

Tài liệu môn Phát triển giao diện ứng dụng

1. React Component

1.1. Function Component

- Là các component được khai báo bằng function JavaScript.
- Dễ đọc, dễ bảo trì, thường dùng với React Hooks.

Ví dụ:

```
function Greeting(props) {  
  return  
    <h1>Xin chào, {props.name}!</h1>;  
}
```

```
export default Greeting;
```

sử dụng: `<Greeting name="Trường" />`

1.2. Class Component

- Là các component sử dụng **class**, trước đây được dùng nhiều nhưng hiện tại ít phổ biến hơn **function component**.
- Cần sử dụng `this.state` để quản lý **state** và `this.props` để truy cập **props**.

Ví dụ:

```
import React, { Component } from "react";  
  
class Counter extends Component {  
  constructor(props) {  
    super(props);  
    this.state = { count: 0 };  
  }  
  
  increment = () => {  
    this.setState({ count: this.state.count + 1 });  
  };  
  
  render() {  
    return (  
      <div>  
        <p>Count: {this.state.count}</p>  
        <button onClick={this.increment}>Tăng</button>  
      </div>  
    );  
  }  
}  
export default Counter;
```

1.3. Stateful Component (Component có state)

- Là những component có thể lưu trạng thái bên trong (dùng **useState** trong **function component** hoặc **this.state** trong **class component**).

Ví dụ:

```
import { useState } from "react";

function ToggleButton() {
  const [isOn, setIsOn] = useState(false);
  return (
    <button onClick={() => setIsOn(!isOn)}>
      {isOn ? "Bật" : "Tắt"}
    </button>
  );
}

export default ToggleButton;
```

1.4. Stateless Component (Component không có state)

- Là các component chỉ nhận dữ liệu từ props và hiển thị mà không có trạng thái riêng.

Ví dụ:

```
function Welcome(props) {
  return <h2>Chào mừng, {props.username}!</h2>;
}
```

1.5. State trong React

State là một đối tượng đặc biệt trong React giúp component lưu trữ và quản lý dữ liệu thay đổi theo thời gian. Khi state thay đổi, component sẽ tự động re-render để cập nhật giao diện.

Ví dụ:

```
import { useState } from "react";
function Counter() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>Giá trị: {count}</p>
      <button onClick={() => setCount(count + 1)}>Tăng</button>
    </div>
  );
}

export default Counter;
```

- **count** là *state*.
- **setCount** là hàm dùng để cập nhật *state*.
- Khi nhấn nút, *state* thay đổi, React sẽ *re-render component*.

1.6. Props trong React

Props (Properties) là cách truyền dữ liệu từ component cha xuống component con. Props là **immutable** (không thể thay đổi trong component con).

Ví dụ:

```
function Welcome(props) {  
  return <h1>Xin chào, {props.name}!</h1>;  
}  
export default Welcome;
```

Cách sử dụng trong component cha:

```
<Welcome name="Trường" />
```

- **name** là một **prop** được truyền từ component cha vào Welcome.
- **props.name** được hiển thị trên giao diện.

1.7. Component lồng (Nested Components)

Trong React, một component có thể chứa các component khác. Điều này giúp mã nguồn dễ tổ chức và tái sử dụng

Ví dụ:

```
function Avatar(props) {  
  return <img src={props.imageUrl} alt="Avatar" width="100" />;  
}  
  
function UserProfile() {  
  return (  
    <div>  
      <h2>Thông tin người dùng</h2>  
      <Avatar imageUrl="https://example.com/avatar.jpg" />  
    </div>  
  );  
}
```

1.8. Cách truyền dữ liệu từ Component Cha sang Component Con

- Dữ liệu được truyền từ cha sang con thông qua **props**.

Ví dụ:

```
// Component Cha  
function ParentComponent() {  
  return <ChildComponent message="Xin chào từ cha!" />;  
}  
  
export default ParentComponent;  
  
// Component Con  
function ChildComponent(props) {
```

- ```
 return <p>Dữ liệu từ cha: {props.message}</p>;
 }

```
- **ParentComponent** truyền **prop *message*** xuống **ChildComponent**.
  - **ChildComponent** hiển thị nội dung prop trên UI.

## 1.9. Cách truyền hàm từ Cha sang Con (Callback Props)

Đôi khi, con cần gửi dữ liệu ngược lại cho cha, ta truyền một hàm từ cha xuống.

Ví dụ:

```
function Child(props) {
 return (
 <button onClick={() => props.onButtonClick("Dữ liệu từ con")}>
 Gửi dữ liệu cho cha
 </button>
);
}

function Parent() {
 const handleDataFromChild = (data) => {
 alert("Nhận được từ con: " + data);
 };

 return <Child onClick={handleDataFromChild} />;
}

export default Parent;

```

## 1.10. Bài tập

**Bài tập 1:** sử dụng **class component** xây dựng một ứng dụng web có dạng:

My name is: An  
Age: 29

Your name:   
Your Age:

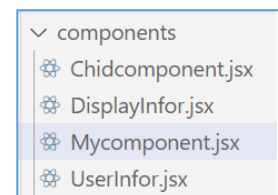
---

Hide list User  
User name is: Dung  
User Age: 49

---

User name is: Hoang  
User Age: 17

Hướng dẫn



- Tạo thư mục components trong thư mục src, trong components, tạo các tập tin theo cấu trúc như hình
- Lồng component như hình

| ▼ Mycomponent |
|---------------|
| UserInfor     |
| DisplayInfor  |

Trong Mycomponent.jsx

```
class Mycomponent extends React.Component {
 state = {
 listUser: [
 { id: 1, Name: "Dung", Age: 49 },
 { id: 2, Name: "Hoang", Age: 17 },
 { id: 3, Name: "Chien", Age: 32 },
]
 }
 handleAddnewUser = (userObject) => {
 this.setState({
 listUser: [userObject, ...this.state.listUser]
 })
 }
 handleDeleteUser = (userID) => {
 let listUserClone = this.state.listUser;
 listUserClone = listUserClone.filter(item => item.id !== userID)
 this.setState({
 listUser: listUserClone
 })
 }
 render() {
 Trong phần render() nhúng component con
 <AddUserInfor handleAddnewUser={this.handleAddnewUser}></AddUserInfor>
 <hr />
 <DisplayInfor listUser={this.state.listUser}
 handleDeleteUser={this.handleDeleteUser}>
 </DisplayInfor>
 }
}
```

Trong AddnewUserInfor.jsx

```
handleOnSubmit = (event) => {
 event.preventDefault(); //ngăn việc tải lại trang
 this.props.handleAddnewUser({
 id: Math.floor((Math.random() * 100) + 1) + "user",
 Name: this.state.Name,
 Age: this.state.Age
 })
}
```

Trong phần render của AddNewUserInfor.jsx

```

<form action="" onSubmit={(event) => this.handleOnSubmit(event)}>
 <label htmlFor="">Your name:</label>
 <input type="text" value={this.state.Name}
 onChange={(event) => this.handleChangeInput(event)} />

 <label htmlFor="">Your Age:</label>
 <input type="text"
 onChange={(event) => this.handleChangeAge(event)}
 value={this.state.Age} />
 <button>Submit</button>
</form>

```

Trong component DisplayInfor, hiển thị danh sách user lên trình duyệt

```

class DisplayInfor extends React.Component {
 render() {
 const {listUser} = this.props; //truyền từ cha sang con
 return (
 <div>
 {listUser.map((user) => {
 return (
 <div key={user.id}>
 <div>My name is: {user.Name}</div>
 <div>My Age: {user.Age}</div>
 <hr />
 </div>
)
 })}
 </div>
)
 }
}

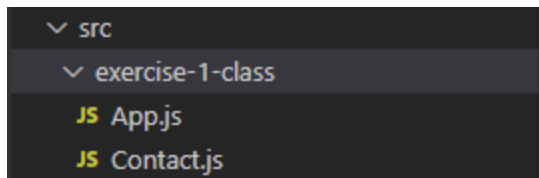
```

**Bài tập 2:** Chuyển các **class component** trong bài 1 sang dạng **function component**

**Bài tập 3:** sinh viên tự làm React Props & State

### Exercise 1: Class Components

- + Expand the **src** folder, then the exercise-1-class folder.



- + In this folder there are two class components, the **App** component and the **Contact** component.

### The Goal

- + We want our Contact component to be able to show information about any contact. We need it to work dynamically and react to input data that another component gives it.
- + For example, the App component has an example contact

```

const chidi = {
 firstName: "Chidi",
 lastName: "Anagonye",
 phone: "555-366-8987",
 address: "St. John's University, Sydney, Australia"
}

```

- + Our App component should give this contact to our Contact component and the Contact component should display it like this:

|                                                          |
|----------------------------------------------------------|
| <p><b>Chidi</b></p> <p>Anagonye</p>                      |
| <p>Phone: 555-366-8987</p>                               |
| <p>Address: St. John's University, Sydney, Australia</p> |

## Exercise 1: Functional Components

- + Expand the src folder, then the **exercise-1-functional** folder.
- + In this folder there are two **functional components**, the **App** component and the **Contact** component.

### The Goal

Just like with the class component version, we want our Contact component to be able to show information about any contact. We need it to work dynamically and react to input data that another component gives it.

## Exercise 2: Class Components

- + Expand the src folder, then the exercise-2-class folder.
- + In this folder there are three class components, the App component, the Contact component, and the ContactList component. You can mix functional and class components, but for this exercise we'll use all class components to practice with them.
- + There is also a data.js file that holds some static data that we will use. In a real app, we might pull our data from a server, but for this exercise we'll just pull it from the data.js file.

### The Goal

- + In [Exercise 1](#) we set up our Contact component to show data about a contact.

- + Now we want our app to actually maintain some internal state data (a list of contacts) and show that data using our Contact List component, which uses the Contact component.
- + Our App should react to any changes in the list of contacts and re-render, which will re-render the ContactList component and the Contact components inside it.
- + The list of contacts should start out set to the array in the variable INITIAL\_CONTACTS in the data.js file.
- + So, when it's all completed, our app should look like this:

|                                                                                                         |                                                                                                            |                                                                                                      |                                                                                              |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <b>Chidi</b><br>Anagonye<br>Phone: 555-366-8987<br>Address: St. John's University,<br>Sydney, Australia | <b>Eleanor</b><br>Shellstrop<br>Phone: 555-483-1457<br>Address: 335 Avalon St, Apt 2C,<br>Pheonix, Arizona | <b>Tahani</b><br>Al-Jamil<br>Phone: 555-276-7991<br>Address: 1 Lancaster Terrace,<br>London, England | <b>Jason</b><br>Mendoza<br>Phone: 555-113-8388<br>Address: 779 William St, Miami,<br>Florida |
|---------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|

## Tips

- + The Contact component is identical to the one in [Exercise 1](#). You can copy your code from Contact.js in the exercise-1-class folder, into Contact.js in the exercise-2-class folder.
- + We will store our contacts in the state of the App component. Our ContactList and Contact components are purely presentational. Presentational means they only know how to present the data, they don't manage the data or have any data in state. They just react to changes in their props, which will flow down to **ContactList** from App (and to Contact from ContactList).
- + React needs to be able to tell exactly what changed when it rerenders your component. Any component that displays a list needs to put a key attribute on the top JSX element (component or HTML element) of each item. The key should be set to something that is unique to each item in the list. It's most often an ID.

## Exercise 2: Functional Components

- + Expand the **src** folder, then the **exercise-2-functional** folder.
- + In this folder there are three functional components, the **App** component, the **Contact** component, and the **ContactList** component. You can mix functional and class components, but for this exercise we'll use all functional components to practice with them.
- + There is also a data.js file that holds some static data that we will use. In a real app, we might pull our data from a server, but for this exercise we'll just pull it from the data.js file.

## The Goal

1. In [Exercise 1](#) we set up our Contact component to show data about a contact.
2. Now we want our app to actually maintain some internal state data (a list of contacts) and show that data using our Contact List component, which uses the Contact component.



3. Our App should react to any changes in the list of contacts and re-render, which will re-render the ContactList component and the Contact components inside it.
4. The list of contacts should start out set to the array in the variable `INITIAL_CONTACTS` in the `data.js` file.
5. So, when it's all completed, our app should look like this:

|                                                      |                                                     |                                                  |                                            |
|------------------------------------------------------|-----------------------------------------------------|--------------------------------------------------|--------------------------------------------|
| <b>Chidi</b><br>Anagonye                             | <b>Eleanor</b><br>Shellstrop                        | <b>Tahani</b><br>Al-Jamil                        | <b>Jason</b><br>Mendoza                    |
| Phone: 555-366-8987                                  | Phone: 555-483-1457                                 | Phone: 555-276-7991                              | Phone: 555-113-8388                        |
| Address: St. John's University,<br>Sydney, Australia | Address: 335 Avalon St, Apt 2C,<br>Pheonix, Arizona | Address: 1 Lancaster Terrace,<br>London, England | Address: 779 William St, Miami,<br>Florida |

## Tips

- + The Contact component is identical to the one in Exercise 1. You can copy your code from `Contact.js` in the `exercise-1-functional` folder, into `Contact.js` in the `exercise-2-functional` folder.
- + We will store our contacts in the state of the App component. Our `ContactList` and `Contact` components are purely presentational. Presentational means they only know how to present the data, they don't manage the data or have any data in state. They just react to changes in their props, which will flow down to `ContactList` from `App` (and to `Contact` from `ContactList`).

## Exercise 3: Class Components

- + Expand the `src` folder, then the `exercise-3-class` folder.
- + In this folder there are three class components, the **App** component, the **Contact** component, and the **ContactList** component. You can mix functional and class components, but for this exercise we'll use all class components to practice with them.
- + There is also a **data.js** file that holds some static data that we will use. In a real app, we might pull our data from a server, but for this exercise we'll just pull it from the `data.js` file.

## The Goal

- + In [Exercise 1](#) we set up our Contact component to show data about a contact.
- + In [Exercise 2](#) we added internal state data (a list of contacts) to our app and set up our `ContactList` component to show a list of contacts.
- + Now we want our app to be dynamic, and allow the user to change the state of the app.
- + There are two buttons: one in the **Contact** component that deletes one contact, another in the **ContactList** component that deletes all the contacts. These buttons should change the internal state data which will trigger a re-render of the app.
- + So, when it's all completed, our app will work like this:

|                                                                                                                   |                                                                                                                      |                                                                                                                |                                                                                                        |
|-------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------|
| <b>Chidi</b><br>Anagonye<br>Phone: 555-366-8987<br>Address: St. John's University,<br>Sydney, Australia<br>Delete | <b>Eleanor</b><br>Shellstrop<br>Phone: 555-483-1457<br>Address: 335 Avalon St, Apt 2C,<br>Pheonix, Arizona<br>Delete | <b>Tahani</b><br>Al-Jamil<br>Phone: 555-276-7991<br>Address: 1 Lancaster Terrace,<br>London, England<br>Delete | <b>Jason</b><br>Mendoza<br>Phone: 555-113-8388<br>Address: 779 William St, Miami,<br>Florida<br>Delete |
| Delete All                                                                                                        |                                                                                                                      |                                                                                                                |                                                                                                        |

## Tips

The Contact component is nearly identical to the one in [Exercise 1](#), with an addition. You can copy your code from Contact.js in the exercise-1-class folder, into Contact.js in the exercise-3-class folder. Then add in the new Delete button:

```
<li className="list-group-item text-end">
 <button className="btn btn-danger">Delete</button>

```

The ContactList component is nearly identical to the one in [Exercise 2](#), with an addition. You can copy your code from ContactList.js in the exercise-2-class folder, into ContactList.js in the exercise-3-class folder. Then add in the new Delete All button.

```
<div className="row mb-3">
 <div className="col text-end">
 <button className="btn btn-danger btn-lg">Delete All</button>
 </div>
</div>
```

Because we now want to return two divs from our render function, and React only allows us to return one parent element, we need to use React.Fragment to wrap both divs, like this:

```
<>
 <div className="row">
 CONTACTS HERE
 </div>
 <div className="row mb-3">
 <div className="col text-end">
 <button className="btn btn-danger btn-lg">Delete All</button>
 </div>
 </div>
</>
```

When we update our state, we should not modify the state directly. That means that if you are storing an array in state, you should not add or remove items directly from

that array. You can use an array method that makes a copy (some examples are `.filter()` and `.map()` and `.concat()` and `.slice()` ), or make a copy yourself and then make the change to the copy. You then set the state to the changed copy.

In React, the component that stores the data in its state is the only one that can change the data. State is internal to a component and no other component can change it. Usually the component that stores the data in its state will have functions that modify the state in all the needed ways (like deleting something, creating something, or updating something). If any of its child components need to be able to be able to modify the data, it will pass the needed function down to that child component through props. Then the child component can call that function when it needs to.

### Exercise 3: Functional Components

- + Expand the src folder, then the exercise-3-functional folder.
- + In this folder there are three functional components, the App component, the Contact component, and the ContactList component. You can mix functional and class components, but for this exercise we'll use all functional components to practice with them.
- + There is also a data.js file that holds some static data that we will use. In a real app, we might pull our data from a server, but for this exercise we'll just pull it from the data.js file.

### The Goal

- + In [Exercise 1](#) we set up our Contact component to show data about a contact.
- + In [Exercise 2](#) we added internal state data (a list of contacts) to our app and set up our ContactList component to show a list of contacts.
- + Now we want our app to be dynamic, and allow the user to change the state of the app. There are two buttons: one in the Contact component that deletes one contact, another in the ContactList component that deletes all the contacts. These buttons should change the internal state data which will trigger a re-render of the app.
- + So, when it's all completed, our app will work like this:

### Tips

The Contact component is nearly identical to the one in [Exercise 1](#), with an addition. You can copy your code from Contact.js in the exercise-1-functional folder, into Contact.js in the exercise-3-functional folder. Then add in the new Delete button.

The ContactList component is nearly identical to the one in [Exercise 2](#), with an addition. You can copy your code from ContactList.js in the exercise-2-functional folder, into ContactList.js in the exercise-3-functional folder. Then add in the new Delete All button.

Because we now want to return two divs from our render function, and React only allows us to return one parent element, we need to use `React.Fragment` to wrap both divs.

When we update our state, we should not modify the state directly. That means that if you are storing an array in state, you should not add or remove items directly from that array.

You can use an array method that makes a copy (some examples are `.filter()` and `.map()` and `.concat()` and `.slice()` ), or make a copy yourself and then make the change to the copy. You then set the state to the changed copy.

In React, the component that stores the data in its state is the only one that can change the data. State is internal to a component and no other component can change it. Usually the component that stores the data in its state will have functions that modify the state in all the needed ways (like deleting something, creating something, or updating something). If any of its child components need to be able to be able to modify the data, it will pass the needed function down to that child component through props. Then the child component can call that function when it needs to.

---

## 2. React hooks

---

**React hooks** là một tính năng mạnh mẽ trong React, giúp sử dụng state và các tính năng khác mà không cần phải sử dụng các lớp (class). **React hooks** được giới thiệu trong React 16.8 và làm cho việc viết mã React trở nên dễ dàng và rõ ràng hơn, đặc biệt là trong các function component.

### 2.1. useState

**useState** là hook dùng để khai báo **state** trong một **function component**.

Cú pháp:

```
const [state, setState] = useState(initialState);
```

- *state*: biến chứa giá trị state hiện tại.
- *setState*: hàm dùng để cập nhật state.
- *initialState*: giá trị ban đầu của state.

Ví dụ

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

function Counter() {
 const [count, setCount] = useState(0);

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

Giải thích:

- **useState(0)** khai báo một state **count** với giá trị ban đầu là 0.
- Mỗi khi người dùng nhấn nút, **setCount** sẽ cập nhật giá trị của **count**.

### 2.2. useEffect

**useEffect** là hook dùng để **thực thi các tác vụ phụ** (side effects) như **gọi API**, **cập nhật DOM**, hoặc các thao tác ngoài React.

Cú pháp:

```
useEffect(() => {
 // thực hiện tác vụ
}, [dependencies]);
```

- Hàm trong **useEffect** sẽ được gọi mỗi khi **component render** hoặc khi giá trị của **dependencies** thay đổi.
- **dependencies** là một mảng các giá trị mà khi thay đổi, **useEffect** sẽ được gọi lại.

Ví dụ:

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

function Timer() {
 const [seconds, setSeconds] = useState(0);

 useEffect(() => {
 const timer = setInterval(() => {
 setSeconds(prevSeconds => prevSeconds + 1);
 }, 1000);

 return () => clearInterval(timer); // cleanup khi component bị unmount
 }, []); // [] làm cho effect này chỉ chạy một lần khi component mount

 return <p>Time: {seconds}s</p>;
}
```

Giải thích:

**useEffect** sẽ thiết lập một bộ đếm thời gian khi **component** được **render** lần đầu và sẽ dừng khi **component** bị **unmount**.

### 2.3. useContext

**useContext** cho phép bạn truy cập vào một giá trị **context** trong React mà không cần phải sử dụng **Context.Consumer**.

Cú pháp:

```
const value = useContext(MyContext);
```

Ví dụ:

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

const MyContext = React.createContext();

function ChildComponent() {
 const value = useContext(MyContext);
 return <p>{value}</p>;
}

function ParentComponent() {
 return (
 <MyContext.Provider value="Hello from context">
 <ChildComponent />
 </MyContext.Provider>
);
}
```

#### Giải thích:

- **useContext** cho phép **ChildComponent** truy cập giá trị **context** mà không cần phải sử dụng **Context.Consumer**.

## 2.4. useReducer

- **useReducer** là một hook thay thế **useState** khi bạn cần quản lý state phức tạp hơn, đặc biệt khi có nhiều **state** hoặc các hành động thay đổi **state**.

Cú pháp:

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- **reducer**: hàm nhận vào state hiện tại và một action, trả về **state** mới.
- **initialState**: giá trị ban đầu của **state**.

Ví dụ:

```

import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
 switch (action.type) {
 case 'increment':
 return { count: state.count + 1 };
 case 'decrement':
 return { count: state.count - 1 };
 default:
 return state;
 }
}

function Counter() {
 const [state, dispatch] = useReducer(reducer, initialState);

 return (
 <div>
 <p>Count: {state.count}</p>
 <button onClick={() => dispatch({ type: 'increment' })}>Increment</button>
 <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</button>
 </div>
);
}

```

#### Giải thích:

- **useReducer** giúp quản lý các hành động như "increment" và "decrement" trong một state phức tạp.

## 2.5. useRef

**useRef** là hook cho phép bạn *giữ một giá trị không thay đổi qua các lần render* của component. Nó có thể dùng để tham chiếu đến các phần tử **DOM** hoặc lưu trữ dữ liệu mà không gây **render lại component**.

Cú pháp:

```
const myRef = useRef(initialValue);
```

Ví dụ:



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

function FocusInput() {
 const inputRef = useRef();

 const focusInput = () => {
 inputRef.current.focus();
 };

 return (
 <div>
 <input ref={inputRef} />
 <button onClick={focusInput}>Focus the input</button>
 </div>
);
}
```

**Giải thích:**

- **useRef** được sử dụng để tham chiếu đến phần tử **input**. Khi người dùng nhấn nút, **inputRef.current.focus()** sẽ đặt con trỏ vào ô input.

## 2.6. useMemo

**useMemo** dùng để tối ưu hóa việc tính toán các giá trị phức tạp, chỉ tính toán lại khi các giá trị trong mảng dependencies thay đổi.

**Cú pháp:**

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Ví dụ:

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

function ExpensiveComputation({ a, b }) {
 const computedValue = useMemo(() => {
 console.log('Computing...');
 return a * b;
 }, [a, b]);

 return <p>Computed value: {computedValue}</p>;
}
```

**Giải thích:**

- **useMemo** giúp tránh việc tính toán lại giá trị **computedValue** khi các giá trị **a** và **b** không thay đổi.

## 2.7. Bài tập

**Bài tập 1:** xây dựng một ứng dụng **"To-Do List"** với khả năng thêm, xoá, đánh dấu hoàn thành và lưu trữ trạng thái của các công việc.

### Yêu cầu:

1. Sử dụng **useState** để quản lý danh sách công việc và trạng thái của các công việc (hoàn thành hay chưa).
2. Sử dụng **useEffect** để lưu danh sách công việc vào **localStorage** mỗi khi danh sách thay đổi, và lấy danh sách từ **localStorage** khi ứng dụng khởi động.
3. Sử dụng **useRef** để quản lý tham chiếu đến ô input khi người dùng thêm công việc mới.
4. Sử dụng **useMemo** để tối ưu việc lọc các công việc đã hoàn thành và chưa hoàn thành.
5. Sử dụng **useReducer** để quản lý các hành động như thêm, xoá và thay đổi trạng thái công việc.

### Thực thi ứng dụng:

- Thêm công việc vào danh sách bằng cách nhập vào ô input và nhấn nút **"Add Todo"**.
- Đánh dấu công việc là đã hoàn thành bằng cách nhấn vào **tên công việc**.
- Xoá công việc bằng cách nhấn vào nút **"Delete"**.

### Hướng dẫn code tham khảo

```
import React, { useState, useEffect, useRef, useMemo, useReducer } from 'react';

// Hàm reducer để xử lý các hành động
const todoReducer = (state, action) => {
 switch (action.type) {
 case 'ADD':
 return [...state, action.payload];
 case 'TOGGLE':
 return state.map(todo =>
 todo.id === action.payload ? { ...todo, completed: !todo.completed } : todo
);
 case 'DELETE':
 return state.filter(todo => todo.id !== action.payload);
 default:
 return state;
 }
};
```

```
const App = () => {
 // State quản lý danh sách công việc và giá trị của ô input
 const [todos, dispatch] = useReducer(todoReducer, []);
 const [inputValue, setInputValue] = useState('');
 const inputRef = useRef();
```

```
// Lưu danh sách công việc vào localStorage khi danh sách thay đổi
useEffect(() => {
 const storedTodos = JSON.parse(localStorage.getItem('todos'));
 if (storedTodos) {
 dispatch({ type: 'SET', payload: storedTodos });
 }
}, []);

useEffect(() => {
 localStorage.setItem('todos', JSON.stringify(todos));
}, [todos]);
```

```
// Sử dụng useMemo để lọc các công việc đã hoàn thành và chưa hoàn thành
const filteredTodos = useMemo(() => {
 return {
 incomplete: todos.filter(todo => !todo.completed),
 completed: todos.filter(todo => todo.completed),
 };
}, [todos]);
```

```
const handleAddTodo = () => {
 if (!inputValue) return;
 const newTodo = {
 id: Date.now(),
 text: inputValue,
 completed: false,
 };
 dispatch({ type: 'ADD', payload: newTodo });
 setInputValue('');
 inputRef.current.focus();
};

const handleToggleTodo = (id) => {
 dispatch({ type: 'TOGGLE', payload: id });
};

const handleDeleteTodo = (id) => {
 dispatch({ type: 'DELETE', payload: id });
};
```

## Bài tập 2

Xây dựng một ứng dụng giúp thêm, sửa, xoá và tính điểm trung bình của sinh viên

### **Yêu cầu:**

1. Sử dụng **useState** để lưu trữ danh sách sinh viên và điểm của họ.
2. Sử dụng **useEffect** để lưu trữ và tải danh sách sinh viên vào và từ `localStorage`.
3. Sử dụng **useRef** để quản lý các tham chiếu đến ô input khi người dùng thêm hoặc sửa thông tin sinh viên.
4. Sử dụng **useMemo** để tính toán điểm trung bình của các sinh viên.
5. Sử dụng **useReducer** để quản lý các hành động như thêm, sửa và xoá sinh viên.

### **Thực thi ứng dụng**

- **Thêm sinh viên:** Nhập tên và điểm của sinh viên vào ô input và nhấn "Thêm sinh viên".
- **Sửa sinh viên:** Nhấn "Sửa" trên sinh viên cần chỉnh sửa, sau đó chỉnh sửa tên và điểm và nhấn "Sửa điểm".
- **Xoá sinh viên:** Nhấn "Xoá" để xoá sinh viên khỏi danh sách.
- **Điểm trung bình:** Ứng dụng sẽ tự động tính và hiển thị điểm trung bình của tất cả các sinh viên.

---

## 3. Navigation and Routing

---

**React Router** là một thư viện điều hướng phổ biến trong React giúp quản lý nhiều trang (routes) trong một ứng dụng web mà không cần tải lại trang.

### 3.1. Các thành phần chính trong React Router

1. **BrowserRouter**: Bọc toàn bộ ứng dụng để kích hoạt tính năng routing.
2. **Routes & Route**: Xác định các đường dẫn (routes) và các component tương ứng.
3. **Link & NavLink**: Tạo liên kết để điều hướng giữa các trang mà không tải lại trang.
4. **useNavigate**: Hook giúp điều hướng trang động trong code.
5. **useParams**: Hook lấy tham số động từ URL.
6. **useLocation**: Hook lấy thông tin về URL hiện tại.

### 3.2. Các bước sử dụng React Router

#### Bước 1: Cài đặt React Router

```
npm install react-router-dom
```

#### Bước 2: Cấu hình Router trong App.js

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Home from "../pages/Home";
import About from "../pages/About";
import NotFound from "../pages/NotFound";
function App() {
 return (
 <Router>
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="*" element={<NotFound />} />
 </Routes>
 </Router>
);
}
export default App;
```

### 3.3. Điều hướng giữa các trang

#### Cách 1: Sử dụng <Link> hoặc <NavLink>

- **<Link>**: Điều hướng mà không làm mới trang.
- **<NavLink>**: Giống <Link>, nhưng có thể thêm class "active" khi trang hiện tại khớp với đường dẫn.

```
import { Link, NavLink } from "react-router-dom";
function Navbar() {
 return (
 <nav>
 <Link to="/">Home</Link>
 <NavLink to="/about"
activeClassName="active">About</NavLink>
 </nav>
);
}
```

## Cách 2: Sử dụng useNavigate trong JavaScript

```
import { useNavigate } from "react-router-dom";
function HomePage() {
 const navigate = useNavigate();
 const goToAbout = () => {
 navigate("/about");
 };
 return <button onClick={goToAbout}>Go to About</button>;
}
```

## 3.4. Truyền và lấy tham số từ URL

### + Truyền tham số động trong URL

```
<Route path="/product/:id" element={<ProductDetail />} />
```

### + Lấy tham số bằng useParams

```
import { useParams } from "react-router-dom";

function ProductDetail() {
 const { id } = useParams();
 return <h2>Product ID: {id}</h2>;
}
```

### + Lấy thông tin URL với useLocation

```
import { useLocation } from "react-router-dom";
function PageInfo() {
 const location = useLocation();
 return <p>Current Path: {location.pathname}</p>;
}
```

### 3.5. Redirect (Chuyển hướng)

```
import { useEffect } from "react";
import { useNavigate } from "react-router-dom";

function ProtectedPage() {
 const navigate = useNavigate();

 useEffect(() => {
 const isAuthenticated = false;
 if (!isAuthenticated) {
 navigate("/login");
 }
 }, []);

 return <h1>Protected Content</h1>;
}
```

### 3.6. Bài tập có hướng dẫn

Xây dựng ứng dụng **React Router** quản lý danh sách người dùng

#### Yêu cầu

1. Tạo ứng dụng React với React Router quản lý danh sách người dùng.
2. Gồm 3 trang chính:
  - **Trang chủ (/):** Hiển thị danh sách người dùng.
  - **Trang chi tiết người dùng (/user/:id):** Hiển thị thông tin chi tiết của người dùng.
  - **Trang không tìm thấy (\*):** Hiển thị khi URL không hợp lệ.
3. Khi click vào một người, chuyển đến trang chi tiết của người đó.

#### Hướng dẫn

##### 1. Cấu trúc thư mục

```
src/
├── components/
│ ├── UserList.jsx
│ └── UserDetail.jsx
├── pages/
│ ├── Home.jsx
│ └── NotFound.jsx
├── App.jsx
└── main.jsx
```

## 2. Cấu hình React Router

### Cấu hình Router trong App.jsx

```
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import Home from "../pages/Home";
import UserDetail from "../components/UserDetail";
import NotFound from "../pages/NotFound";

function App() {
 return (
 <Router>
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/user/:id" element={<UserDetail />} />
 <Route path="*" element={<NotFound />} />
 </Routes>
 </Router>
);
}

export default App;
```

## 3. Tạo các component

+ Trang chủ Home.jsx



```
import UserList from "../components/UserList";

function Home() {
 return (
 <div>
 <h1>Danh sách người dùng</h1>
 <UserList />
 </div>
);
}

export default Home;
```

#### + Danh sách người dùng `UserList.jsx`

```
import { Link } from "react-router-dom";
const users = [
 { id: 1, name: "Nguyễn Văn A" },
 { id: 2, name: "Trần Thị B" },
 { id: 3, name: "Lê Văn C" }
];
function UserList() {
 return (

 {users.map((user) => (
 <li key={user.id}>
 <Link to={` /user/${user.id}`}>{user.name}</Link>

))}

);
}
export default UserList;
```

- Hiển thị danh sách người dùng.
- Sử dụng `<Link>` để điều hướng đến trang chi tiết (`/user/:id`)

#### + Trang chi tiết người dùng `UserDetail.jsx`

```
import { useParams } from "react-router-dom";
const users = [
 { id: 1, name: "Nguyễn Văn A", age: 25, email: "a@example.com" },
 { id: 2, name: "Trần Thị B", age: 30, email: "b@example.com" },
 { id: 3, name: "Lê Văn C", age: 28, email: "c@example.com" }
];
function UserDetail() {
 const { id } = useParams();
 const user = users.find((u) => u.id === parseInt(id));
 if (!user) {
 return <h2>Người dùng không tồn tại</h2>;
 }
 return (
 <div>
 <h1>{user.name}</h1>
 <p>Tuổi: {user.age}</p>
 <p>Email: {user.email}</p>
 </div>
);
}
export default UserDetail;
```

#### + Lấy id từ URL bằng `useParams()`.

- Tìm kiếm user theo id.
- Nếu không tìm thấy, hiển thị lỗi.

#### + Trang lỗi `NotFound.jsx`

```
function NotFound() {
 return <h1>404 - Trang không tồn tại!</h1>;
}
export default NotFound;
```

## 4. Cập nhật `main.jsx`

Trong Vite, `main.jsx` là nơi bọc ứng dụng bằng **React Router**:

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root")).render(
 <React.StrictMode>
 <App />
 </React.StrictMode>
);
```

### 3.7. Bài tập sinh viên tự làm

Tạo một ứng dụng Web Travel App, với 4 trang: Home, About, Contact, Login, với component Header chứa menu và footer, xuất hiện ở tất cả các trang, điều hướng trang với router, sử dụng react bootstrap trang trí menu, trang Login có form đăng nhập gồm username và Password, sử dụng react hook

**Sinh viên tự trình bày, bổ sung nội dung của mỗi trang, sử dụng react - bootstrap**

Gợi ý:

```
travel-app/
├── public/
│ └── index.html
├── src/
│ ├── components/
│ │ ├── Header.jsx
│ │ └── Footer.jsx
│ ├── pages/
│ │ ├── Home.jsx
│ │ ├── About.jsx
│ │ ├── Contact.jsx
│ │ └── Login.jsx
│ ├── App.jsx
│ └── main.jsx
├── package.json
└── vite.config.js
```

Gợi ý

```
import { BrowserRouter as Router, Route, Routes, Link } from "react-router-dom";
import { useState } from "react";
import "bootstrap/dist/css/bootstrap.min.css";
import { Container, Form, Button, Navbar, Nav } from "react-bootstrap";
```

```
function Header() {
 return (
 <Navbar bg="dark" variant="dark" expand="lg">
```

```

 <Container>
 <Navbar.Brand as={Link} to="/">Travel App</Navbar.Brand>
 <Navbar.Toggle aria-controls="basic-navbar-nav" />
 <Navbar.Collapse id="basic-navbar-nav">
 <Nav className="me-auto">
 <Nav.Link as={Link} to="/">Home</Nav.Link>
 <Nav.Link as={Link} to="/about">About</Nav.Link>
 <Nav.Link as={Link} to="/contact">Contact</Nav.Link>
 <Nav.Link as={Link} to="/login">Login</Nav.Link>
 </Nav>
 </Navbar.Collapse>
 </Container>
 </Navbar>
);
}

```

---

```

function Footer() {
 return (
 <footer className="text-center mt-4 p-3 bg-dark text-white">
 © 2024 Travel App. All Rights Reserved.
 </footer>
);
}

```

---

```

function Home() {
 return <Container className="mt-4"><h1>Welcome to Travel
App</h1></Container>;
}

```

---

```

function About() {
 return <Container className="mt-4"><h1>About Us</h1></Container>;
}

```

---

```

function Contact() {
 return <Container className="mt-4"><h1>Contact
Us</h1></Container>;
}

```

---

```

function Login() {
 const [username, setUsername] = useState("");
 const [password, setPassword] = useState("");

 const handleSubmit = (e) => {
 e.preventDefault();
 alert(`Logging in with Username: ${username}`);
 };
}

```

---

```

function App() {
 return (

```

```
<Router>
 <Header />
 <Routes>
 <Route path="/" element={<Home />} />
 <Route path="/about" element={<About />} />
 <Route path="/contact" element={<Contact />} />
 <Route path="/login" element={<Login />} />
 </Routes>
 <Footer />
</Router>
);
}
```

```
export default App;
```

---

---

## 4. State Management

---

### 4.1. Context API

- **Khái niệm**

Context API là một tính năng của React dùng để truyền dữ liệu xuống nhiều cấp của component tree mà không cần sử dụng **props** drilling (truyền props qua nhiều component trung gian).

- **Cách hoạt động**

Context API gồm ba thành phần chính:

- **React.createContext()**: Tạo một Context object.
- **Provider**: Cung cấp dữ liệu cho các **component** con thông qua **value**.
- **Consumer (useContext)**: Nhận dữ liệu từ Context.

### Ví dụ

Giả sử ta có một ứng dụng cần chia sẻ trạng thái theme (dark/light) giữa nhiều component.

```
import React, { createContext, useContext, useState } from "react";
```

```
// Tạo Context
```

```
const ThemeContext = createContext();
```

```
export function ThemeProvider({ children }) {
 const [theme, setTheme] = useState("light");
 return (
 <ThemeContext.Provider value={{ theme, setTheme }}>
 {children}
 </ThemeContext.Provider>
);
};
```

```
// Component con sử dụng useContext để lấy theme
```

```
function ThemeSwitcher() {
 const { theme, setTheme } = useContext(ThemeContext);

 return (
 <button onClick={() => setTheme(theme === "light" ? "dark":
"light")}>
 Switch to {theme === "light"? "dark": "light"} mode
 </button>
);
}
```

```
export default function App() {
 return (
 <ThemeProvider>
```

```

 <ThemeSwitcher />
 </ThemeProvider>
);
}

```

### Sử dụng Context API:

- Khi cần chia sẻ state giữa nhiều component mà không muốn truyền props qua nhiều cấp.
- Khi state đơn giản và không cần middleware như Redux (ví dụ: theme, ngôn ngữ, trạng thái xác thực)

## 4.2. Redux

- **Khái niệm**

Redux là một thư viện quản lý state tập trung giúp ứng dụng dễ dàng quản lý và cập nhật dữ liệu nhất quán trên toàn bộ ứng dụng.

- **Cách hoạt động**

#### Redux có ba phần chính:

- + **Store:** Lưu trữ **state** toàn cục.
- + **Actions:** Mô tả các sự kiện cần thay đổi state.
- + **Reducers:** Hàm thuần nhận state hiện tại và action rồi trả về **state** mới.

#### Ví dụ

Cấu trúc một **Redux store** đơn giản để quản lý danh sách sản phẩm:

```

import { createStore } from "redux";

// 1. Định nghĩa action types
const ADD_PRODUCT = "ADD_PRODUCT";
// 2. Tạo action
const addProduct = (product) => ({
 type: ADD_PRODUCT,
 payload: product,
});

// 3. Tạo reducer
const initialState = { products: [] };

function productReducer(state = initialState, action) {
 switch (action.type) {
 case ADD_PRODUCT:
 return { ...state, products: [...state.products,
 action.payload] };
 default:
 return state;
 }
}

```

```
// 4. Tạo store
const store = createStore(productReducer);

// 5. Dispatch action
store.dispatch(addProduct({ id: 1, name: "Laptop" }));
console.log(store.getState()); // { products: [{ id: 1, name: "Laptop" }] }
```

### Sử dụng Redux

- Khi ứng dụng có nhiều state phức tạp và cần quản lý tập trung.
- Khi nhiều component cần truy cập hoặc cập nhật chung một state.
- Khi cần ghi log, undo/redo, hoặc kiểm soát state thông qua middleware như Redux Thunk hoặc Redux Saga.

### 4.3. Redux Toolkit

- **Khái niệm**  
Redux Toolkit (RTK) là một thư viện giúp đơn giản hóa Redux, giảm boilerplate code và tối ưu hiệu suất.
  - **Cách hoạt động**  
**RTK cung cấp:**
    - + **configureStore()**: Giúp tạo store nhanh hơn.
    - + **createSlice()**: Kết hợp reducer và action vào một nơi duy nhất.
    - + **createAsyncThunk()**: Hỗ trợ xử lý async actions dễ dàng.
- Ví dụ: quản lý danh sách sản phẩm bằng Redux Toolkit:

```
import { configureStore, createSlice } from "@reduxjs/toolkit";

// 1. Tạo slice
const productSlice = createSlice({
 name: "products",
 initialState: { items: [] },
 reducers: {
 addProduct: (state, action) => {
 state.items.push(action.payload);
 },
 },
});

// 2. Xuất action & reducer
export const { addProduct } = productSlice.actions;
const productReducer = productSlice.reducer;

// 3. Tạo store
const store = configureStore({
 reducer: { products: productReducer },
```



```
});
```

```
// 4. Dispatch action
store.dispatch(addProduct({ id: 1, name: "Smartphone" }));
console.log(store.getState()); // { products: { items: [{ id: 1,
name: "Smartphone" }] } }
```

## Sử dụng Redux Toolkit

- Khi sử dụng Redux nhưng muốn code ngắn gọn, dễ hiểu hơn.
- Khi cần tích hợp xử lý bất đồng bộ (fetch API, xử lý dữ liệu từ server) với `createAsyncThunk`.
- Khi làm việc với Redux và muốn có hiệu suất tối ưu mà không cần cấu hình nhiều.

## So sánh Context API vs Redux vs Redux Toolkit

Tiêu chí	Context API	Redux	Redux Toolkit
Mức độ phức tạp	Đơn giản	Phức tạp hơn	Đơn giản hơn Redux
Tính tập trung	Không có store trung tâm	Có store tập trung	Có store tập trung
Boilerplate code	Ít	Nhiều	Ít hơn Redux
Hiệu suất	Tốt cho state nhỏ	Tốt cho state lớn	Tối ưu hơn Redux truyền thống
Middleware	Không hỗ trợ tốt	Hỗ trợ (Redux Thunk, Saga)	Hỗ trợ (tích hợp sẵn Thunk)
Trường hợp sử dụng	Theme, auth, language	Ứng dụng lớn, phức tạp	Ứng dụng lớn, cần đơn giản hóa

## 4.4. Bài tập

### Bài tập 1: Quản lý giỏ hàng với Context API

Xây dựng một ứng dụng giỏ hàng đơn giản, người dùng có thể:

1. Thêm sản phẩm vào giỏ hàng
2. Xóa sản phẩm khỏi giỏ hàng
3. Xem tổng số lượng sản phẩm trong giỏ hàng

### Bước 1: Tạo `CartContext.jsx` để quản lý giỏ hàng

Tạo file `src/CartContext.jsx`:

```
import { createContext, useState, useContext } from "react";
```

```

// 1. Tạo context
const CartContext = createContext();

// 2. Provider để bọc ứng dụng
export const CartProvider = ({ children }) => {
 const [cart, setCart] = useState([]);

 // Thêm sản phẩm vào giỏ hàng
 const addToCart = (product) => {
 setCart((prevCart) => [...prevCart, product]);
 };

 // Xóa sản phẩm khỏi giỏ hàng
 const removeFromCart = (id) => {
 setCart((prevCart) => prevCart.filter((product) => product.id
 !== id));
 };

 // Đếm tổng số sản phẩm trong giỏ
 const totalItems = cart.length;

 return (
 <CartContext.Provider value={{ cart, addToCart,
 removeFromCart, totalItems }}>
 {children}
 </CartContext.Provider>
);
};

// 3. Custom hook để sử dụng CartContext dễ dàng hơn
export const useCart = () => useContext(CartContext);

```

## Bước 2: Sử dụng CartProvider trong App.jsx

```

import { CartProvider } from "../CartContext";
import ProductList from "../ProductList";
import Cart from "../Cart";
function App() {
 return (
 <CartProvider>
 <div style={{ padding: "20px" }}>
 <h1> Shopping Cart</h1>
 <ProductList />
 <Cart />
 </div>
 </CartProvider>
);
}
export default App;

```

## Bước 3: Tạo danh sách sản phẩm (ProductList.jsx)

```

import { useCart } from "../CartContext";

const products = [
 { id: 1, name: "iPhone 15", price: 999 },
 { id: 2, name: "Samsung Galaxy S24", price: 899 },
 { id: 3, name: "Google Pixel 8", price: 799 }
];

const ProductList = () => {
 const { addToCart } = useCart();

 return (
 <div>
 <h2>📦 Available Products</h2>
 {products.map((product) => (
 <div
 key={product.id}
 style={{
 display: "flex",
 justifyContent: "space-between",
 alignItems: "center",
 padding: "10px",
 margin: "5px 0",
 border: "1px solid #ddd"
 }}
 >
 {product.name} - ${product.price}
 <button onClick={() => addToCart(product)}>Add to
 Cart</button>
 </div>
))}
 </div>
);
};

export default ProductList;

```

#### Bước 4: Hiển thị giỏ hàng (Cart.jsx)

```

import { useCart } from "../CartContext";

const Cart = () => {
 const { cart, removeFromCart, totalItems } = useCart();
 return (
 <div>
 <h2>Cart ({totalItems} items)</h2>
 {cart.length === 0 ? (
 <p>Your cart is empty</p>
) : (
 cart.map((product) => (

```

```

 <div
 key={product.id}
 style={{
 display: "flex",
 justifyContent: "space-between",
 alignItems: "center",
 padding: "10px",
 margin: "5px 0",
 border: "1px solid #ddd"
 }}
 >
 {product.name} - ${product.price}
 <button onClick={() =>
removeFromCart(product.id)}>Remove</button>
 </div>
))
)}
 </div>
);
};

export default Cart;

```

### **yêu cầu thực thi:**

1. Click "Add to Cart" để thêm sản phẩm vào giỏ.
2. Click "Remove" để xóa sản phẩm khỏi giỏ.
3. Số lượng sản phẩm trong giỏ sẽ tự động cập nhật.

### **Bài tập 2: Sinh viên tự làm**

Xây dựng một hệ thống quản lý đơn hàng trong một cửa hàng thương mại điện tử.

Ứng dụng sẽ có các tính năng sau:

1. Thêm sản phẩm vào giỏ hàng
2. Xóa sản phẩm khỏi giỏ hàng
3. Cập nhật số lượng sản phẩm trong giỏ hàng
4. Tính tổng tiền đơn hàng
5. Xác nhận đơn hàng và hiển thị danh sách đơn hàng đã đặt

---

## 5. Fetch API và Axios

---

**Fetch API và Axios** đều là các công cụ dùng để thực hiện các yêu cầu HTTP trong JavaScript. Chúng giúp gửi và nhận dữ liệu từ máy chủ, thường được sử dụng trong các ứng dụng web để giao tiếp với API.

Đặc điểm	Fetch API	Axios
Cung cấp sẵn trong JS?	Có (built-in)	Không (cần cài đặt)
Hỗ trợ Promises?	Có	Có (với cú pháp đơn giản hơn)
Xử lý JSON tự động?	Không (phải dùng <code>.json()</code> )	Có (tự động parse JSON)
Xử lý lỗi HTTP?	Không tự động báo lỗi nếu HTTP status không phải 200	Tự động báo lỗi nếu HTTP status không phải 200
Hỗ trợ hủy request?	Không	Có
Hỗ trợ timeout?	Không	Có
Hỗ trợ request/response interception?	Không	Có

### 5.1. Fetch API

**Xử lý bất đồng bộ trong Javascript:** muốn xử lý code nhanh, câu lệnh nào nhanh thì chạy trước, thì có những câu lệnh chạy rất nhanh, còn những câu lệnh mất nhiều thời gian nghĩa là các câu lệnh chạy song song. Nên cần có cơ chế xử lý bất đồng bộ.

**Fetch API** là một giao diện hiện đại của JavaScript dùng để gửi yêu cầu HTTP (GET, POST, PUT, DELETE,...) và xử lý phản hồi từ máy chủ.

Nó thay thế XMLHttpRequest cũ, giúp code trở nên ngắn gọn, dễ hiểu hơn nhờ cú pháp **Promise-based** và **async/await**.

**Fetch API** là một phương thức có sẵn trong JavaScript để gửi yêu cầu HTTP. Nó trả về một **Promise** và sử dụng cú pháp `fetch(url, options)`.

#### Ví dụ Fetch API

##### 5.1.1. Gửi yêu cầu GET để lấy dữ liệu từ API

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
 .then(response => response.json()) // Chuyển đổi dữ liệu JSON
 .then(data => console.log(data)) // Hiển thị dữ liệu
 .catch(error => console.error('Lỗi:', error)); // Xử lý lỗi
```

**Giải thích:**

## 1. fetch(url)

- Hàm fetch() được sử dụng để gửi một yêu cầu HTTP đến địa chỉ **https://jsonplaceholder.typicode.com/posts/1**.
- fetch() trả về một **Promise**, giúp xử lý các tác vụ bất đồng bộ.

## 2. .then(response => response.json())

- Khi yêu cầu HTTP hoàn tất, nó trả về một **đối tượng Response**.
- .json() là một phương thức của Response, giúp chuyển đổi dữ liệu từ dạng JSON sang một đối tượng JavaScript.
- Phương thức này cũng trả về một **Promise**, chứa dữ liệu đã được chuyển đổi.

## 3. .then(data => console.log(data))

- Khi JSON đã được chuyển đổi thành đối tượng JavaScript, nó được truyền vào biến data.
- console.log(data) in dữ liệu ra màn hình console.

## 4. .catch(error => console.error('Lỗi:', error))

- Nếu có bất kỳ lỗi nào xảy ra trong quá trình gửi yêu cầu hoặc xử lý dữ liệu, .catch() sẽ bắt lỗi đó và hiển thị thông báo "Lỗi:" kèm theo thông tin lỗi trong console.

### 5.1.2. Gửi yêu cầu POST (Gửi dữ liệu lên server)

```
axios.post('https://jsonplaceholder.typicode.com/posts', {
 title: 'Axios Example',
 body: 'Nội dung bài viết',
 userId: 1
})
.then(response => console.log('Đã tạo:', response.data))
.catch(error => console.error('Lỗi:', error));
```

#### Giải thích

- + **axios.post(url, data)** gửi yêu cầu POST với dữ liệu JSON.
- + **Axios** tự động thêm **Content-Type: application/json**.

### 5.1.3. Sử dụng với async/await

async/await là một cách viết **đễ đọc hơn** để xử lý các tác vụ **bất đồng bộ (asynchronous)** trong JavaScript. Nó giúp thay thế việc sử dụng **Promise** với **.then()** và **.catch()**, làm cho code trở nên rõ ràng và dễ bảo trì hơn.

- **async** được sử dụng để **định nghĩa một hàm bất đồng bộ (asynchronous function)**.
- Một hàm có từ khóa **async** sẽ luôn trả về một **Promise**.

Ví dụ:

```
async function getData() {
 return "Hello World";
}

getData().then(console.log); // Output: "Hello World"
```

Hàm `getData()` tự động trả về một **Promise** chứa giá trị "Hello World".

- **await** chỉ có thể được sử dụng bên trong một **hàm async**.
- Nó giúp **tạm dừng** việc thực thi của hàm async cho đến khi một **Promise** được **giải quyết (resolved)**.
- `await` giúp code **trông giống như đồng bộ (synchronous)**, nhưng thực tế vẫn là **bất đồng bộ**.

ví dụ:

```
async function fetchData() {
 let response = await
 fetch('https://jsonplaceholder.typicode.com/posts/1');
 let data = await response.json();
 console.log(data);
}
fetchData();
```

- `async/await` giúp code dễ đọc hơn so với `.then()`

ví dụ: Cách viết bằng `.then()`

```
fetch('https://jsonplaceholder.typicode.com/posts/1')
 .then(response => response.json())
 .then(data => console.log(data))
 .catch(error => console.error('Lỗi:', error));
```

**Ví dụ: cách viết bằng `async/await`**

```
async function fetchData() {
 try {
 let response = await
 fetch('https://jsonplaceholder.typicode.com/posts/1');
 let data = await response.json();
 console.log(data);
 } catch (error) {
 console.error('Lỗi:', error);
 }
}

fetchData();
```

## 5.2. Axios

Axios là một thư viện HTTP client dựa trên XMLHttpRequest, có nhiều tính năng mạnh hơn Fetch API.

- **Cài đặt Axios**

Nếu dùng Node.js hoặc trình duyệt, bạn cần cài đặt Axios: `npm install axios`

**Hoặc dùng trực tiếp trong HTML:**

```
<script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js">
</script>
```

- **Gửi yêu cầu GET bằng Axios**

```
axios.get("https://jsonplaceholder.typicode.com/posts/1")
.then(response => console.log(response.data))
.catch(error => console.error("Lỗi:", error));
+ axios.get(url): Gửi yêu cầu GET.
+ response.data: Axios tự động parse JSON.
```

- **Gửi yêu cầu POST bằng Axios**

```
axios.post("https://jsonplaceholder.typicode.com/posts", {
 title: "Bài viết mới",
 body: "Nội dung bài viết",
 userId: 1
})
.then(response => console.log(response.data))
.catch(error => console.error("Lỗi:", error));
Axios không cần JSON.stringify() vì nó tự động chuyển đổi dữ liệu.
```

- **Xử lý lỗi trong Axios**

```
+ Axios tự động báo lỗi nếu HTTP status không phải 200:
axios.get("https://jsonplaceholder.typicode.com/posts/9999")
.then(response => console.log(response.data))
.catch(error => {
 if (error.response) {
 console.error(`Lỗi HTTP: ${error.response.status}`);
 } else {
 console.error("Lỗi mạng:", error.message);
 }
});
```

## 5.3. So sánh chi tiết Fetch API vs Axios

Tiêu chí	Fetch API	Axios
Dễ sử dụng	Phức tạp hơn (phải xử lý JSON, lỗi thủ công)	Dễ dùng hơn (tự động xử lý JSON, lỗi)



<b>Tính năng nâng cao</b>	Không hỗ trợ hủy request, timeout	Hỗ trợ hủy request, timeout
<b>Intercept Requests/Responses</b>	Không hỗ trợ	Có hỗ trợ
<b>Hỗ trợ trình duyệt cũ</b>	Không hỗ trợ IE11	Hỗ trợ IE11 (dùng XMLHttpRequest)

#### 5.4. Khi nào nên dùng Fetch API và khi nào nên dùng Axios?

- **Dùng Fetch API nếu:**
  - Bạn muốn code nhẹ nhàng, không cần thư viện bên ngoài.
  - Bạn đang làm dự án nhỏ, không cần xử lý lỗi nâng cao.
- **Dùng Axios nếu:**
  - Bạn muốn code dễ đọc hơn, xử lý lỗi đơn giản.
  - Bạn cần các tính năng như timeout, interceptors, hoặc hủy request.

#### 5.5. Bài tập

##### Bài 1:

- Sử dụng fetch để lấy dữ liệu từ API công khai (JSONPlaceholder hoặc [The Dog API](#)). Hiển thị danh sách dữ liệu từ API trên giao diện.
- Hiển thị danh sách bài viết hoặc hình ảnh từ API.
- Có nút "Tải lại" để gọi API và cập nhật dữ liệu.
- Sử dụng useState, useEffect để quản lý trạng thái.
- Viết code có cấu trúc rõ ràng, có thể tái sử dụng.

Hướng dẫn:

Tạo component **PostList.jsx** để hiển thị danh sách bài viết

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

const PostList = () => {
 const [posts, setPosts] = useState([]);
 const [loading, setLoading] = useState(true);

 const fetchPosts = async () => {
 setLoading(true);
 try {
 const response = await
fetch("https://jsonplaceholder.typicode.com/posts");
 const data = await response.json();
 setPosts(data.slice(0, 10)); // Lấy 10 bài viết đầu tiên
 } catch (error) {

```

```

 console.error("Lỗi khi lấy dữ liệu:", error);
 } finally {
 setLoading(false);
 }
};

useEffect(() => {
 fetchPosts();
}, []);

return (
 <div>
 <h2>Danh sách bài viết</h2>
 <button onClick={fetchPosts}>Tải lại</button>
 {loading ? <p>Đang tải...</p> : (

 {posts.map(post => (
 <li key={post.id}>
 {post.title}
 <p>{post.body}</p>

))}

)}
 </div>
);
};

export default PostList;

```

---

## Sử dụng component trong App.jsx

```
import PostList from "../PostList";
```

```

function App() {
 return (
 <div>
 <h1>Ứng dụng Fetch API</h1>
 <PostList />
 </div>
);
}

```

```
export default App;
```

**Bài 2:** Tạo một ứng dụng React trên Vite để hiển thị danh sách người dùng từ API công khai.

Phần 1:

Yêu cầu:

### 1. Fetch API

- Gọi API từ <https://jsonplaceholder.typicode.com/users>.
- Hiển thị danh sách người dùng trên giao diện.

## 2. Hiển thị danh sách

- Mỗi người dùng hiển thị **tên**, **email**, và **địa chỉ (city)**.
- Cho phép tìm kiếm người dùng theo tên bằng thanh input.

## 3. Thiết kế giao diện

- Sử dụng Tailwind CSS hoặc CSS module.
- Giao diện tối giản, dễ sử dụng.

Thêm cấu hình vào `tailwind.config.js`:

```
/** @type {import('tailwindcss').Config} */
export default {
 content: ["/index.html", "./src/**/*.{js,ts,jsx,tsx}"],
 theme: {
 extend: {},
 },
 plugins: [],
};
```

---

### Tạo component `UserList.jsx`

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

const UserList = () => {
 const [users, setUsers] = useState([]);
 const [search, setSearch] = useState("");

 useEffect(() => {
 fetch("https://jsonplaceholder.typicode.com/users")
 .then((response) => response.json())
 .then((data) => setUsers(data))
 .catch((error) => console.error("Lỗi khi lấy dữ liệu:",
error));
 }, []);

 const filteredUsers = users.filter((user) =>
 user.name.toLowerCase().includes(search.toLowerCase())
);

 return (
 <div className="p-4">
 <input
 type="text"
 placeholder="Tìm kiếm..."
 className="border p-2 w-full mb-4"
 value={search}
 />
 </div>
);
};
```

```

 onChange={(e) => setSearch(e.target.value)}
 />
 <ul className="space-y-2">
 {filteredUsers.map((user) => (
 <li key={user.id} className="p-3 border rounded
shadow">
 <h2 className="font-bold">{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>City: {user.address.city}</p>

))}

 </div>
);
};

```

```
export default UserList;
```

---

### Sử dụng trong App.jsx

```

import UserList from "../UserList";

function App() {
 return (
 <div className="container mx-auto mt-10">
 <h1 className="text-2xl font-bold text-center">Danh sách
người dùng</h1>
 <UserList />
 </div>
);
}

export default App;

```

---

## Phần 2

1. Thêm chức năng hiển thị chi tiết người dùng khi click vào.
2. Sử dụng Axios thay vì Fetch API.
3. Thêm hiệu ứng loading khi fetch dữ liệu.

Hướng dẫn

- Cài đặt Axios: **npm install axios**

1. Sử dụng **Axios** để gọi API.
2. Thêm hiệu ứng **loading** khi fetch dữ liệu.
3. Khi click vào người dùng, hiển thị **chi tiết người dùng**.

```

import { useState, useEffect } from "react";
import axios from "axios";

const UserList = () => {
 const [users, setUsers] = useState([]);

```

```

const [selectedUser, setSelectedUser] = useState(null);
const [search, setSearch] = useState("");
const [loading, setLoading] = useState(true);

useEffect(() => {
 axios
 .get("https://jsonplaceholder.typicode.com/users")
 .then((response) => {
 setUsers(response.data);
 setLoading(false);
 })
 .catch((error) => {
 console.error("Lỗi khi lấy dữ liệu:", error);
 setLoading(false);
 });
}, []);

const filteredUsers = users.filter((user) =>
 user.name.toLowerCase().includes(search.toLowerCase())
);

return (
 <div className="p-4">
 <input
 type="text"
 placeholder="Tìm kiếm..."
 className="border p-2 w-full mb-4"
 value={search}
 onChange={(e) => setSearch(e.target.value)}
 />

 {loading ? (
 <p className="text-center text-gray-500">Đang tải dữ
liệu...</p>
) : (
 <ul className="space-y-2">
 {filteredUsers.map((user) => (
 <li
 key={user.id}
 className="p-3 border rounded shadow cursor-pointer
hover:bg-gray-100"
 onClick={() => setSelectedUser(user)}
 >
 <h2 className="font-bold">{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>City: {user.address.city}</p>

))}

)}
 </div>
)

```

```

 {selectedUser && (
 <div className="mt-6 p-4 border rounded bg-gray-50 shadow">
 <h2 className="text-lg font-bold text-blue-600">
 {selectedUser.name}
 </h2>
 <p>Email: {selectedUser.email}</p>
 <p>Phone: {selectedUser.phone}</p>
 <p>Website: {selectedUser.website}</p>
 <p>Company:
 {selectedUser.company.name}</p>
 <button
 className="mt-4 px-4 py-2 bg-red-500 text-white rounded"
 onClick={() => setSelectedUser(null)}
 >
 Đóng
 </button>
 </div>
)}
 </div>
);
};

export default UserList;

```

---

## Cập nhật App.jsx

```

import UserList from "../UserList";

function App() {
 return (
 <div className="container mx-auto mt-10">
 <h1 className="text-2xl font-bold text-center mb-4">Danh sách
 người dùng</h1>
 <UserList />
 </div>
);
}

export default App;

```

## Phần 3

1. Thêm animation khi mở chi tiết người dùng.

2. Phân trang danh sách người dùng.
3. Dùng Context API hoặc Redux để quản lý state tốt hơn.

Cài thư viện: npm install axios framer-motion

Cấu trúc thư mục

```
/src
├── components
│ ├── UserList.jsx
│ ├── UserDetails.jsx
├── context
│ ├── UserContext.jsx
├── App.jsx
└── main.jsx
```

Dùng **Context API** để quản lý danh sách người dùng và trạng thái loading.

```
import { createContext, useState, useEffect } from "react";
import axios from "axios";
```

```
export const UserContext = createContext();
```

```
export const UserProvider = ({ children }) => {
 const [users, setUsers] = useState([]);
 const [loading, setLoading] = useState(true);
```

```
 useEffect(() => {
 axios
 .get("https://jsonplaceholder.typicode.com/users")
 .then((response) => {
 setUsers(response.data);
 setLoading(false);
 })
 .catch((error) => {
 console.error("Lỗi khi lấy dữ liệu:", error);
 setLoading(false);
 });
 }, []);
```

```
 return (
 <UserContext.Provider value={{ users, loading }}>
 {children}
 </UserContext.Provider>
);
};
```

**Phân trang:** Hiển thị 5 người dùng mỗi trang.

**Chọn người dùng** để xem chi tiết.

```

import { useState, useContext } from "react";
import { UserContext } from "../context/UserContext";
import { motion } from "framer-motion";
import UserDetails from "../UserDetails";

const UserList = () => {
 const { users, loading } = useContext(UserContext);
 const [selectedUser, setSelectedUser] = useState(null);
 const [search, setSearch] = useState("");
 const [currentPage, setCurrentPage] = useState(1);
 const usersPerPage = 5;

 const filteredUsers = users.filter((user) =>
 user.name.toLowerCase().includes(search.toLowerCase())
);

 const indexOfLastUser = currentPage * usersPerPage;
 const indexOfFirstUser = indexOfLastUser - usersPerPage;
 const currentUsers = filteredUsers.slice(indexOfFirstUser,
 indexOfLastUser);

 return (
 <div className="p-4">
 <input
 type="text"
 placeholder="Tìm kiếm..."
 className="border p-2 w-full mb-4"
 value={search}
 onChange={(e) => setSearch(e.target.value)}
 />

 {loading ? (
 <p className="text-center text-gray-500">Đang tải dữ
liệu...</p>
) : (
 <>
 <ul className="space-y-2">
 {currentUsers.map((user) => (
 <motion.li
 key={user.id}
 className="p-3 border rounded shadow cursor-pointer
hover:bg-gray-100"
 onClick={() => setSelectedUser(user)}
 whileHover={{ scale: 1.02 }}
 >
 <h2 className="font-bold">{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>City: {user.address.city}</p>
 </motion.li>
))}

 </>
)}
 </div>
);
}

```



```


 {/* Phân trang */}
 <div className="flex justify-center mt-4 space-x-2">
 {[...Array(Math.ceil(filteredUsers.length /
usersPerPage))].map(
 (_, index) => (
 <button
 key={index}
 className={`px-3 py-1 border rounded ${
 currentPage === index + 1 ? "bg-blue-500 text-
white" : ""
 }`}
 onClick={() => setCurrentPage(index + 1)}
 >
 {index + 1}
 </button>
)
)}
 </div>
 </>
)}

 {selectedUser && <UserDetails user={selectedUser} onClose={()
=> setSelectedUser(null)} />}
 </div>
);
};

export default UserList;

```

---

**Tạo UserDetails.jsx** với animation

**Hiệu ứng mờ chi tiết người dùng bằng Framer Motion.**  
Click **Đóng** để quay lại danh sách.

```

import { motion } from "framer-motion";

const UserDetails = ({ user, onClose }) => {
 return (
 <motion.div
 className="fixed inset-0 flex items-center justify-center
bg-black bg-opacity-50"
 initial={{ opacity: 0 }}
 animate={{ opacity: 1 }}
 exit={{ opacity: 0 }}
 >
 <motion.div
 className="bg-white p-6 rounded shadow-lg max-w-sm w-
full"

```

```

 initial={{ y: 50, opacity: 0 }}
 animate={{ y: 0, opacity: 1 }}
 exit={{ y: 50, opacity: 0 }}
 >
 <h2 className="text-lg font-bold text-blue-600">{user.name}</h2>
 <p>Email: {user.email}</p>
 <p>Phone: {user.phone}</p>
 <p>Website: {user.website}</p>
 <p>Company: {user.company.name}</p>
 <button
 className="mt-4 px-4 py-2 bg-red-500 text-white rounded"
 onClick={onClose}
 >
 Đóng
 </button>
 </motion.div>
 </motion.div>
);
};

```

```
export default UserDetails;
```

---

Cập nhật App.jsx để dùng UserProvider

```

import { UserProvider } from "../context/UserContext";
import UserList from "../components/UserList";

function App() {
 return (
 <UserProvider>
 <div className="container mx-auto mt-10">
 <h1 className="text-2xl font-bold text-center mb-4">
 Danh sách người dùng
 </h1>
 <UserList />
 </div>
 </UserProvider>
);
}

export default App;

```

---

## 6. Bài tập Backend với Node.js API: Quản lý Sản Phẩm

---

### 6.1. Mục tiêu:

Xây dựng một RESTful API đơn giản để quản lý danh sách sản phẩm, bao gồm các chức năng:

- Thêm sản phẩm
- Lấy danh sách sản phẩm
- Lấy chi tiết một sản phẩm
- Cập nhật sản phẩm
- Xóa sản phẩm

### Công nghệ sử dụng:

- Node.js với Express.js
- Cơ sở dữ liệu giả lập bằng một mảng trong bộ nhớ

### Bước 1: Khởi tạo dự án Node.js

1. Mở terminal và chạy lệnh sau để tạo một dự án mới:

```
mkdir product-api && cd product-api
npm init -y
```

Lệnh này sẽ tạo file `package.json`.

2. Cài đặt các thư viện cần thiết:

```
npm install express nodemon
```

- **express**: Framework để tạo API
- **nodemon**: Giúp tự động **restart server** khi có thay đổi

### Bước 2: Tạo Server với Express.js

Tạo file **server.js** và thêm mã sau:

```
const express = require("express");
const app = express();
const port = 3000;

app.use(express.json()); // Middleware để đọc JSON từ request body
app.get("/", (req, res) => {
 res.send("Welcome to Product API");
});
app.listen(port, () => {
 console.log(`Server is running at http://localhost:${port}`);
});
```

Chạy server bằng lệnh:

```
nodemon server.js
```

Nếu thành công, mở trình duyệt và vào **http://localhost:3000/** sẽ thấy **Welcome to Product API**.

### Bước 3: Xây dựng API Quản lý Sản phẩm

Thêm danh sách sản phẩm giả lập vào **server.js**:

```
let products = [
 { id: 1, name: "Laptop", price: 1000 },
 { id: 2, name: "Phone", price: 500 },
];
```

#### 1. API Lấy danh sách sản phẩm

```
app.get("/products", (req, res) => {
 res.json(products);
});
```

Test bằng cách mở trình duyệt vào **http://localhost:3000/products** hoặc dùng **Postman**.

#### 2. API Lấy chi tiết một sản phẩm

```
app.get("/products/:id", (req, res) => {
 const product = products.find((p) => p.id ===
 parseInt(req.params.id));
 if (!product) return res.status(404).json({ message: "Product
not found" });
 res.json(product);
});
```

Test với URL: **http://localhost:3000/products/1**

#### 3. API Thêm sản phẩm mới

```
app.post("/products", (req, res) => {
 const { name, price } = req.body;
 if (!name || !price) return res.status(400).json({ message:
 "Invalid input" });

 const newProduct = { id: products.length + 1, name, price };
 products.push(newProduct);
 res.status(201).json(newProduct);
});
```

Dùng Postman với phương thức **POST**, gửi JSON:

```
{
 "name": "Tablet",
 "price": 700
}
```

#### 4. API Cập nhật sản phẩm

```
app.put("/products/:id", (req, res) => {
 const product = products.find((p) => p.id ===
 parseInt(req.params.id));
 if (!product) return res.status(404).json({ message: "Product
 not found" });

 const { name, price } = req.body;
 if (name) product.name = name;
 if (price) product.price = price;

 res.json(product);
});
```

Test bằng cách gửi **PUT request** với dữ liệu cập nhật.

#### 5. API Xóa sản phẩm

```
app.delete("/products/:id", (req, res) => {
 products = products.filter((p) => p.id !==
 parseInt(req.params.id));
 res.json({ message: "Product deleted" });
});
```

Test bằng **DELETE request**.

#### Bước 4: Kiểm tra API với Postman

Sau khi hoàn thành, kiểm tra bằng cách gửi request đến từng API trên Postman hoặc trình duyệt.

#### 6.2. Nâng cấp API quản lý sản phẩm với mongodb

##### Mục tiêu:

- Thay thế mảng tĩnh bằng MongoDB để lưu trữ dữ liệu thật.
  - Sử dụng thư viện mongoose để thao tác với MongoDB.
  - Hoàn thiện CRUD (Create, Read, Update, Delete) với cơ sở dữ liệu.
-

## Bước 1: Cài đặt MongoDB và Thư Viện Liên Quan

### 1. Cài đặt MongoDB

- Nếu chưa có, tải và cài đặt MongoDB từ:  
<https://www.mongodb.com/try/download/community>
  - Chạy MongoDB trên local: mongod
  - Hoặc sử dụng MongoDB Atlas (dịch vụ cloud).

### 2. Cài đặt thư viện Mongoose

```
npm install mongoose dotenv
```

- **mongoose**: Thư viện giúp kết nối và tương tác với MongoDB.
- **dotenv**: Quản lý biến môi trường (giúp bảo mật thông tin kết nối DB).

---

## Bước 2: Kết Nối MongoDB

- Tạo file **.env** để lưu thông tin kết nối:

```
MONGO_URI=mongodb://127.0.0.1:27017/product_db
PORT=3000
```

- Chỉnh sửa **server.js** để kết nối MongoDB:

```
require("dotenv").config();
const express = require("express");
const mongoose = require("mongoose");

const app = express();
const port = process.env.PORT || 3000;

// Kết nối MongoDB
mongoose
 .connect(process.env.MONGO_URI)
 .then(() => console.log("MongoDB connected"))
 .catch((err) => console.error("MongoDB connection error:",
err));
app.use(express.json());

app.get("/", (req, res) => {
 res.send("Welcome to Product API with MongoDB!");
});
app.listen(port, () => {
 console.log(`Server is running at
http://localhost:${port}`);
});
```

- Chạy server:

```
nodemon server.js
```

- Nếu kết nối thành công, terminal sẽ hiển thị:  
**MongoDB connected**

### Bước 3: Tạo Model Sản Phẩm

Tạo thư mục **models/** và file **Product.js**:

```
const mongoose = require("mongoose");

const productSchema = new mongoose.Schema({
 name: { type: String, required: true },
 price: { type: Number, required: true },
});

const Product = mongoose.model("Product", productSchema);

module.exports = Product;
```

### Bước 4: Xây Dựng API CRUD với MongoDB

#### 1. Lấy danh sách sản phẩm

```
const Product = require("./models/Product");

app.get("/products", async (req, res) => {
 try {
 const products = await Product.find();
 res.json(products);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});
```

#### 2. Lấy chi tiết một sản phẩm

```
app.get("/products/:id", async (req, res) => {
 try {
 const product = await Product.findById(req.params.id);
 if (!product) return res.status(404).json({ message: "Product not found" });
 res.json(product);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});
```

### 3. Thêm sản phẩm mới

```
app.post("/products", async (req, res) => {
 try {
 const { name, price } = req.body;
 if (!name || !price) return res.status(400).json({ message:
 "Invalid input" });

 const newProduct = new Product({ name, price });
 await newProduct.save();
 res.status(201).json(newProduct);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});
```

#### Test trên Postman

- **Method:** POST
- **URL:** `http://localhost:3000/products`
- **Body (JSON):**

```
{
 "name": "Smartwatch",
 "price": 300
}
```

### 4. Cập nhật sản phẩm

```
app.put("/products/:id", async (req, res) => {
 try {
 const { name, price } = req.body;
 const product = await Product.findByIdAndUpdate(
 req.params.id,
 { name, price },
 { new: true }
);
 if (!product) return res.status(404).json({ message: "Product
not found" });
 res.json(product);
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});
```

#### Test trên Postman

- **Method:** PUT
- **URL:** `http://localhost:3000/products/{id}`
- **Body (JSON):**



```
{
 "name": "Smartwatch Pro",
 "price": 350
}
```

## 5. Xóa sản phẩm

```
app.delete("/products/:id", async (req, res) => {
 try {
 const product = await
 Product.findByIdAndDelete(req.params.id);
 if (!product) return res.status(404).json({ message: "Product
not found" });
 res.json({ message: "Product deleted" });
 } catch (error) {
 res.status(500).json({ message: error.message });
 }
});
```

### Test trên Postman

- Method: DELETE
- URL: `http://localhost:3000/products/{id}`

---

## Bước 5: Kiểm Tra API với Postman

1. Lấy danh sách sản phẩm (GET /products)
2. Lấy chi tiết một sản phẩm (GET /products/:id)
3. Thêm sản phẩm mới (POST /products)
4. Cập nhật sản phẩm (PUT /products/:id)
5. Xóa sản phẩm (DELETE /products/:id)

### 6.3. Yêu cầu nâng cao:

**Kết nối MongoDB Atlas:** Dùng database trên cloud thay vì local.