Batch: D-2	**Roll No.: 16010123325**
Experiment / assignment / tutorial No._6___	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

Title: Demonstrate axios to Create Mock API Server

AIM: To Implement the React Axios

Problem Definition:

Build a React application that interacts with a RESTful API using Axios to perform CRUD (Create, Read, Update, Delete) operations. The application should allow users to view, add, update, and delete data from the server. The application should allow users to view, add, update, and delete student data, with smooth navigation between different views using the `useNavigate` hook.

Requirements:

- Create a new React application using create-react-app.
- Install Axios using npm install axios.
- Install react-router-dom to handle navigation (npm install react-router-dom).

Data Fetching:

Create a component (StudentList.js) that fetches a list of students from a RESTful API endpoint (e.g., https://api.example.com/students) and displays them in a table or list. Handle loading states and errors during the fetch process.

Adding a New Student:

- Implement a form component (AddStudent.js) that allows users to add a new student record.
- Use Axios to send a POST request to the API with the new student data.
- Upon successful submission, navigate the user back to the student list view using useNavigate and display the newly added student in the list.

Updating Student Data:

- Implement an edit functionality in a separate component (EditStudent.js) that allows users to update an existing student's information.
- Use Axios to send a PUT request to the API with the updated student data.
- Upon successful submission, navigate the user back to the student list view using useNavigate, and reflect the updated student information in the list.

Deleting a Student:

- Add a delete button next to each student in the list.
- When the delete button is clicked, use Axios to send a DELETE request to the API.
- Upon successful deletion, the student should be removed from the list without requiring a page reload.

Navigation:

- Use useNavigate to smoothly navigate between different components/views (StudentList, AddStudent, EditStudent).
- Ensure that the browser's back and forward buttons work correctly to navigate between the views.

Resources used:
https://axios-http.com/docs/intro
https://react.dev/reference/react

Expected OUTCOME of Experiment:

CO 2:. Illustrate the concepts of various front-end, back-end web application development technologies & frameworks using different web development tools.

Books/ Journals/ Websites referred:

1. Shelly Powers Learning Node O' Reilly 2 nd Edition, 2016.

Pre Lab/ Prior Concepts:

Write details about the following content

useNavigate

- A hook provided by **React Router DOM**.
- Allows navigation between pages programmatically (without using links).
- Example use: after form submission, navigate back to the list page.
- Replaces the older `useHistory` hook in React Router v6.

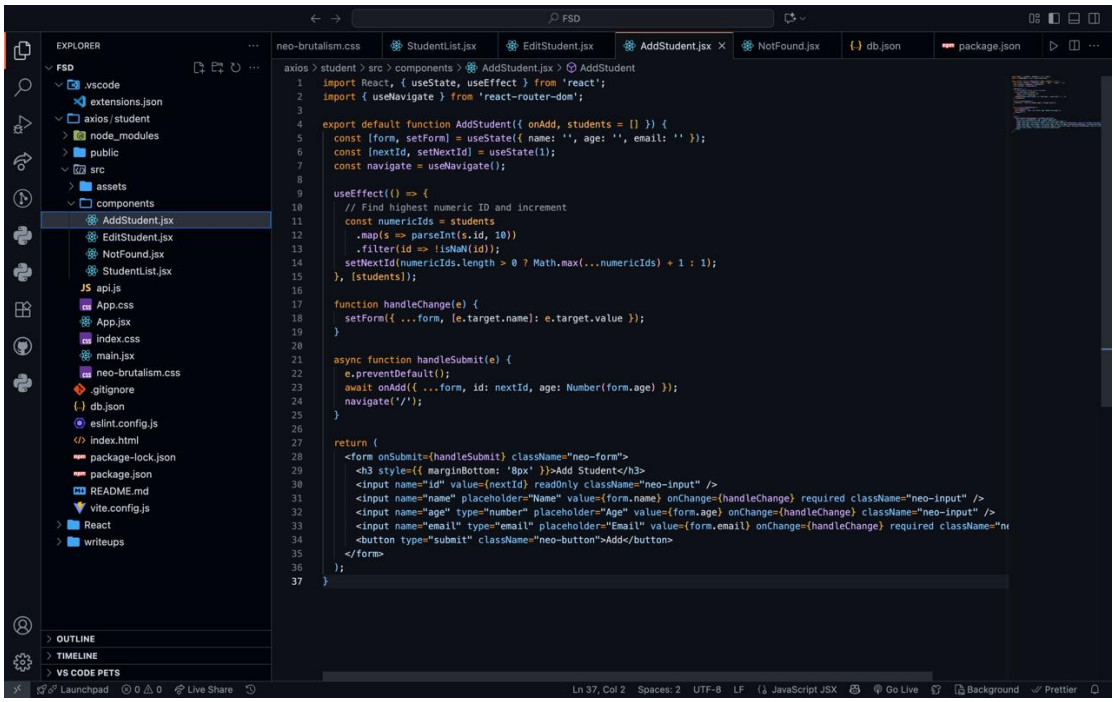
Axios

- A **JavaScript library** used to make HTTP requests (GET, POST, PUT, DELETE).
- Simplifies communication between React frontend and backend APIs.
- Supports features like request/response interceptors, error handling, and automatic JSON conversion.
- Commonly used for RESTful API integration.

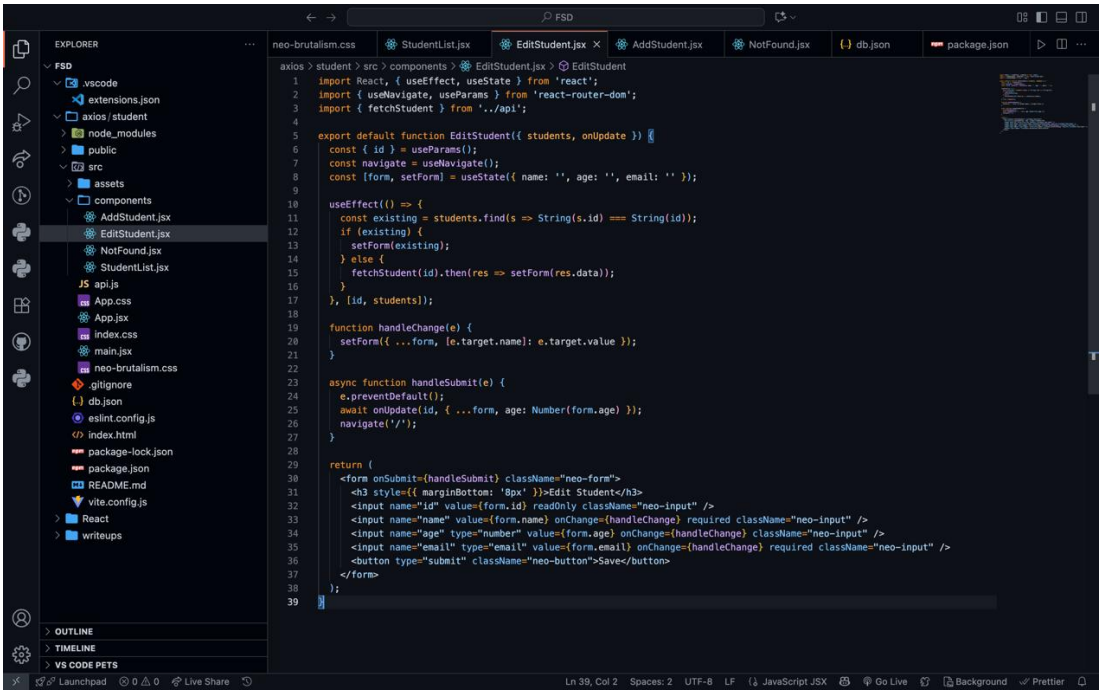
Routes in React

- Part of **React Router DOM** used to define different views (pages) in a single-page application.
- Each `<Route>` matches a URL path and renders the specified component.
- Example: `/` → StudentList, `/add` → AddStudent, `/edit/:id` → EditStudent.
- Helps build multi-page navigation without reloading the whole page.

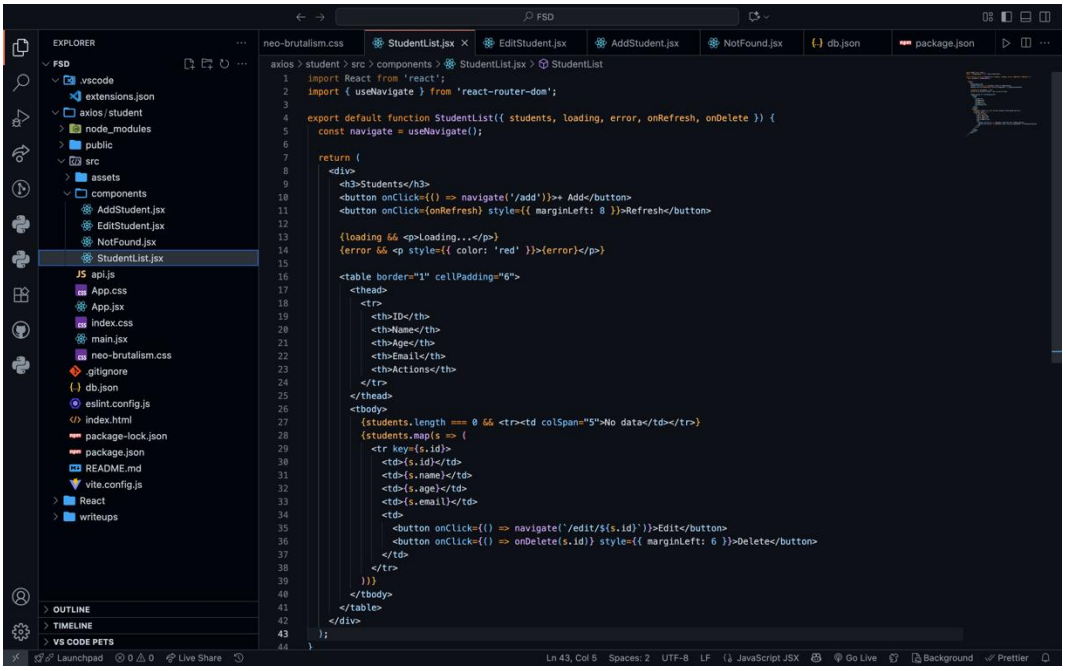
Methodology:
AddStudent



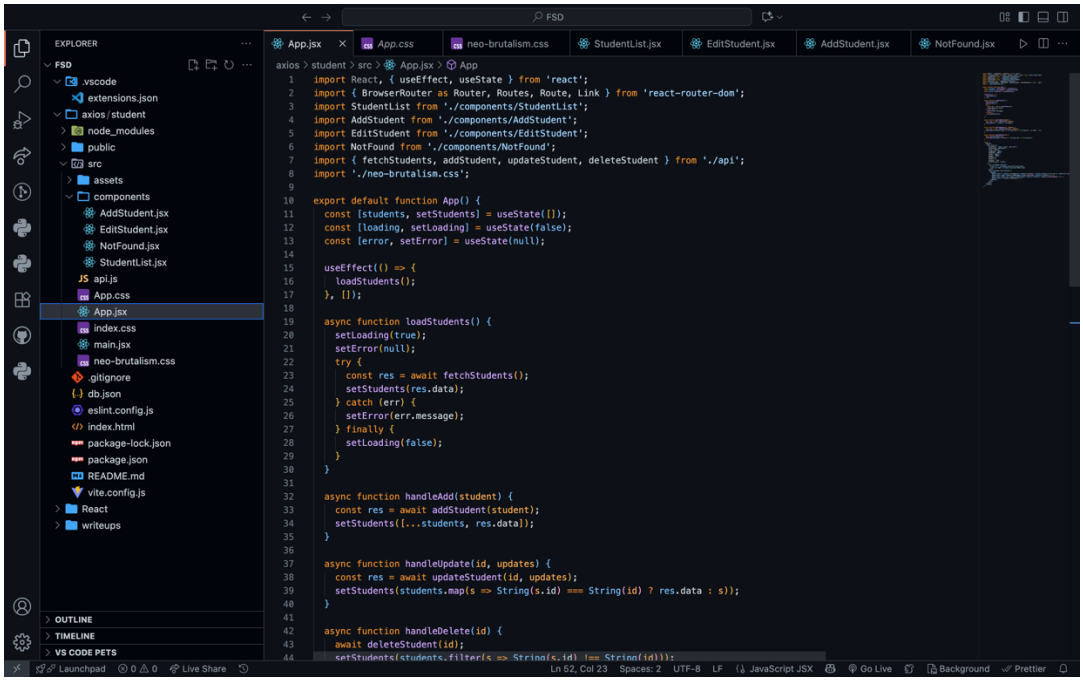
EditStudent



StudentList



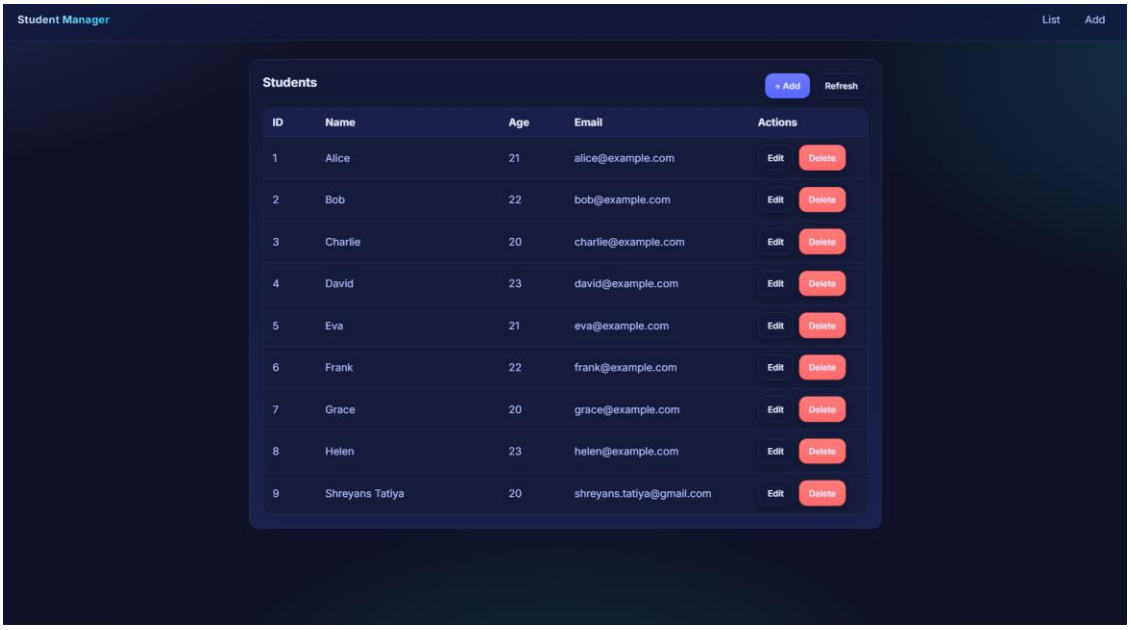
App



Implementation Details:

StudentList

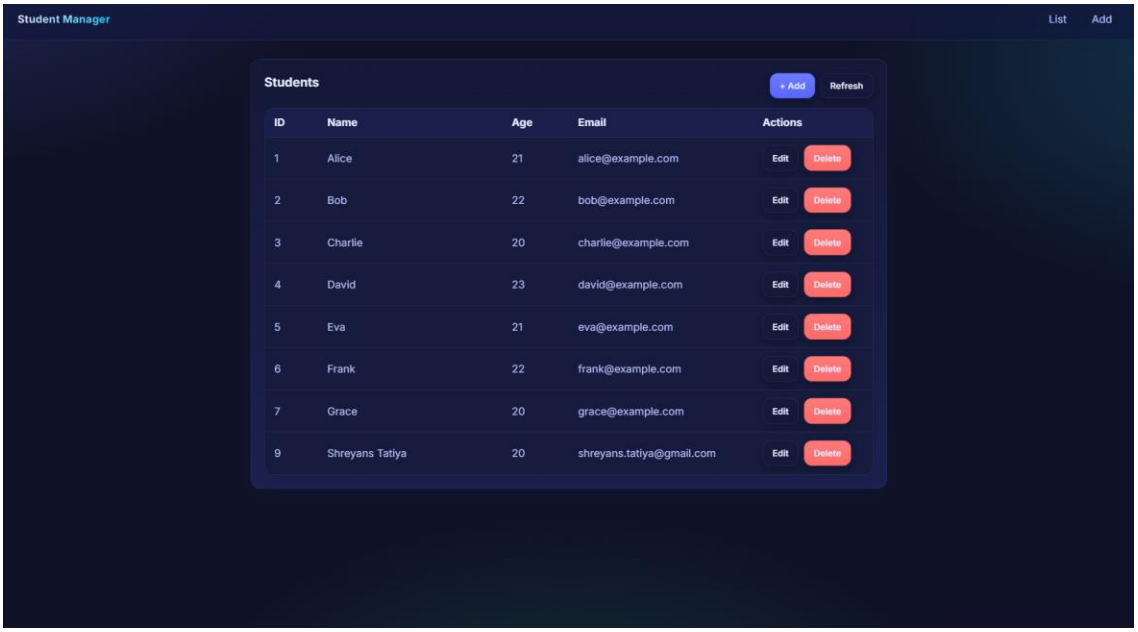
1. App.jsx passes down:
   - students → array from parent state.
   - loading → boolean while fetching.
   - error → error message if load failed.
   - onRefresh → the loadStudents() function in App.
   - onDelete → the handleDelete() function in App.
2. Inside StudentList:
   - If loading === true, it shows "Loading…".
   - If error is set, it shows the error message and a retry button that calls onRefresh.
   - Otherwise it renders a table of students.
3. For each student row:
   - **Edit button** → navigate("/edit/" + s.id) → goes to your EditStudent route.
   - **Delete button** → calls onDelete(s.id) → that triggers App's handleDelete.

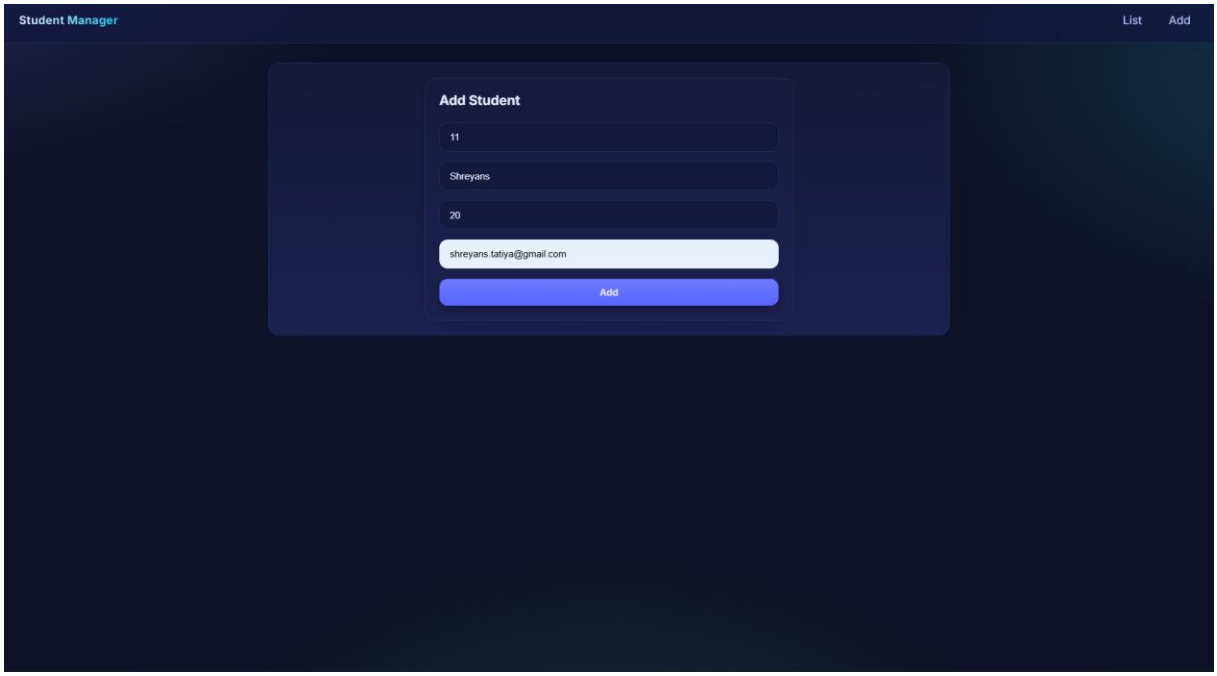


Delete

1. User clicks Delete next to a student.
2. A confirmation prompt appears to prevent accidental deletion.

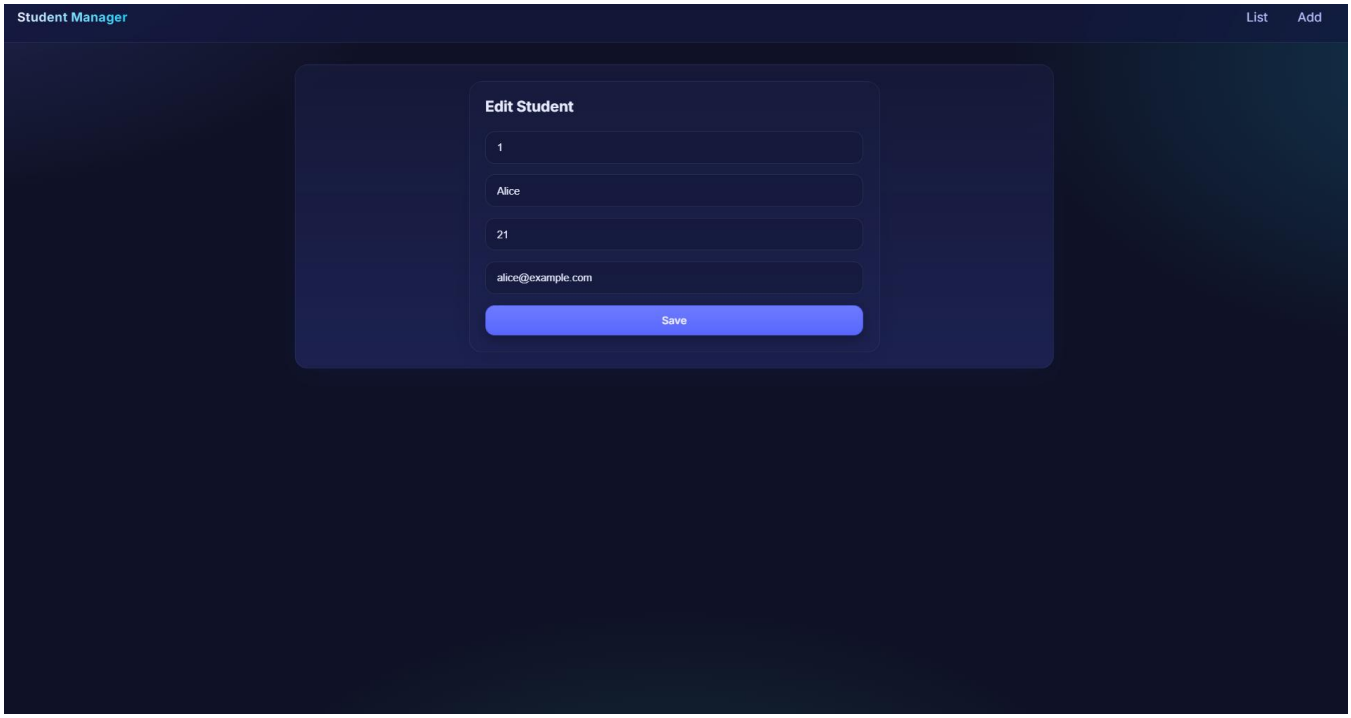
3. If confirmed:
    - The component calls the parent's `onDelete(id)` function.
    - That function sends a DELETE request to the API (`/students/:id`).
    - On success, the parent removes that student from its local `students` array.
4. The StudentList re-renders automatically with the updated array, so the row disappears.
5. If the API call fails, the user is shown an error (for example, an alert message).





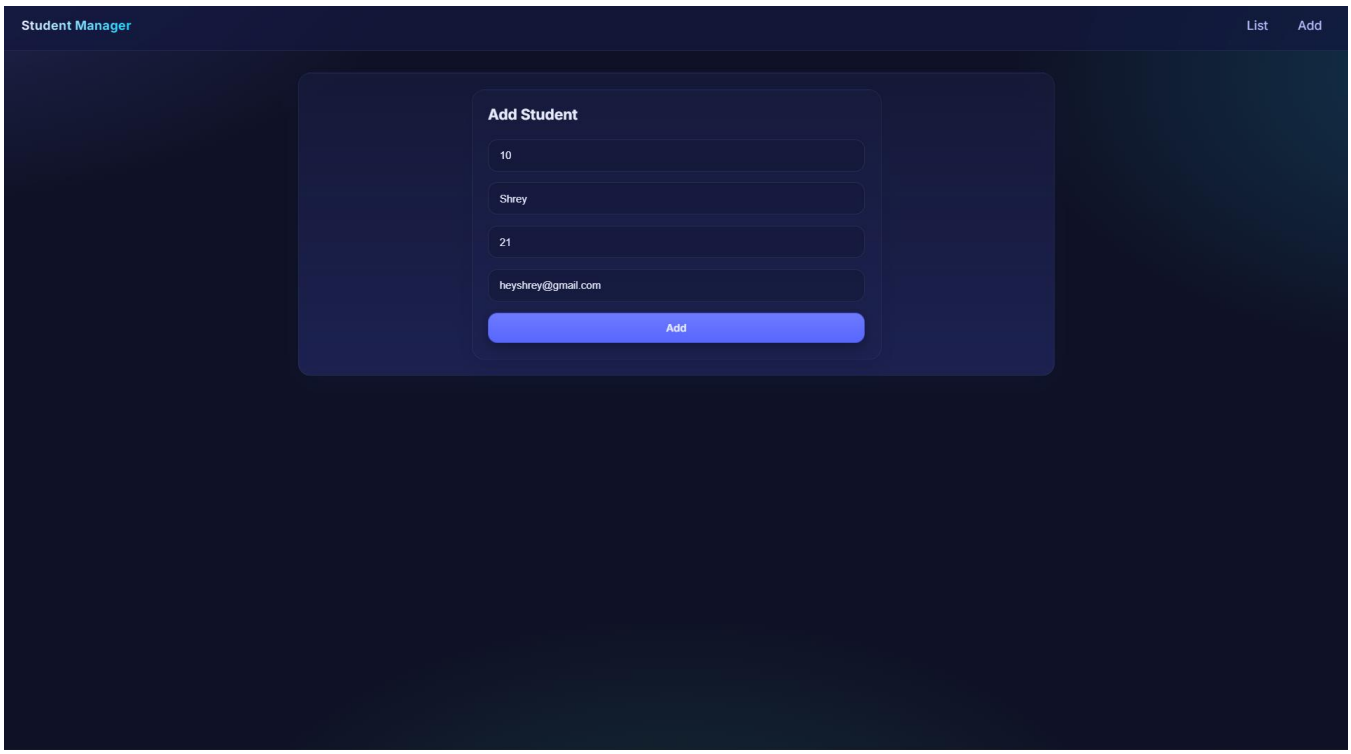
Edit

1. User clicks Edit next to a student in the list.
2. App navigates to /edit/:id and loads the EditStudent component.
3. EditStudent tries to pre-fill the form:
    - First, by checking if the student already exists in the parent's students array.
    - If not found, it fetches that student directly from the API (/students/:id).
4. The form displays the current values (name, age, email).
5. User modifies the fields and clicks Save.
6. EditStudent builds a new object with the updated details (including the id).
7. It calls the parent's onUpdate(id, updatedStudent).
8. The parent sends a PUT request to the API (/students/:id) with the full updated object.
9. If the request succeeds:
    - The parent replaces that student in its local students array with the updated version.
    - EditStudent navigates back to the list view.
    - The StudentList now shows the updated values.



Add

1. User clicks **"Add"** link in the navbar → navigates to /add.
2. App.jsx renders the AddStudent component and passes down onAdd={handleAdd}.
3. AddStudent shows an empty form with fields (name, age, email).
4. User types into the form → handleChange updates the local form state for each field.
5. When the user submits the form:
    - handleSubmit prevents page reload.
    - It builds a payload object (ensures age is a number).
    - Calls onAdd(payload) from props.
6. In App.jsx, handleAdd runs:
    - Sends a POST request to the API (/students) with the new data.
    - API responds with the new student (including generated id).
    - App updates its students state by appending this new object.
7. After the add succeeds, AddStudent calls navigate("/").
8. App re-renders the StudentList with the updated students array, so the new student appears in the list.
9. If the API call fails, the form could display an error instead of navigating (currently your code doesn't show it).



Requirements

- Node.js and npm
- Vite (React) setup
- Axios for API requests
- React Router DOM for navigation
- JSON Server as mock REST API

Steps for Execution

1. Set up a React (Vite) project and install required dependencies.
2. Configure JSON Server with a sample student dataset to act as the backend API.
3. Implement **Student List** view to display all students and provide Edit/Delete options.
4. Implement **Add Student** form to collect new student details and insert them via API.
5. Implement **Edit Student** form to update existing student details through the API.
6. Enable **Delete Student** functionality to remove a record permanently.
7. Use React Router for navigation between List, Add, and Edit views.
8. Run the backend (JSON Server) and frontend (Vite) together to test full CRUD flow.

Conclusion:

The experiment successfully demonstrates how to perform CRUD operations in a React application using Axios for API communication, React Router for navigation, and JSON Server as a backend. It shows effective integration of frontend state with RESTful APIs to manage student records dynamically.

Postlab questions:

1) Different ways to Add Api in React/Javascript with example.

1. Using Fetch API (built-in)

- **Usage:** Simple and lightweight, returns a Promise.

```
fetch('https://api.example.com/students')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

2. Using Axios (external library)

- **Usage:** Automatically parses JSON and has better error handling.

```
import axios from 'axios';

axios.get('https://api.example.com/students')
  .then(res => console.log(res.data))
  .catch(err => console.error(err));
```

3. Using async/await (with Fetch or Axios)

- **Description:** Cleaner syntax for asynchronous code; avoids chaining .then().

```
async function getStudents() {
  try {
    const res = await axios.get('https://api.example.com/students');
    console.log(res.data);
  } catch (err) {
    console.error(err);
  }
}
```

4. Using External State

- **Usage:** Useful for larger apps, handles caching, re-fetching, and synchronization automatically.

```
import { useQuery } from '@tanstack/react-query';
import axios from 'axios';

function StudentList() {
  const { data, error, isLoading } = useQuery(['students'], () =>
    axios.get('https://api.example.com/students').then(res => res.data)
  );
}
```