**1.Difference between npm and npx(node package execute)**

**npm install create-react-app p1**

**or**

**npx create-react-app p1**

To understand this, I'm going to use one very simple package called Cowsay.

**npm install cowsay -g**

**cowsay hello**

Then there is a cow getting displayed and it says hello.Whatever message you have passed, in short, is displayed along with this cow image.This is how the cow say package works. Now, if I want to check whether this package is installed or not, I can use NPM list cow say if I run this command.

**npm list cowsay**

it shows empty because in the current folder I don’t have cowsay

**npm list cowsay -g**

[**--cowsay@1.5.0**](mailto:--cowsay@1.5.0)

So in my system cowsay installed and it occupy space also

Le me remove this

**npm uninstall cowsay -g**

**npx cowsay hello**

**npm list cowsay -g**

**empty**

So now cowsay is not installed it is just executed.

NPM is a package management that is used to install ,uninstall and update Javascript-package on your system.

NPX is package executor whis is used directly execute all jqavascript packages without installing them

How to create react application?

*//JSX Html inside JSX*

*//JSX will return one reusable component there we can wirte the code to define our*

*//pages*

*//Rule*

*// 1. html code should be enclosed with in return(<div></div)*

*// 2 we can use react fragment return(<></>)*

*//In jsx all the expressions enclose with in {} <h1 style={{color:red}}></h1>*

*//In jsx the calss name should follow camelCalse*

*/// <h1 className="h1"/>*

**function** App() {

  return(

<div>

<p>Welcome</p>

</div>);

}

export default App;

**How to add One Componenet into another?**

1. Create MenuComponent

**function** MenuComponent() {

  return <div>Menu</div>;

}

export default MenuComponent;//is used to ensure that our component can be used in another component

1. Create about us an dinclude in App.js

**function** Aboutus() {

  return <div>About us</div>;

}

export default Aboutus;

import Aboutus from "./aboutus";

import "./App.css";

import MenuComponent from "./MenuComponent";

**function** App() {

  return (

    <div>

      <MenuComponent />

      Welcome

      <Aboutus />

    </div>

  );

}

export default App;

1. How to pass data from one component to another component?

import "./App.css";

import Person from "./components/Person";

**function** App() {

  return (

    <div>

      <Person name={"abc"} age={23} />

    </div>

  );

}

export default App;

*/\**

*Argument passed to react componeney via HTML attribute*

*<Person name={abc} age={23}/>  function Person({name:"abc",age:"23"})*

*\*/*

**function** Person(props) {

  return (

    <div>

      name:{props.name}

      age:{props.age}

    </div>

  );

}

export default Person;

**Scenario:**  
You are building an e-commerce application where the product list is displayed on the homepage. Each product card needs to show the product image, name, price, and a button to add the product to the cart.

The ProductCard component receives the product details as props and displays them accordingly.

**Tasks:**

1. **Pass the Product Data:** Write a parent component that passes the necessary product data (e.g., name, price, image, and inStock) as props to the ProductCard component.
2. **Conditionally Render UI Elements:** In the ProductCard component, use the received props to display the product name, image, and either the price or "Sold Out" label. Also, disable the "Add to Cart" button if the product is out of stock.
3. **Handle Button Click:** Implement a callback function in the parent component that adds the product to the cart. This function should be passed as a prop to the ProductCard component and invoked when the "Add to Cart" button is clicked.

**Discussion Points:**

* How would you structure the ProductCard component to ensure it is reusable for different types of products?
* What are the advantages of passing the addToCart function as a prop instead of handling the logic directly within the ProductCard component?
* How would you test the ProductCard component to ensure it properly handles different states (e.g., in stock vs. out of stock)?

This scenario gives a practical and relatable example that ties React props to real-world project requirements.

**Answer:**

**1. Pass the Product Data**

First, let's create a parent component that will pass the necessary product data to the ProductCard component.

import React from 'react';

import ProductCard from './ProductCard';

const products = [

{

id: 1,

name: 'Smartphone',

price: 699,

image: '/images/smartphone.jpg',

inStock: true,

},

{

id: 2,

name: 'Headphones',

price: 199,

image: '/images/headphones.jpg',

inStock: false,

},

// Add more products as needed

];

const ProductList = () => {

const addToCart = (product) => {

console.log(`${product.name} added to cart!`);

// Add product to cart logic here

};

return (

<div>

{products.map((product) => (

<ProductCard

key={product.id}

name={product.name}

price={product.price}

image={product.image}

inStock={product.inStock}

addToCart={() => addToCart(product)}

/>

))}

</div>

);

};

export default ProductList;

**2. Conditionally Render UI Elements**

Now, let's implement the ProductCard component. This component will display the product details and handle the conditional rendering of the price or "Sold Out" label. The "Add to Cart" button will be disabled if the product is out of stock.

import React from 'react';

const ProductCard = ({ name, price, image, inStock, addToCart }) => {

return (

<div className="product-card">

<img src={image} alt={name} />

<h2>{name}</h2>

{inStock ? <p>${price}</p> : <p>Sold Out</p>}

<button onClick={addToCart} disabled={!inStock}>

Add to Cart

</button>

</div>

);

};

export default ProductCard;

**3. Handle Button Click**

The addToCart function is passed as a prop to the ProductCard component. When the "Add to Cart" button is clicked, this function is invoked, and the product is added to the cart.

**Discussion Points:**

**• Structuring the ProductCard Component for Reusability:**

* The ProductCard component is designed to be reusable by accepting props like name, price, image, and inStock. This allows it to be used for different products without modification.
* If different types of products have unique attributes, you can pass additional props or use children to render specific content inside the ProductCard.

**• Advantages of Passing addToCart Function as a Prop:**

* **Separation of Concerns:** Passing the addToCart function as a prop keeps the ProductCard component focused on rendering the UI, while the parent component manages the logic for adding items to the cart. This separation makes the component more reusable and easier to test.
* **Flexibility:** The parent component can control the behavior of the addToCart function, allowing different implementations for different contexts (e.g., logging, updating state, etc.).

**• Testing the ProductCard Component:**

* **Snapshot Testing:** Create snapshots of the ProductCard component in different states (e.g., in stock, out of stock) to ensure consistent rendering.
* **Unit Testing:** Use tools like Jest and React Testing Library to test the component's behavior. For example, you can simulate clicks on the "Add to Cart" button and check if the addToCart function is called only when the product is in stock.
* **Conditional Rendering:** Verify that the "Sold Out" label is displayed when inStock is false and that the "Add to Cart" button is disabled.

This scenario provides a practical example of how React props can be used in real-world projects, ensuring that the ProductCard component is flexible, reusable, and easy to maintain.

**1.Object Destructuing in React props**

function ProductCard({ name, price, image, inStock }) {*//1. Object Destructuring*

  return (

    <div>

      Name:{name}

      price:{price}

      Image:{image}

      Instock:{inStock}

    </div>

  );

}

export default ProductCard;

**2. How to display Array of values with jsx**

**function** Person() {

**const** personNames = ["abc", "bcd", "cde"];

  return (

    <div>

      <ul>

        {personNames.map((n) **=>** (

          <li>{n}</li>

        ))}

      </ul>

    </div>

  );

}

export default Person;

*/\**

*Argument passed to react componeney via HTML attribute*

*<Person name={abc} age={23}/>  function Person({name:"abc",age:"23"})*

*\*/*

**function** Person() {

**const** personNames = [

    { name: "abc", age: "23" },

    { name: "bcd", age: "23" },

  ];

  return (

    <div>

      <ul>

        {personNames.map((p) **=>** (

          <li>

            {p.name}-{p.age}

          </li>

        ))}

      </ul>

    </div>

  );

}

export default Person;

1. **Consitional Rendering**

function App() {

  return (

    <div>

      <ProductCard name={"p1"} price={23} inStock={false} />

    </div>

  );

}

function ProductCard({ name, price, image, inStock }) {

*//1. Object Destructuring*

  return (

    <div>

      Name:{name}

      {inStock ? <p>${price}</p> : "Sold out"}

    </div>

  );

}

export default ProductCard;

The **React Virtual DOM** is a key concept in React that helps improve the performance of web applications by efficiently updating the user interface (UI). Here’s a breakdown of how it works:

**What is the Virtual DOM?**

* **DOM (Document Object Model):** The DOM is a tree-like structure representing the UI of a web application. Every time a user interacts with the application, the browser needs to update the DOM to reflect changes, which can be slow if done frequently or inefficiently.
* **Virtual DOM:** The Virtual DOM is an in-memory representation of the real DOM elements created by React. Instead of directly manipulating the actual DOM, React creates a virtual copy of it. When a change occurs in the application (e.g., a user input or data update), React updates the Virtual DOM first.

**How Does the Virtual DOM Work?**

1. **Render Phase:**
   * When a component’s state or props change, React re-renders the component, creating a new Virtual DOM.
   * This new Virtual DOM is compared with the previous Virtual DOM using a process called **reconciliation**.
2. **Reconciliation:**
   * During reconciliation, React identifies the differences (or **diffs**) between the old and new Virtual DOM.
   * React then determines the minimum number of changes needed to update the actual DOM, ensuring that only the necessary parts of the UI are modified.
3. **Update Phase:**
   * React efficiently updates the real DOM with the changes, avoiding unnecessary re-renders and improving performance.

**Why is the Virtual DOM Efficient?**

* **Batch Updates:** React batches updates to the DOM, minimizing the number of reflows and repaints, which are resource-intensive operations.
* **Optimized Diffing Algorithm:** React’s diffing algorithm is highly optimized, allowing it to quickly identify changes and apply updates in a way that’s faster than directly manipulating the real DOM.
* **Selective Updates:** React ensures that only the components that actually changed are re-rendered, rather than the entire UI.

**Example**

Imagine a to-do list application where a user checks off an item as completed:

* Without the Virtual DOM, each check could potentially cause the entire list to re-render, which can be slow.
* With the Virtual DOM, React only updates the specific item that changed, resulting in a faster and smoother experience.

**Conclusion**

The Virtual DOM is a foundational concept in React that significantly boosts the performance of web applications by making UI updates more efficient and less resource-intensive. It allows developers to write declarative code without worrying about the performance costs of frequent updates to the DOM.

**State of the React Component**

import { useState } from "react";

*/\**

*<!--here textbox value is related to this componenet this is called state variable*

*To hold the data with in the component we can use state variable-->*

*\*/*

function Person() {

  let v; *//we could not maintain state of the component using this normal variable*

  const [value, setValue] = useState(0); *//is builtin function called react hook to create state variable*

*//State variable is used to maintain state of the component  here textbox value*

  const display = () => {

    setValue(document.getElementById("t1").value);

  };

  return (

    <div>

      <input type="text" id="t1" />

      <div id="result">{value}</div>

      <button onClick={display}>OK</button>

    </div>

  );

}

export default Person;

import { useEffect, useState } from "react";

**function** Sample() {

**const** [cnt, setCnt] = useState(0);

**const** increment = () **=>** {

    setCnt(cnt + 1);

*// console.log(cnt);*

  };

  useEffect(() **=>** {

    console.log(cnt);

  }, [cnt]);

  return (

    <div>

      {cnt}

      <button

        onClick={() **=>** {

          increment();

        }}

      >

        Increment

      </button>

    </div>

  );

}

export default Sample;

To build a CRUD (Create, Read, Update, Delete) application in React, you need to be familiar with several fundamental concepts. Below is a list of topics to learn, along with explanations and examples to help you understand how to use them in a CRUD application.

**1. React Components**

* **Concept:** React applications are built using components, which are reusable pieces of the UI. Components can be either functional or class-based.
* **Example:** A simple functional component to display a list of items.

function ItemList({ items }) {

return (

<ul>

{items.map((item) => (

<li key={item.id}>{item.name}</li>

))}

</ul>

);

}

**2. State Management**

* **Concept:** State is used to store data that can change over time. In a CRUD application, you need to manage the state of the data being created, read, updated, or deleted.
* **Example:** Using useState to manage a list of items.

import React, { useState } from 'react';

function App() {

const [items, setItems] = useState([]);

}

**3. Handling Forms**

* **Concept:** Forms are used to collect input from users. You'll need to learn how to create and manage forms in React to handle creating and updating items.
* **Example:** A form component to add a new item.

function AddItemForm({ onAddItem }) {

const [name, setName] = useState('');

const handleSubmit = (e) => {

e.preventDefault();

onAddItem({ id: Date.now(), name });

setName('');

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

value={name}

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

placeholder="Enter item name"

/>

<button type="submit">Add Item</button>

</form>

);

}

**4. Props and Event Handling**

* **Concept:** Props are used to pass data and event handlers between components. You'll use props to pass the current state and functions to child components.
* **Example:** Passing the onAddItem function as a prop.

function App() {

const [items, setItems] = useState([]);

const handleAddItem = (newItem) => {

setItems([...items, newItem]);

};

return (

<div>

<AddItemForm onAddItem={handleAddItem} />

<ItemList items={items} />

</div>

);

}

**5. Making API Calls with Axios or Fetch**

* **Concept:** In a real-world CRUD application, data is often stored on a server. You'll need to learn how to make HTTP requests to interact with a backend API for performing CRUD operations.
* **Example:** Using Axios to fetch data from an API.

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

import axios from 'axios';

function App() {

const [items, setItems] = useState([]);

useEffect(() => {

axios.get('/api/items')

.then((response) => setItems(response.data))

.catch((error) => console.error('Error fetching data:', error));

}, []);

return (

<div>

<ItemList items={items} />

</div>

);

}

**6. Routing with React Router**

* **Concept:** Routing allows you to navigate between different pages in your application. For CRUD, you'll often have different routes for listing items, viewing details, editing, and deleting.
* **Example:** Setting up basic routes for listing items and adding a new item.

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

function App() {

return (

<Router>

<Switch>

<Route path="/" exact component={ItemListPage} />

<Route path="/add" component={AddItemPage} />

{/\* Add more routes as needed \*/}

</Switch>

</Router>

);

}

**7. Handling Updates**

* **Concept:** Updating an item typically involves retrieving the existing data, modifying it, and then saving the changes.
* **Example:** Editing an item in the list.

javascript

Copy code

function EditItemForm({ item, onUpdateItem }) {

const [name, setName] = useState(item.name);

const handleSubmit = (e) => {

e.preventDefault();

onUpdateItem({ ...item, name });

};

return (

<form onSubmit={handleSubmit}>

<input

type="text"

value={name}

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

/>

<button type="submit">Update Item</button>

</form>

);

}

**8. Deleting Items**

* **Concept:** Deleting an item involves removing it from the state and making a delete request to the server.
* **Example:** Handling the deletion of an item.

function App() {

const [items, setItems] = useState([]);

const handleDeleteItem = (id) => {

setItems(items.filter((item) => item.id !== id));

// Also send a DELETE request to the server

};

return (

<div>

<ItemList items={items} onDeleteItem={handleDeleteItem} />

</div>

);

}

**9. Using Context API or Redux for State Management**

* **Concept:** For larger applications, managing state with useState might become cumbersome. The Context API or Redux allows you to manage global state across multiple components.
* **Example:** Using Context API to provide global state.

const ItemsContext = React.createContext();

function App() {

const [items, setItems] = useState([]);

return (

<ItemsContext.Provider value={{ items, setItems }}>

{/\* Components that need access to items will be wrapped here \*/}

</ItemsContext.Provider>

);

}

**10. Error Handling**

* **Concept:** Proper error handling is crucial to ensure the application behaves correctly when something goes wrong.
* **Example:** Handling errors during an API call.

axios.get('/api/items')

.then((response) => setItems(response.data))

.catch((error) => {

console.error('Error fetching data:', error);

// Display an error message to the user

});

By learning and applying these concepts, you will be well-equipped to build a complete CRUD application in React that interacts with an Express backend and MongoDB database.