạn quản lý danh sách học sinh ở trạng thái chung (ví dụ dùng `Context` hoặc nâng state lên `App`), thì có thể cập nhật trực tiếp vào state trước khi `navigate`.
- **Bước 5: Kiểm thử chức năng sửa** – Thử thêm vài học sinh (bài 2) rồi tiến hành sửa: Nhấn nút "Sửa" trên giao diện cho một học sinh, thay đổi một vài thông tin, gửi cập nhật. Đảm bảo:
 - API trả về kết quả thành công (HTTP 200 và dữ liệu đã thay đổi).
 - Dữ liệu trong MongoDB thực sự được cập nhật.

- Giao diện sau khi trở về (hoặc ngay tại chỗ) hiển thị thông tin mới của học sinh đó.

Bài 4: Thêm Chức năng Xóa Học sinh

Yêu cầu

Hoàn thiện chức năng cuối cùng của ứng dụng: **xóa học sinh**. Yêu cầu: Mỗi học sinh trong danh sách có nút "Xóa"; khi bấm, hệ thống sẽ xóa học sinh đó khỏi cơ sở dữ liệu (gọi **API DELETE** trên backend) và cập nhật lại danh sách trên giao diện. (Không yêu cầu xác nhận khi xóa ở mức cơ bản, nhưng khuyến khích thông báo/confirm để tránh xóa nhầm).

Hướng dẫn thực hiện

- **Bước 1: Tạo API xóa học sinh (HTTP DELETE)** – Thêm route trên backend để xóa theo ID. Ví dụ:

```
app.delete('/api/students/:id', async (req, res) => {
  try {
    const id = req.params.id;
    const deleted = await Student.findByIdAndDelete(id);
    if (!deleted) {
      return res.status(404).json({ error: "Student not found" });
    }
    res.json({ message: "Đã xóa học sinh", id: deleted._id });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

API này xóa document khỏi DB và trả về thông báo đơn giản hoặc ID của bản ghi vừa xóa để client biết.

- **Bước 2: Tích hợp nút "Xóa" trên giao diện** – Trong bảng danh sách học sinh trên React, thêm nút "Xóa" cho mỗi dòng (cạnh nút Sửa). Sử dụng sự kiện onClick để gọi hàm xử lý xóa. Ví dụ:

```
<td><button onClick={() => handleDelete(s._id)}>Xóa</button></td>
```

Tạo hàm handleDelete nhận vào ID học sinh.

- **Bước 3: Gửi yêu cầu xóa từ frontend** – Trong handleDelete, sử dụng `axios.delete` để gọi API xóa. Sau khi nhận phản hồi thành công, cập nhật lại danh sách students trên state. Có hai cách:

- Gọi lại API GET danh sách tất cả học sinh để lấy dữ liệu mới.
- Hoặc tự xóa phần tử khỏi mảng students trong state để nhanh hơn.

Cách 2 hiệu quả và đủ dùng cho dữ liệu ít:

```
const handleDelete = (id) => {
  if (!window.confirm("Bạn có chắc muốn xóa học sinh này?")) return;
  axios.delete(`http://localhost:5000/api/students/${id}`)
    .then(res => {
      console.log(res.data.message);
      setStudents(prevList => prevList.filter(s => s._id !== id));
    })
    .catch(err => console.error("Lỗi khi xóa:", err));
};
```

(Đoạn code trên còn dùng window.confirm để hỏi lại người dùng cho an toàn. Nếu người dùng chọn OK thì mới tiến hành xóa.)

- **Bước 4: Kiểm thử chức năng xóa** – Thêm vài học sinh và hiển thị trên trang (các bài trước). Thực hiện xóa:
- Bấm "Xóa" với một học sinh, xác nhận xóa.
- Kiểm tra API trả về phản hồi thành công (có message).
- Kiểm tra trong MongoDB, học sinh đó đã bị xóa.
- Kiểm tra trên giao diện, học sinh đó không còn trong danh sách.

Thử các trường hợp đặc biệt: Xóa liên tiếp nhiều học sinh; xóa học sinh mới thêm; đảm bảo không lỗi và danh sách luôn cập nhật đúng.

Bài 5: Tìm Kiểm Học sinh theo Tên

Yêu cầu

Bổ sung tính năng **tìm kiếm** cho phép người dùng nhanh chóng lọc danh sách học sinh theo tên. Ví dụ: có một ô nhập "Tìm kiếm..." – khi người dùng nhập chữ "An", danh sách sẽ chỉ hiện những học sinh có tên chứa "An".

Hướng dẫn thực hiện

- **Bước 1: Thêm ô tìm kiếm trên giao diện** – Ở phần trên bảng danh sách (hoặc trên thanh menu), thêm một input text cho phép nhập từ khóa tìm kiếm. Sử dụng state để lưu trữ chuỗi truy vấn. Ví dụ:

```
const [searchTerm, setSearchTerm] = useState("");

```

```
type="text"
placeholder="Tìm kiếm theo tên..."
value={searchTerm}
onChange={e => setSearchTerm(e.target.value)}
/>
```

Giao diện này có thể đặt ngay trên đầu trang danh sách hoặc trong một component `SearchBar` riêng.

- **Bước 2: Lọc danh sách dựa trên từ khóa** – Có hai hướng để thực hiện lọc:
- **Lọc trên client:** Đã có danh sách `students` trong state, ta có thể tạo một biến `filteredStudents = students.filter(...)` để chỉ gồm các học sinh có tên phù hợp. Việc so sánh nên không phân biệt hoa thường (sử dụng `.toLowerCase()` để so sánh). Cách này đơn giản và hiệu quả khi số lượng học sinh ít.
- **Lọc trên server (sử dụng API):** Gửi truy vấn lên server, ví dụ call `GET /api/students?name=<searchTerm>`, và điều chỉnh backend trả về kết quả lọc. Cách này hữu ích khi dữ liệu rất lớn, nhưng phức tạp hơn (cần viết thêm logic trên backend để tìm kiếm theo tên với \$regex hoặc tương tự).

Với bài tập này, có thể chọn cách 1 (lọc trên client) cho đơn giản. Ví dụ:

```
const filteredStudents = students.filter(s =>
  s.name.toLowerCase().includes(searchTerm.toLowerCase()));
```

Sau đó khi render, thay vì `students.map` thì dùng `filteredStudents.map` để hiển thị bảng.

- **Bước 3: Cập nhật hiển thị theo thời gian thực** – Với cách lọc client, mỗi lần `searchTerm` thay đổi (`onChange` input), React sẽ render lại danh sách theo mảng đã lọc. Người dùng gõ đến đâu, kết quả lọc xuất hiện đến đó (có thể thêm một độ trễ nhỏ hoặc `debounce` nếu muốn, nhưng không bắt buộc).
- **Bước 4: Kiểm thử tính năng tìm kiếm** – Thêm một số học sinh với tên khác nhau (ví dụ: "An", "Anh", "Bình", "Lan"...). Thử nhập các từ khóa khác nhau vào ô tìm kiếm:
 - Nhập tên đầy đủ hoặc một phần, danh sách chỉ hiển thị các kết quả khớp.
 - Xóa nội dung ô tìm kiếm, danh sách trở về đầy đủ.
 - Đảm bảo việc tìm kiếm không làm thay đổi dữ liệu gốc (chỉ là hiển thị tạm thời).

Bài 6: Sắp Xếp Danh Sách Học sinh theo Tên

Yêu cầu

Thêm tính năng **sắp xếp** để người dùng có thể thay đổi thứ tự sắp xếp danh sách học sinh, ví dụ sắp xếp theo tên ($A \rightarrow Z$ hoặc $Z \rightarrow A$).

Hướng dẫn thực hiện

- **Bước 1: Thêm nút hoặc điều khiển để sắp xếp** – Có thể sử dụng một nút bấm "Sắp xếp theo tên" mà khi nhấn sẽ đảo trạng thái sắp xếp (tăng dần \leftrightarrow giảm dần), hoặc làm các nút riêng biệt ("Tên A-Z", "Tên Z-A"). Cũng có thể cho phép click trực tiếp vào tiêu đề cột "Họ Tên" để sắp xếp. Ở đây chọn cách đơn giản: một nút toggle.

```
const [sortAsc, setSortAsc] = useState(true);
<button onClick={() => setSortAsc(prev => !prev)}>
  Sắp xếp theo tên: {sortAsc ? 'A  $\rightarrow$  Z' : 'Z  $\rightarrow$  A'}
</button>
```

Nút này hiển thị trạng thái hiện tại và khi bấm sẽ đổi sortAsc.

- **Bước 2: Thực hiện sắp xếp trên danh sách** – Tương tự tìm kiếm, ta thao tác trên mảng students (hoặc filteredStudents nếu kết hợp với tìm kiếm). JavaScript cung cấp hàm .sort() cho mảng. Ví dụ:

```
const sortedStudents = [...filteredStudents].sort((a, b) => {
  if (a.name < b.name) return sortAsc ? -1 : 1;
  if (a.name > b.name) return sortAsc ? 1 : -1;
  return 0;
});
```

Ở đây ta sao chép mảng trước khi sort ([...]) để không làm thay đổi mảng gốc trong state (giữ tính bất biến). So sánh chuỗi tên để trả về thứ tự mong muốn tùy theo sortAsc.

- **Bước 3: Hiển thị danh sách đã sắp xếp** – Khi render, sử dụng mảng sau khi sắp xếp: sortedStudents.map(...). Mỗi lần người dùng nhấn nút toggle, state sortAsc đổi giá trị, component render lại và mảng sortedStudents được tính lại theo thứ tự mới.
- **Bước 4: Kiểm thử tính năng sắp xếp** – Đảm bảo khi nhấn nút sắp xếp, thứ tự các hàng trong bảng thay đổi đúng hướng. Thử nhấn nhiều lần để xem có đảo qua lại giữa $A \rightarrow Z$ và $Z \rightarrow A$ hay không. Kiểm tra với dữ liệu có nhiều tên, bao gồm cả

trường hợp tên chữ thường/chữ hoa (nên đưa về cùng dạng khi so sánh để tránh sai thứ tự).

[1] How to mount external volume for mongoDB using docker-compose and docker-machine - Stack Overflow

<https://stackoverflow.com/questions/34390220/how-to-mount-external-volume-for-mongodb-using-docker-compose-and-docker-machine>

[2] Mongoose v9.0.0: Connecting to MongoDB

<https://mongoosejs.com/docs/connections.html>