What is React?

React is a JavaScript library for building user interfaces. It helps you create interactive UIs in a modular and reusable way.

Prerequisites

- Basic knowledge of HTML, CSS, and JavaScript.
- Node.js installed (download from https://nodejs.org/en/download).
- A code editor like VS Code.

REACT JS

- https://react.dev/blog/2023/03/16/introducing-react-dev

Step 1: Setup a React App using vite

```
# 1. Create a new Vite + React project
npm create vite@latest my-react-app -- --template react

# 2. Go into the project folder
cd my-react-app

# 3. Install dependencies
npm install

# 4. Start the dev server
npm run dev
```

Project Structure

You'll mostly work in `src/App.js` for now.

Step 2: Learn React Hooks (with Examples)

1. `useState` — for managing state

```
import React, { useState } from 'react';

function Counter() {
 const [count, setCount] = useState(0); // count = current value, setCount =
updater

 return (
   <div>
     <p>You clicked {count} times</p>
     <button onClick={() => setCount(count + 1)}>Click Me</button>
   </div>
 );
}
```

2. `useEffect` — for side effects (like fetching data)

```
import React, { useEffect } from 'react';

function Timer() {
useEffect(() => {
console.log('Component mounted!');

return () => {
console.log('Component unmounted!');
};
}, []); // Empty dependency = run once when mounted
}
```

Step 3: Build a Basic To-Do App

Features:

- Add tasks
- Display task list
- Delete tasks

Code

src/App.js:

```
import React, { useState } from 'react';

import './App.css';

function App() {

 const [task, setTask] = useState('');

 const [tasks, setTasks] = useState([]);

 const handleAdd = () => {

   if (task.trim() === '') return;

   setTasks([...tasks, task]);

   setTask('');

 };

 const handleDelete = (index) => {

   const newTasks = tasks.filter((_, i) => i !== index);

   setTasks(newTasks);

 };

 return (

   <div className="App">

     <h1>To-Do App</h1>

     <input

       type="text"

       placeholder="Enter a task"

       value={task}

       onChange={(e) => setTask(e.target.value)}

     />

     <button onClick={handleAdd}>Add Task</button>



     <ul>

       {tasks.map((t, i) => (

         <li key={i}>

           {t} <button onClick={() => handleDelete(i)}>Delete</button>

         </li>
```

```
        ))}
      </ul>
    </div>
  );
}
export default App;
```

`src/App.css` (optional styling):

```
.App {
  font-family: sans-serif;
  padding: 2rem;
  text-align: center;
}

input {
  padding: 0.5rem;
  margin-right: 0.5rem;
}

button {
  padding: 0.5rem;
  margin-left: 0.5rem;
}
```

Exercise Suggestions

1. Add "Delete All button" button to remove all task

Using `useState` with an Object

When you're managing multiple related pieces of data (like form fields), it’s often convenient to store them in a single object.

Example: Handling a Form with `useState`

```jsx
import React, { useState } from 'react';

function UserForm() {

 const [formData, setFormData] = useState({

   name: '',

   email: '',

 });

 const handleChange = (e) => {

   const { name, value } = e.target;

   setFormData((prevData) => ({

     ...prevData,          // keep previous values

     [name]: value,        // update the changed field

   }));

 };

 return (

   <div>

     <h2>User Form</h2>

     <input

       type="text"

       name="name"

       value={formData.name}

       placeholder="Enter your name"

       onChange={handleChange}

     />

     <input

       type="email"

       name="email"

       value={formData.email}

       placeholder="Enter your email"

       onChange={handleChange}

     />

     <p>Name: {formData.name}</p>

     <p>Email: {formData.email}</p>

   </div>
```

```
  );

}

export default UserForm;
```

Explanation

1. **Initial State is an object:**

```
useState({ name: '', email: '' });
```

2. `handleChange` **function** dynamically updates the correct key using:

```
[name]: value
```

3. `...prevData` spreads the existing object to preserve unchanged values.

Exercise 1

- use input for react js and TextInput for react native for user input.
  Part 1:
  In this exercise, you will create a form to collect the personal information of the user. The form should include the following fields:

    - **Required Fields**:
        - First Name
        - Last Name
        - Age
        - Birthdate
        - Address
    - **Optional Fields**:
        - Middle Name
        - Suffix (e.g., Jr., Sr., III)

  The form should validate that all required fields have valid inputs before submission. Once the user fills in all the required fields and submits the form:

- Display a modal showing the complete information entered by the user (both required and optional fields).
- Ensure the modal is neatly formatted to clearly present the data.

Optional fields should display as "N/A" or similar if left empty. Use proper error handling and validation to guide the user during input (e.g., show an error if required fields are left blank).

Part 2:

In this exercise, you will create a table to display all the personal information submitted from Part 1. The table should have the following functionalities:

1. **Display Submitted Data**:
    - After a user submits their personal information, their data should be added as a new row in the table.
    - The table should show all required and optional fields from the form.
2. **Add New Information**:
    - Provide a button or form to allow users to add new entries to the table.
    - Use a modal to confirm the addition of new data before saving it to the table.
3. **Edit Existing Information**:
    - Include an option to edit details for any row in the table.
    - Display a modal with the current data pre-filled, allowing the user to make changes.
    - Confirm the changes through a modal before updating the table.
4. **Delete Entries**:
    - Allow users to delete any row from the table.
    - Use a modal to confirm the deletion before removing the data.

Each function (add, edit, delete) must include proper validation and a confirmation modal to ensure data integrity and a seamless user experience. The table should update dynamically to reflect any changes made by the user.

Exercise 2

- Use axios for fetching data from api
- Sample code for fetching data using axios

```
const [result, setResult] = useState([])

useEffect(()=>{
        getData();
,[])

const getData = async()=>{
        const response = await axios.get(“link”);
        setResult(response.data);
}
```

- **Weather Data**

  In this exercise, you will fetch weather data from an API and display the results. The steps are as follows:

1. **API Endpoint**: Use Axios to fetch data from the following API:
   https://goweather.herokuapp.com/weather/{manila}
   Replace {manila} with the desired city (e.g., "manila") to fetch the weather details.
2. **Display Fields**: Extract and display the following information from the API response:
   - Temperature
   - Wind
   - Description
3. **Error Handling**: Implement proper error handling to display an appropriate message if the API request fails or if the response data is incomplete.
4. **UI Requirements**:
   - Present the fetched weather information in a clean and user-friendly format.
   - Clearly label each field (e.g., "Temperature: 30°C").
   - Include a loading indicator while the data is being fetched.
5. **Enhancement**: Allow the user to input a city name dynamically and fetch weather data for the entered city.

Use proper error handling, clear labels, and feedback to guide the user during interaction (e.g., show an error if the city is invalid).

- **Ecommerce**
- **https://fakestoreapi.com/**

In this exercise, you will fetch product data from an API and display it in interactive cards. The steps are as follows:

1. **API Endpoint**: Use Axios to fetch data from the following API:
   https://fakestoreapi.com/products/
2. **Display in Cards**: For each product, display the following information on a card:
   - Title
   - Price
   - Category
   - Rating (e.g., "Rating: 4.5")
3. **Card Interaction**:
   - When a user taps or clicks on a card, open a modal or navigate to a details screen to display the complete information of the selected product.
   - Include the following fields in the detailed view:
     - Title
     - Price
     - Category
     - Rating
     - Description
     - Available Stock

4. **UI Requirements**:
   - Ensure the cards are styled neatly with clear labeling for each field.

    - The detailed view (modal or screen) should present the information in a clean and readable format.
5. **Error Handling**:
    - Display an error message if the API request fails or if the response data is empty.
    - Handle edge cases like missing fields gracefully (e.g., show "N/A" if a value is unavailable).
6. **Optional Enhancements**:
    - Add a loading indicator while fetching the product data.
    - Allow the user to search or filter the product list by title or category.