Contents

title: React Core Concepts (Part 2) date: July 5, 2025

---

1 Connecting CSS Files in React

- Importing `index.css` in `main.jsx` applies styles globally to all components.
- Importing `App.css` in `App.jsx` scopes styles to the App component.
- Importing `Parent.css` in `Parent.jsx` scopes styles to the Parent component.
- To style any component, import its CSS file directly in the component file.

---

2 The onSale Property Example

Parent.jsx

```jsx
import React, { useState } from "react";
import Child from "../Child/Child";

export default function Parent() {
  const [allProducts, setAllProducts] = useState([
    { name: "Samsung", price: 2000, category: "Mobile", onSale: true },
    { name: "Dell", price: 3000, category: "Laptop", onSale: false },
    { name: "HP", price: 4000, category: "Laptop", onSale: true },
    { name: "LG", price: 6000, category: "PC", onSale: false },
  ]);
  return (
    <div className="container">
      <div className="row g-4 bg-red-500">
        <h2 className="text-center">Hello in Home Page</h2>
        {allProducts.map((product) => (
          <Child ProductInfo={product} />
        ))}
      </div>
    </div>
  );
}
```

Explanation:

- Uses `map` to render a `Child` for each product.
- Passes product info as props.

Child.jsx

```jsx
import React from "react";

export default function Child({ ProductInfo }) {
  const { name, price, category, onSale } = ProductInfo;
  return (
    <div className="col-md-4">
      <div className="bg-teal-400 p-4 relative">
        <h3>
          <i className="fa-brands fa-facebook text-dark pe-1"></i>
          {name}
        </h3>
        <h4>Product Price is: {price}</h4>
        <h5>Product Category is: {category}</h5>
        {onSale && (
          <div className="bg-blue-700 absolute top-0 right-2
          ↪  p-1">Sale</div>
        )}
      </div>
    </div>
  );
```

```
}
```

Explanation:

- Uses object destructuring for props.
- Conditionally renders a "Sale" tag if `onSale` is true.
- Uses utility classes for styling.

2.1 Note About the `key` Prop Warning

- React requires a unique `key` prop for each child in a list to optimize rendering.
- Using the array index as a key is not recommended; use a unique identifier like an `id` or product name (if unique).

Warning Example:

```
{
  allProducts.map((product) => <Child ProductInfo={product} />);
}
```

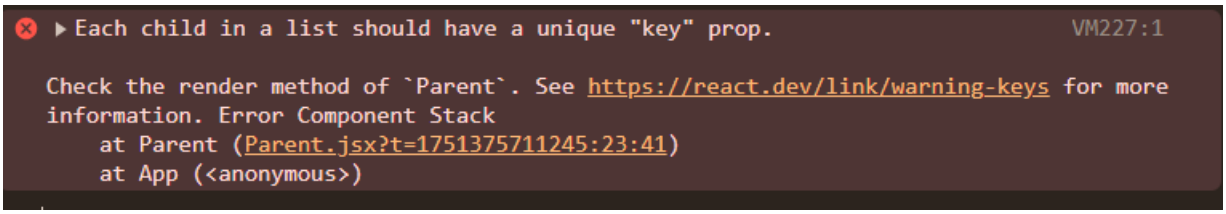

Figure 1: Unique Key Prop Warning

Solution (not recommended):

```
{
  allProducts.map((product, index) => (
    <Child key={index} ProductInfo={product} />
  ));
}
```

Best Solution:

```
{
  allProducts.map((product) => (
    <Child key={product.name} ProductInfo={product} />
  ));
}
```

- Prefer a unique `id` if available.

---

3 Best Practice: Defining Functions Near State

```
- Explain the code
```

```
  - I will use the object destructuring to get the values from the object
  - I will use the conditional rendering to show the Sale tag if the product is on sale
  - I will use the absolute position to show the Sale tag on the top right of the produc
  - I will use the Font Awesome icons to show the icon in the product name
  - I will use the Bootstrap classes to style the product card
  - I will use the Tailwind CSS classes to style the product card
  - I will use the relative position to show the Sale tag on the top right of the produc

## Note about Warning

- If you see a warning in the console about the key prop, it means that you need to add

![Unique Key Prop Warning](./Pics/Key%20Warning.png)

- This is the Case of the Warning

```jsx
{
  allProducts.map((product) => {
    return <Child ProductInfo={product} />;
  });
}
```

- This is the Solution of the Warning

```jsx
{
  allProducts.map((product, index) => {
    return <Child key={index} ProductInfo={product} />;
  });
}
```

- This is the Best Solution of the Warning
- Explain the code
  - I will use the name of the product as the key prop, but it's not recommended if you have duplicate names in your products. Instead, you should use a unique identifier for each product, such as an ID.

```jsx
{
  allProducts.map((product) => {
    return <Child key={product.name} ProductInfo={product} />;
  });
}
```

- Explain the code
  - I will use the name of the product as the key prop, but it's not recommended if you have duplicate names in your products. Instead, you should use a unique identifier for each product, such as an ID.
  - I will use the index of the array as the key prop, but it's not recommended. Instead, you should use a unique identifier for each product, such as an ID.

```

4 Creating Delete Function in The Recommended Place

- **Note On Best Practice:** If a function modifies or depends on a state variable, define that function in the same component where the state is declared. This ensures proper access to the state and keeps your component logic organized and maintainable.
- Parent.jsx file

```jsx
import React from "react";
import Child from "../Child/Child";
import { useState } from "react";

export default function Parent() {
  const [allProducts, setAllProducts] = useState([
    {
      name: "Samsung",
      price: 2000,
      category: "Mobile",
      onSale: true,
    },
    {
      name: "Dell",
      price: 3000,
      category: "Laptop",
      onSale: false,
    },
    {
      name: "HP",
      price: 4000,
      category: "Laptop",
      onSale: true,
    },
    {
      name: "LG",
      price: 6000,
      category: "PC",
      onSale: false,
    },
  ]);
  // Function to delete an element beside the state
  // This function is defined in the same component where the state is
  ↪  declared
  function deleteElement() {
    console.log("Deleted");
  }
  return (
    <>
      <div className="container-fluid">
        <div className=" row g-4 bg-red-400  ">
          <h2 className="text-center">Hello in Home Page</h2>
```

```
          {allProducts.map((product) => {
            return <Child ProductInfo={product} del={deleteElement} />;
          })}
        </div>
      </div>
    </>
  );
}
```

- Child.jsx file

```
import React from "react";

// Props are always passed as a single object, which we can destructure to
↪  access individual properties easily.
export default function Child({ ProductInfo, del }) {
  const { name, price, category, onSale } = ProductInfo; // Object
  ↪  Destructuring

  return (
    <>
      <div className="col-md-4  ">
        <div className="bg-teal-400 p-4  relative ">
          <h3 className="">
            <i className="fa-brands fa-facebook text-white pe-1"></i>
            {name}
          </h3>
          <h4>Product Price is : {price}</h4>
          <h5>Product Category is : {category}</h5>
          {onSale ? (
            <div className="bg-blue-700 absolute top-0 right-2
            ↪  p-1">Sale</div>
          ) : undefined} {/* if onSale is true, then display the Sale tag
↪  */}
          <button onClick={del} className="bg-amber-500 rounded-lg p-2
          ↪  w-full ">
            Delete
          </button>
        </div>
      </div>
    </>
  );
}
```

---

5 How To Know Which Object We need to Update, Why Don't We Use ID as an Index?

- Parent.jsx file

5 How To Know Which Object We need to Update, Why Don't We Use ID as an Index?

```
import React from "react";
import Child from "../Child/Child";
import { useState } from "react";

export default function Parent() {
  const [allProducts, setAllProducts] = useState([
    {
      name: "Samsung",
      price: 2000,
      category: "Mobile",
      onSale: true,

      id: 1,
    },
    {
      name: "Dell",
      price: 3000,
      category: "Laptop",
      onSale: false,
      id: 2,
    },
    {
      name: "HP",
      price: 4000,
      category: "Laptop",
      onSale: true,

      id: 3,
    },
    {
      name: "LG",
      price: 6000,
      category: "PC",
      onSale: false,
      id: 4,
    },
  ]);
  function deleteElement(id) {
    console.log("Deleted", id);
    allProducts.splice(id, 1);
    console.log(allProducts);
  }
  return (
    <>
      <div className="container-fluid">
        <div className=" row g-4 bg-red-400  ">
          <h2 className="text-center">Hello in Home Page</h2>

          {allProducts.map((product) => {
            return <Child ProductInfo={product} del={deleteElement} />;
```

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

- Child.jsx file

```
import React from "react";

export default function Child({ ProductInfo, del }) {
  // Props are always passed as a single object, which we can destructure
  ↪  to access individual properties easily.
  const { name, price, category, onSale, id } = ProductInfo; // Object
  ↪  Destructuring

  return (
    <>
      <div className="col-md-4  ">
        <div className="bg-teal-400 p-4  relative ">
          <h3 className="">
            <i className="fa-brands fa-facebook text-white pe-1"></i>
            {name}
          </h3>
          <h4>Product Price is : {price}</h4>
          <h5>Product Category is : {category}</h5>
          <h5>Product ID is : {id}</h5>
          {onSale ? (
            <div className="bg-blue-700 absolute top-0 right-2
            ↪  p-1">Sale</div>
          ) : undefined}{" "}
          <button
            onClick={() => {
              del(id);
            }}
            className="bg-amber-500 rounded p-2 w-full "
          >
            Delete
          </button>
        </div>
      </div>
    </>
  );
}
```

- This Code have a Problem With Id which is :
  - The id is not passed to the deleteElement function, so it will not work as expected.
  - The splice method is not updating the state, so the UI will not reflect the changes.
  - The ID won't work as an index because the index is not unique, and it will change when the array is modified.

- **allProducts.splice(id, 1);**
  - This line is trying to remove an element from the allProducts array using the id as the index, but it will not work as expected because the id is not the index of the element in the array.
  - The splice method modifies the original array, but it does not update the state, so the UI will not reflect the changes.
- **Solution:**
- Use the filter method to create a new array without the element that has the specified id, and then update the state with the new array.

---

6 Completing the Delete Process With Another Info About Taking New Copy

- Use the filter method to create a new array without the element that has the specified id, and then update the state with the new array.

```
function deleteElement(id) {
  console.log("Deleted", id);
  const newProducts = allProducts.filter((product) => product.id !== id);
  setAllProducts(newProducts);
}
```

- This code will create a new array without the element that has the specified id, and then update the state with the new array.
- The filter method creates a new array with all elements that pass the test implemented by the provided function. In this case, it will return all products except the one with the specified id.

- **Note:** When you want to update (change any data) in your state, it is best practice to first create a copy of your data, apply the changes to the copy, and then update the state with the new data. This ensures immutability and prevents unexpected side effects in your React components.
- **By using Deep Copy not Shallow Copy**, you ensure that the original data remains unchanged, which is crucial for maintaining the integrity of your state in React.
- This is especially important when dealing with complex data structures like arrays or objects, where direct mutations can lead to bugs and unpredictable behavior in your application.
- **First Solution:**

```
const newCopy = JSON.parse(JSON.stringify(allProducts));
```

- This code creates a deep copy of the allProducts array, ensuring that any modifications to newCopy will not affect the original allProducts array.
- **Second Solution:**

```
const newCopy = structuredClone(allProducts);
```

- This code also creates a deep copy of the allProducts array, similar to the first solution, but it uses the `structuredClone` method which is a built-in JavaScript function designed for deep cloning objects and arrays.
- This method is more efficient and handles more complex data types compared to the JSON.parse(JSON.stringify()) method.
- **Note:** The `structuredClone` method is supported in modern browsers, but if you need to support older browsers, you may want to use a polyfill or the JSON method as a fallback.
- **Note About Spread Operator:** Don't use the spread operator to create a deep copy of an array or object, as it only creates a shallow copy. This means that nested objects or arrays will still reference the original data, leading to potential bugs if you modify them.
- Example of Shallow Copy:

```
const shallowCopy = [...allProducts]; // Shallow copy using spread
↪  operator because it only copies the top-level array.
// Modifying shallowCopy will affect allProducts if it contains nested
↪  objects or arrays (Reference Type).

// if allProducts has an primitive Datatype like string or number, then it
↪  will not affect the original array.

// Example of modifying a primitive value
const allProducts = [1, 2, 3];
const deepCopy = [...allProducts]; // deep copy using spread operator
deepCopy[0] = 100; // This will not affect allProducts
```

- Parent.jsx file

```
import React from "react";
import Child from "../Child/Child";
import { useState } from "react";

export default function Parent() {
  const [allProducts, setAllProducts] = useState([
    {
      name: "Samsung",
      price: 2000,
      category: "Mobile",
      onSale: true,

      id: 1,
    },
    {
      name: "Dell",
      price: 3000,
      category: "Laptop",
      onSale: false,
      id: 2,
    },
    {
      name: "HP",
```

```
      price: 4000,
      category: "Laptop",
      onSale: true,

      id: 3,
    },
    {
      name: "LG",
      price: 6000,
      category: "PC",
      onSale: false,
      id: 4,
    },
  ]);
  function deleteElement(id) {
    console.log("Deleted", id);
    // allProducts.splice(id, 1);
    // console.log(allProducts);

    const result = allProducts.filter((product) => {
      return product.id !== id; // this means that i will return all the
       ↪  products except the one that has the same id
    });
    setAllProducts(result); // this will update the state and re-render
     ↪  the component with the new state
    // console.log(result);
  }
  return (
    <>
      <div className="container-fluid">
        <div className=" row g-4 bg-red-400  ">
          {allProducts.map((product) => {
            return <Child ProductInfo={product} del={deleteElement} />;
          })}
        </div>
      </div>
    </>
  );
}

// Explain The Part of the Delete in Parent:
- The `deleteElement` function is defined in the Parent component, which
 ↪  has access to the `allProducts` state. When the `del` function is
 ↪  called from the Child component, it passes the `id` of the product to
 ↪  be deleted.
- The `deleteElement` function filters out the product with the specified
 ↪  `id` from the `allProducts` array and updates the state with the new
 ↪  array, causing a re-render of the component without the deleted
 ↪  product.
```

- Child.jsx file

6 Completing the Delete Process With Another Info About Taking New Copy

```
import React from "react";

export default function Child({ ProductInfo, del }) {
  // Props are always passed as a single object, which we can destructure
  ↪  to access individual properties easily.
  // Destructuring the ProductInfo object to extract name, price,
  ↪  category, onSale

  const { name, price, category, onSale, id } = ProductInfo; // Object
  ↪  Destructuring

  return (
    <>
      <div className="col-md-4  ">
        <div className="bg-teal-400 p-4  relative ">
          <h3 className="">
            <i className="fa-brands fa-facebook text-white pe-1"></i>
            {name}
          </h3>
          <h4>Product Price is : {price}</h4>
          <h5>Product Category is : {category}</h5>
          <h5>Product ID is : {id}</h5>
          {onSale ? (
            <div className="bg-blue-700 absolute top-0 right-2
            ↪  p-1">Sale</div>
          ) : undefined}{" "}
          {/* if onSale is true, then display the Sale tag */}
          <button
            onClick={() => {
              del(id);
            }}
            className="bg-amber-500 rounded p-2 w-full "
          >
            Delete
          </button>
        </div>
      </div>
    </>
  );
}

// Explain The Part of the Delete on Child:
- When the Delete button is clicked, it calls the `del` function passed
↪  from the Parent component, passing the `id` of the product to be
↪  deleted. This triggers the `deleteElement` function in the Parent
↪  component, which filters out the product with the specified `id` and
↪  updates the state, causing a re-render of the component without the
↪  deleted product.
```

7 Update Element

- **Note:** When you want to update (change any data) in your state, it is best practice to first create a copy of your data, apply the changes to the copy, and then update the state with the new data. This ensures immutability and prevents unexpected side effects in your React components.
- **By using Deep Copy not Shallow Copy**, you ensure that the original data remains unchanged, which is crucial for maintaining the integrity of your state in React.
- This is especially important when dealing with complex data structures like arrays or objects, where direct mutations can lead to bugs and inconsistencies in your application state.
- **First Solution:**

```
const newCopy = JSON.parse(JSON.stringify(allProducts));
```

- This code creates a deep copy of the allProducts array, ensuring that any modifications to newCopy will not affect the original allProducts array.
- **Second Solution:**

```
const newCopy = structuredClone(allProducts);
```

- This code also creates a deep copy of the allProducts array, similar to the first solution, but it uses the `structuredClone` method which is a built-in JavaScript function designed for deep cloning objects and arrays.
- This method is more efficient and handles more complex data types compared to the JSON.parse(JSON.stringify()) method.
- Update Function in Parent File

```
function updateElement(product) {
  console.log("Updated", product);
  const productIndex = allProducts.indexOf(product);
  console.log("Product Index", productIndex);

  const newProductsCopy = structuredClone(allProducts); // this will create
  ↪  a deep copy of the allProducts array
  newProductsCopy[productIndex].count += 1; // this will update the count
↪  of the product that has been clicked
  setAllProducts(newProductsCopy); // this will update the state and
  ↪  re-render the component
}
```

- Parent.jsx File

```
import React from "react";
import Child from "../Child/Child";
import { useState } from "react";

export default function Parent() {
  const [allProducts, setAllProducts] = useState([
    {
      name: "Samsung",
```

```
    price: 2000,
    category: "Mobile",
    onSale: true,

    id: 1,
    count: 7,
  },
  {
    name: "Dell",
    price: 3000,
    category: "Laptop",
    onSale: false,
    id: 2,
    count: 0,
  },
  {
    name: "HP",
    price: 4000,
    category: "Laptop",
    onSale: true,

    id: 3,
    count: 17,
  },
  {
    name: "LG",
    price: 6000,
    category: "PC",
    onSale: false,
    id: 4,
    count: 10,
  },
]);
function deleteElement(id) {
  console.log("Deleted", id);
  // allProducts.splice(id, 1);
  // console.log(allProducts);

  const result = allProducts.filter((product) => {
    return product.id !== id; // this means that i will return all the
    ↪  products except the one that has the same id
  });
  setAllProducts(result); // this will update the state and re-render
  ↪  the component with the new state
  // console.log(result);
}

function updateElement(product) {
  console.log("Updated", product);
  const productIndex = allProducts.indexOf(product);
```

```
    console.log("Product Index", productIndex);

    const newProductsCopy = structuredClone(allProducts); // this will
    ↪  create a deep copy of the allProducts array
    newProductsCopy[productIndex].count += 1; // this will update the count
↪   of the product that has been clicked
    setAllProducts(newProductsCopy); // this will update the state and
    ↪  re-render the component
  }
  return (
    <>
      <div className="container-fluid">
        <div className=" row g-4 bg-red-400  ">
          <h2 className="text-center">Hello in Home Page</h2>

          {allProducts.map((product) => {
            return (
              <Child
                ProductInfo={product}
                del={deleteElement}
                upd={updateElement}
              />
            );
          })}
        </div>
      </div>
    </>
  );
}

// Explain The Part Of the Update in Parent:
- When the Update button is clicked, it calls the `updateElement` function,
↪  passing the current product as an argument. This triggers the
↪  updateElement function in the Parent component, which finds the index
↪  of the product in the allProducts array, creates a deep copy of the
↪  array, increments the count of the product, and updates the state with
↪  the new array. This causes a re-render of the component with the
↪  updated count value.
```

- Child.jsx File

```
import React from "react";

export default function Child({ ProductInfo, del, upd }) {
  const { name, price, category, onSale, id, count } = ProductInfo; //
  ↪  Object Destructuring

  return (
    <>
      <div className="col-md-4  ">
        <div className="bg-teal-400 p-4  relative ">
```

```
          <h3 className="">
            <i className="fa-brands fa-facebook text-white pe-1"></i>
            {name}
          </h3>
          <h4>Product Price is : {price}</h4>
          <h5>Product Category is : {category}</h5>
          <h5>Product ID is : {id}</h5>
          <h6>Product Count is : {count}</h6>
          {onSale ? (
            <div className="bg-blue-700 absolute top-0 right-2
            ↪  p-1">Sale</div>
          ) : undefined} {/* if onSale is true, then display the Sale tag
↪  */}
          <button
            onClick={() => {
              del(id);
            }}
            className="bg-amber-500 rounded p-2 w-full  "
          >
            Delete
          </button>
          <button
            onClick={() => {
              upd(ProductInfo);
            }}
            className="bg-red-500 w-full rounded  p-2 mt-2"
          >
            Update
          </button>
        </div>
      </div>
    </>
  );

}
// Explain The Part Of the Update on Child:
- When the Update button is clicked, it calls the `upd` function passed
↪  from the Parent component, passing the current product as an argument.
↪  This triggers the updateElement function in the Parent component, which
↪  modifies the count of the product and updates the state, causing a
↪  re-render of the component with the new count value.
```

---

8 Virtual DOM

- The Virtual DOM is a lightweight copy of the actual DOM that React uses to optimize rendering performance.
- When the state of a component changes, React updates the Virtual DOM first, then compares it to the previous version of the Virtual DOM to determine what has changed.

- After identifying the changes, React updates only the parts of the actual DOM that have changed, rather than re-rendering the entire DOM. This process is known as "reconciliation."
- This approach minimizes the number of direct manipulations to the actual DOM, which can be slow and resource-intensive, leading to better performance and a smoother user experience.
- The Virtual DOM allows React to efficiently update the UI by batching changes and applying them in a single operation, reducing the number of reflows and repaints in the browser.
- This is particularly beneficial for applications with complex UIs or frequent updates, as it helps maintain a responsive and performant user interface.
- The Warning of Key Prop in React

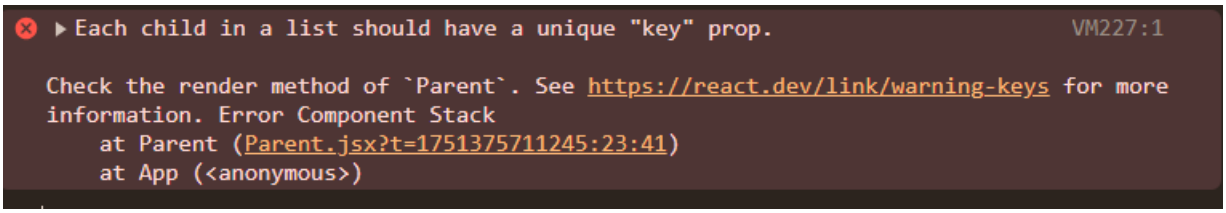

Figure 2: Key Prop Warning

- To fix the warning, you need to provide a unique key prop to each element in the array that is being rendered. The key prop helps React identify which items have changed, are added, or are removed, allowing it to optimize rendering and improve performance.

```
{
  allProducts.map((product) => {
    return (
      <Child
        key={product.id} // this is the unique key for each element in the
        ↪  array
        ProductInfo={product}
        del={deleteElement}
        upd={updateElement}
      />
    );
  });
}
```

- In this example, the `key` prop is set to `product.id`, which is assumed to be a unique identifier for each product. This ensures that React can efficiently manage the list of products and avoid unnecessary re-renders.
- Providing a unique key prop is crucial for maintaining the performance and stability of your React applications, especially when dealing with dynamic lists.

8.1 Difference Between DOM and Virtual DOM

- **DOM (Document Object Model):**

- The DOM is a tree-like structure that represents the actual HTML document in the browser.
- It is a live representation of the page, and any changes to the DOM can cause reflows and repaints, which can be slow and resource-intensive.
- Direct manipulation of the DOM can lead to performance issues, especially in applications with frequent updates or complex UIs.

- **Virtual DOM:**
  - The virtual DOM is a lightweight copy of the actual DOM that React uses to optimize rendering performance.
  - When the state of a component changes, React updates the virtual DOM first, then compares it to the previous version of the virtual DOM to determine what has changed.
  - After identifying the changes, React updates only the parts of the actual DOM that have changed, rather than re-rendering the entire DOM. This process is known as **"reconciliation."**
  - This approach minimizes the number of direct manipulations to the actual DOM, which can be slow and resource-intensive, leading to better performance and a smoother user experience.
  - The virtual DOM allows React to efficiently update the UI by batching changes and applying them in a single operation, reducing the number of reflows and repaints in the browser.
  - This is particularly beneficial for applications with complex UIs or frequent updates, as it helps maintain a responsive and performant user interface.

- **Key Differences:**
  - The DOM is a live representation of the page, while the virtual DOM is a lightweight copy used for optimization.
  - Updates to the DOM can be slow and resource-intensive, whereas updates to the virtual DOM are faster and more efficient.
  - React uses the virtual DOM to minimize direct manipulations of the actual DOM, leading to better performance and a smoother user experience.

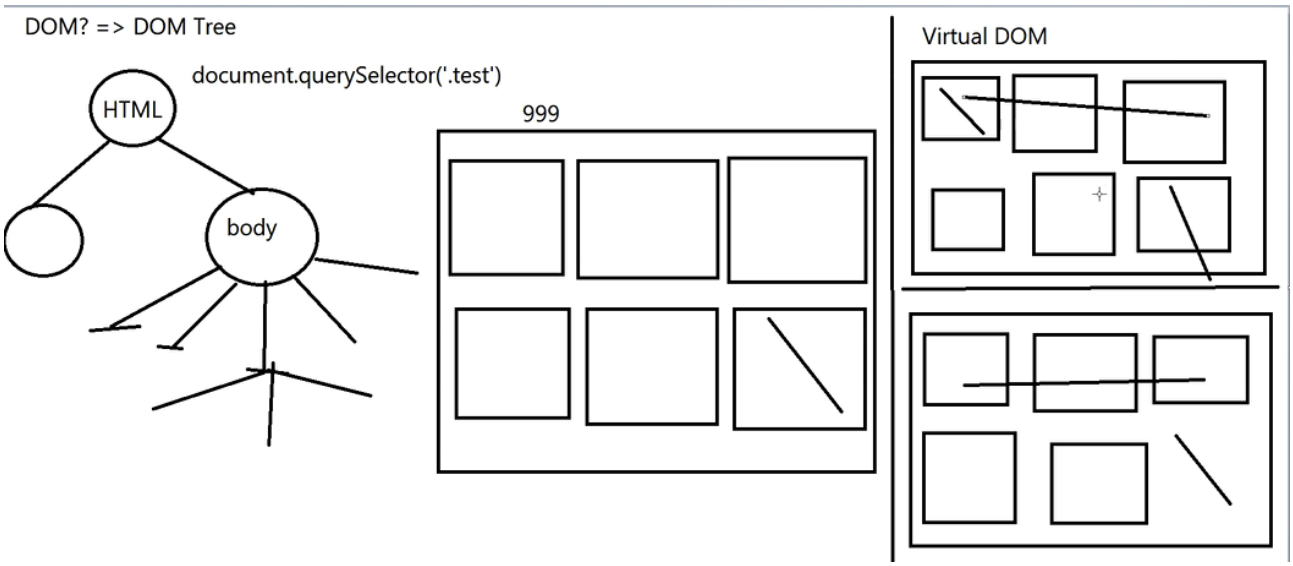

Figure 3: DOM vs Virtual DOM

Aspect	DOM	Virtual DOM
Representation	Live representation of the page	Lightweight copy used for optimization
Update Performance	Slow and resource-intensive	Faster and more efficient
Direct Manipulation	Frequent direct manipulations can cause issues	Minimizes direct manipulations
Reconciliation	Full re-rendering of the DOM	Efficient updates with minimal changes

8.2 Reconciliation Concept

- Reconciliation is the process by which React updates the actual DOM to reflect changes in the virtual DOM.
- When the state of a component changes, React creates a new virtual DOM tree and compares it with the previous virtual DOM tree.
- This comparison is done using a process called "diffing," which identifies the differences between the two virtual DOM trees.
- After identifying the differences, React updates only the parts of the actual DOM that have changed, rather than re-rendering the entire DOM.
- This process is efficient and helps maintain a responsive user interface, as it minimizes the number of direct manipulations to the actual DOM.
- Reconciliation is a key feature of React that allows it to efficiently manage updates to the UI, ensuring that only the necessary changes are made to the actual DOM, leading to better performance and a smoother user experience.
- The reconciliation process is triggered by state or prop changes in a component, and it ensures that the UI remains in sync with the underlying data model.
- React uses a heuristic algorithm to optimize the reconciliation process, which includes:
- **Key Prop Usage:** React uses the `key` prop to identify elements in a list, allowing it to track changes and optimize updates.
- **Component Type Comparison:** React compares the type of components to determine if they can be reused or if new instances need to be created.
- **Efficient Updates:** React batches updates and applies them in a single operation, reducing the number of reflows and repaints in the browser.
- This approach ensures that React can efficiently manage updates to the UI, leading to a responsive and performant application.

8.2.1 Diffing Algorithm

- The diffing algorithm is a key part of the reconciliation process in React, which allows it to efficiently update the actual DOM based on changes in the virtual DOM.
- The algorithm works by comparing the new virtual DOM tree with the previous virtual DOM tree to identify the differences (or "diffs") between them.
- Once the differences are identified, React can update only the affected parts of the actual DOM, rather than re-rendering the entire DOM tree.

- This targeted update process is what makes React's rendering efficient and performant, as it minimizes unnecessary DOM manipulations.
- The diffing algorithm uses a heuristic approach to optimize the comparison process, which includes:
  - **Element Type Comparison:** React compares the type of elements (e.g., `<div>`, `<span>`, etc.) to determine if they can be reused or if new instances need to be created.
  - **Key Prop Usage:** When rendering lists of elements, React uses the `key` prop to uniquely identify each element. This helps React track changes and optimize updates, as it can quickly determine which elements have changed, been added, or removed.
  - **Component Reconciliation:** If a component's type has changed, React will unmount the old component and mount a new one. If the type remains the same, React will update the existing component's props and state.
  - **Efficient Updates:** React batches updates and applies them in a single operation, reducing the number of reflows and repaints in the browser.
- The diffing algorithm is designed to be efficient and fast, allowing React to handle large and complex UIs without significant performance degradation.
- Example of Diffing Algorithm in Action

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

const DiffingExample = () => {
  const [items, setItems] = useState([1, 2, 3]);

  const addItem = () => {
    setItems([...items, items.length + 1]);
  };

  return (
    <div>
      <ul>
        {items.map((item) => (
          <li key={item}>Item {item}</li>
        ))}
      </ul>
      <button onClick={addItem}>Add Item</button>
    </div>
  );
};

// Explanation of the Example
- In this example, we have a simple component that renders a list of items
 ↪  and a button
- When the "Add Item" button is clicked, a new item is added to the list.
- The diffing algorithm comes into play when the state is updated. React
 ↪  will compare the new virtual DOM with the previous one and only update
 ↪  the parts of the actual DOM that have changed.
```

```
- The `key` prop is used to uniquely identify each list item, allowing
↪  React to efficiently track changes and optimize updates.
```

9 Why Index Isn't Recommended?

- Using the index of an array as a key is not recommended because the index can change if items are added, removed, or reordered. This can cause React to incorrectly associate component state with the wrong items, leading to bugs and inefficient updates. For optimal performance and correct behavior, always use a stable and unique identifier (such as an ID from your data) as the key prop.
- If the order of items changes (e.g., items are added, removed, or reordered), React may not correctly identify which items have changed.
- This can result in inefficient updates, as React may re-render components unnecessarily or fail to preserve their state.
- It's generally better to use a stable and unique identifier (e.g., an ID from your data) as the key prop to ensure proper reconciliation.
- Using a stable and unique identifier as the key prop helps React efficiently manage updates and maintain component state, leading to better performance and a smoother user experience.
- Example of Using Index as Key

```
{
  allProducts.map((product, index) => {
    return <Child key={index} ProductInfo={product} del={deleteElement}
    ↪  upd={updateElement} />;
  });
}

// Explanation of the Example
- In this example, the index of the array is used as the key prop for each
↪  `Child` component.
- While this may work in simple cases, it can lead to issues if the order
↪  of items changes or if items are added or removed.
- If the order of products changes, React may not correctly identify which
↪  items have changed, leading to inefficient updates and potential loss
↪  of component state.
```

---

10 Routing Concept

- Routing is a fundamental concept in web applications that allows users to navigate between different views or pages.
- In React, routing is typically handled using libraries like React Router, which provides a declarative way to manage routes and navigation.
- Key concepts in React routing include:
  - **Route:** A route defines a mapping between a URL path and a React component. When the user navigates to a specific path, the corresponding component is rendered.
  - **Link:** The Link component is used to create navigational links between different routes. It prevents full page reloads and allows for a smoother user experience.

  - **Switch:** The Switch component is used to group multiple routes and render only the first matching route. This is useful for defining exclusive routes.
  - **Nested Routes:** React Router supports nested routes, allowing for complex routing structures where child routes are rendered within parent components.
- **Dynamic Routing:** React Router allows for dynamic routing, where routes can be defined based on parameters or state. This enables the creation of flexible and responsive navigation structures.
- **Programmatic Navigation:** React Router provides methods for programmatic navigation, allowing you to navigate to different routes based on user actions or application state changes.
- **Route Parameters:** React Router supports route parameters, allowing you to pass dynamic values in the URL. This is useful for creating routes that depend on user input or application state.
- **404 Not Found Handling:** React Router allows you to define a fallback route for handling 404 Not Found scenarios, ensuring that users are redirected to a specific page when they navigate to an undefined route.
- **Browser History:** React Router uses the browser's history API to manage navigation, allowing users to use the back and forward buttons in their browser to navigate through the application.
- **Hash Routing vs. Browser Routing:** React Router supports both hash-based routing (using the URL hash) and browser-based routing (using the HTML5 history API). Hash routing is useful for applications that need to support older browsers, while browser routing provides a cleaner URL structure.
- **Code Splitting:** React Router supports code splitting, allowing you to load components asynchronously based on the route. This can improve performance by reducing the initial bundle size and loading only the necessary components when needed.

10.1 How To Use React Router

- To use React Router in your React application, you need to follow these steps:

1. **Install React Router:**
   - Use npm or yarn to install the React Router library in your project.

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

   or

   ```
   yarn add react-router-dom
   ```

2. **Set Up the Router:**
   - Wrap your application with the `BrowserRouter` component from React Router. This is typically done in your main entry file (e.g., `index.js` or `App.js`).

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

   function App() {
     return <BrowserRouter>{/* Your application components
      ↪  */}</BrowserRouter>;
   }
   ```

```
export default App;
```

3. **Define Routes:**

   - Use the `Route` component to define your application routes. Each route maps a URL path to a specific component that should be rendered when the user navigates to that path.

```
import { Route, Switch } from "react-router-dom";
import Home from "./Home";
import About from "./About";
import NotFound from "./NotFound";

function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route component={NotFound} />{" "}
        {/* Fallback route for 404 Not Found */}
      </Switch>
    </BrowserRouter>
  );
}

export default App;

<!-- What is RouterProvider? -->
- The `RouterProvider` component is a higher-level component that
↪  provides the routing context for your application. It is used to
↪  manage the routing state and provide access to routing-related
↪  functionality throughout your app.
- It is typically used in conjunction with the `createBrowserRouter`
↪  function to create a router instance that can be passed to the
↪  `RouterProvider`.

import { createBrowserRouter, RouterProvider } from
↪  "react-router-dom";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
  },
  {
    path: "/about",
    element: <About />,
  },
]);
```

```
function App() {
  return (
   <!-- router={router}: is a props -->

    <RouterProvider router={router}>
      {/* Your application components */}
    </RouterProvider>
  );
}
```

3.1. Difference between Switch and RouterProvider

- The `Switch` component is used to group multiple routes and render only the first matching route. This is useful for defining exclusive routes, where only one route should be rendered at a time.
- The `RouterProvider` component, on the other hand, is a higher-level component that provides the routing context for your application. It is used to manage the routing state and provide access to routing-related functionality throughout your app.
- The `Switch` component is typically used within the `RouterProvider` to define the routes for your application, while the `RouterProvider` itself is responsible for managing the routing context and providing access to the router instance.
- Router Provider Example

```
import { createBrowserRouter, RouterProvider } from "react-router-dom";

const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
  },
  {
    path: "/about",
    element: <About />,
  },
]);

function App() {
  return (
    <RouterProvider router={router}>
      {/* Your application components */}
    </RouterProvider>
  );
}
```

- Switch Example

```
import { Switch, Route } from "react-router-dom";

function App() {
  return (
    <BrowserRouter>
```

```
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route component={NotFound} /> {/* Fallback route for 404 Not
        ↪  Found */}
      </Switch>
    </BrowserRouter>
  );
}
export default App;
```

4. **Create Links:**
   - Use the `Link` component to create navigational links between different routes. This allows users to navigate through your application without triggering a full page reload.

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

5. **Access Route Parameters:**
   - If you need to access route parameters (dynamic values in the URL), you can use the `useParams` hook provided by React Router.

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

function UserProfile() {
  const { userId } = useParams(); // Accessing the userId parameter
  ↪  from the URL
  return <div>User Profile for User ID: {userId}</div>;
}
```

6. **Programmatic Navigation:**
   - You can navigate programmatically using the `useHistory` hook or the `useNavigate` hook (in React Router v6 and later).

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

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

  const handleNavigation = () => {
    navigate("/about");
  };
```

```
  return <button onClick={handleNavigation}>Go to About</button>;
}
```

7. **Handling 404 Not Found:**
   - You can define a fallback route to handle 404 Not Found scenarios by placing a `Route` without a `path` prop at the end of your `Switch` block.

```
<Route component={NotFound} />{" "}
{/* Fallback route for 404 Not Found */}
```

8. **Nested Routes:**
   - React Router supports nested routes, allowing you to define child routes within parent components. This is useful for creating complex routing structures.

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

function ParentComponent() {
  return (
    <div>
      <h1>Parent Component</h1>
      <Route path="/parent/child" component={ChildComponent} />
    </div>
  );
}
```

---

11 Issues On Our Routing Codes

1. **Error: No routes matched location "/about"**
   - This error occurs when you try to navigate to a route that has not been defined in your routing configuration.
   - To fix this error, ensure that you have defined a route for the path you are trying to access. For example, if you are trying to navigate to "/about", make sure you have a route defined for that path in your routing configuration.

```
import { BrowserRouter, Route, Switch } from "react-router-dom";
import Home from "./Home";
import About from "./About";
import NotFound from "./NotFound";
function App() {
  return (
    <BrowserRouter>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route component={NotFound} />{" "}
        {/* Fallback route for 404 Not Found */}
      </Switch>
    </BrowserRouter>
  );
```

```
}
export default App;
```

- **We Can Handle it also Like That :**

```
import React from "react";
import Home from "./Components/Home/Home";
import About from "./Components/About/About";
import Gallery from "./Components/Gallery/Gallery";
import Parent from "./Components/Parent/Parent";
import Child from "./Components/Child/Child";
import { createBrowserRouter, RouterProvider } from "react-router-dom"; //
↪  Importing RouterProvider and createBrowserRouter from
↪  react-router-dom

import "./App.css"; // Importing the CSS file for styling (App.css)

const routes = createBrowserRouter([
  { path: "", element: <Home /> },
  { path: "home", element: <Home /> },
  { path: "/about", element: <About /> },
  { path: "/gallery", element: <Gallery /> },
  { path: "/parent", element: <Parent /> },
  { path: "/child", element: <Child /> },
  {
    path: "/*",
    element: (
      <div className="h-screen bg-purple-200 flex items-center
      ↪  justify-center ">
        <h1>404 Not Found</h1>
      </div>
    ),
  },
  // We Handle It by `*`, Which means that if the user tries to access any
  ↪  route that is not defined, it will show the 404 Not Found page.
  // This is a catch-all route that will match any path that does not
  ↪  match the above routes.
  // It is important to place this route at the end of the routes array to
  ↪  ensure that it only matches when no other routes are found.
]);

export default function App() {
  return (
    <>
      <RouterProvider router={routes} />{" "}
      {/* Using RouterProvider to provide the routing context to the
      ↪  application */}
    </>
  );
}
```

2. **The Reload Problem With Path:**

- Navigating directly to a new URL or refreshing the page can cause a full reload, which is not the intended behavior in a single-page application (SPA) like React.
- To prevent this and enable smooth client-side navigation, use a navigation bar (navbar) with React Router's `Link` or `NavLink` components instead of traditional anchor (`<a>`) tags.
- These components handle navigation internally without reloading the page, preserving the SPA experience and improving performance.
- Example:

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

function Navbar() {
  return (
    <nav>
      <Link to="/">Home</Link>
      <Link to="/about">About</Link>
      <Link to="/gallery">Gallery</Link>
    </nav>
  );
}
```

- By using a navbar with `Link` components, you ensure seamless navigation between routes without triggering a full page reload.

---

12 Handling Navbar Design & Link Component

- To create a responsive and visually appealing navbar in React, you can use the `Link` component from React Router to handle navigation between different routes without reloading the page.

```
import React from "react";
import { Link } from "react-router-dom";

function Navbar() {
  return (
    <>
      <nav className="flex justify-between w-full h-12 fixed top-0 left-0
      ↪  z-50 items-center bg-gray-800 p-2 text-white">
        <div className="leftNav ms-10 text-lg font-bold flex items-center
        ↪  ">
          <Link to="/home" className="">
            Real Project
          </Link>
        </div>
        <div className="rightNav m-10 flex items-center justify-center">
          <ul className="flex space-x-7 ">
            <li>
              <Link to="/home" className=" hover:text-gray-300">
                Home
```

```
            </Link>
          </li>
          <li>
            <Link to="/about" className=" hover:text-gray-300">
              About
            </Link>
          </li>
          <li>
            <Link to="/gallery" className=" hover:text-gray-300">
              Gallery
            </Link>
          </li>
          <li>
            <Link to="/parent" className=" hover:text-gray-300">
              Parent
            </Link>
          </li>
          <li>
            <Link to="/child" className=" hover:text-gray-300">
              Child
            </Link>
          </li>
        </ul>
      </div>
    </nav>
  </>
 );
}

export default Navbar;
```

- In this example, the `Navbar` component uses the `Link` component from React Router to create navigational links. The `to` prop specifies the target route, allowing users to navigate between different pages without reloading the entire application.

Difference between Link & a href & to:

Link Component	a href	to Prop
Used for client-side navigation in React Router	Used for traditional navigation	Used within Link component to specify target route
Prevents full page reload	Causes full page reload	Specifies the path to navigate to
Maintains SPA behavior	Breaks SPA behavior	Defines the route within the application

- The `Link` component is specifically designed for client-side navigation in React applications, while the `a href` attribute is used for traditional navigation. The `to` prop is used within the `Link` component to specify the target route.

Difference between `<Link to="/home">` and `<Link to="home">`

- `<Link to="/home">` uses an absolute path. This means it always navigates to `/home` from the root of your application, regardless of the current location.
- `<Link to="home">` uses a relative path. This means it navigates to `home` relative to the current route. For example, if you are currently at `/about`, `<Link to="home">` would navigate to `/about/home`.

Summary Table:

Syntax	Path Type	Example Current Path	Resulting Navigation
`<Link to="/home">`	Absolute	`/about`	`/home`
`<Link to="home">`	Relative	`/about`	`/about/home`

- Use absolute paths (`/home`) for top-level navigation.
- Use relative paths (`home`) for nested or child routes.

---

13 Displaying Navbar in All Screen

```
import React from "react";
import Home from "./Components/Home/Home";
import About from "./Components/About/About";
import Gallery from "./Components/Gallery/Gallery";
import Parent from "./Components/Parent/Parent";
import Child from "./Components/Child/Child";
import Navbar from "./Components/Navbar/Navbar"; // Importing the Navbar
↪  component
import { createBrowserRouter, RouterProvider } from "react-router-dom"; //
↪  Importing RouterProvider and createBrowserRouter from
↪  react-router-dom

import "./App.css"; // Importing the CSS file for styling (App.css)

const routes = createBrowserRouter([
  { path: "/", element: <Home /> },
  { path: "/home", element: <Home /> },

  { path: "/about", element: <About /> },
  { path: "/gallery", element: <Gallery /> },
  { path: "/parent", element: <Parent /> },
  { path: "/child", element: <Child /> },
  {
    path: "/*",
    element: (
      <div className="h-screen bg-purple-200 flex items-center
      ↪  justify-center ">
        <h1>404 Not Found</h1>
      </div>
    ),
```

```
  },
]);

export default function App() {
  return (
    <>
      <Navbar /> {/* Displaying the Navbar component in all screens */}
      <RouterProvider router={routes} /> {/* Using RouterProvider to
       ↪  provide the routing context to the application */}
    </>
  );
}
```

- **Error Explanation:**
  Placing `<Navbar />` outside `<RouterProvider />` will cause an error:
  "useNavigate() may be used only in the context of a component."
  This happens because all React Router hooks and components (like `Link`, `useNavigate`, etc.) must be rendered within a Router context.
  If you render `Navbar` outside the RouterProvider, any `Link` inside it will throw an error and navigation will not work.

- **Best Practice Fix:**
  To ensure the Navbar and its `Link` components work correctly, include the Navbar as part of your route elements, typically by using a layout route.
  Here is the recommended approach using a layout component:

```
// Layout.jsx
import React from "react";
import Navbar from "./Components/Navbar/Navbar";
import { Outlet } from "react-router-dom";

export default function Layout() {
  return (
    <>
      <Navbar />
      <Outlet />
    </>
  );
}
```

```
// App.jsx
import React from "react";
import Home from "./Components/Home/Home";
import About from "./Components/About/About";
import Gallery from "./Components/Gallery/Gallery";
import Parent from "./Components/Parent/Parent";
import Child from "./Components/Child/Child";
import Layout from "./Components/Layout";
import { createBrowserRouter, RouterProvider } from "react-router-dom";

const routes = createBrowserRouter([
  {
```

```
    path: "/",
    element: <Layout />,
    children: [
      { path: "", element: <Home /> },
      { path: "home", element: <Home /> },
      { path: "about", element: <About /> },
      { path: "gallery", element: <Gallery /> },
      { path: "parent", element: <Parent /> },
      { path: "child", element: <Child /> },
      {
        path: "*",
        element: (
          <div className="h-screen bg-purple-200 flex items-center
          ↪  justify-center ">
            <h1>404 Not Found</h1>
          </div>
        ),
      },
    ],
  },
]);

export default function App() {
  return <RouterProvider router={routes} />;
}
```

- **Summary:**
  Always render components that use React Router features (like `Link`) inside the RouterProvider context, typically by using a layout route with `<Outlet />` for nested routing.

---

14 Creating React Layout Requires Nested Routes Handling

- **App.jsx File:**

```
import React from "react";
import Home from "./Components/Home/Home";
import About from "./Components/About/About";
import Gallery from "./Components/Gallery/Gallery";
import Parent from "./Components/Parent/Parent";
import Child from "./Components/Child/Child";
import Navbar from "./Components/Navbar/Navbar"; // Importing the Navbar
↪  component
import { createBrowserRouter, RouterProvider } from "react-router-dom"; //
↪  Importing RouterProvider and createBrowserRouter from
↪  react-router-dom

import "./App.css"; // Importing the CSS file for styling (App.css)
import Projects from "./Components/Projects/Projects";
```

```
const routes = createBrowserRouter([
  { path: "/", element: <Home /> },
  { path: "/home", element: <Home /> },

  { path: "/about", element: <About /> },
  { path: "/gallery", element: <Gallery /> },
  { path: "/parent", element: <Parent /> },
  { path: "/child", element: <Child /> },
  {
    path: "/projects",
    element: <Projects />,
    children: [
      { path: "react", element: <h1>React Projects</h1> },
      { path: "Angular", element: <h1>Angular Projects</h1> },
      { path: "Flutter", element: <h1>Flutter Projects</h1> },
      { path: "Desktop", element: <h1>Desktop Projects</h1> },
    ],
  },
  {
    path: "/*",
    element: (
      <div className="h-screen bg-purple-200 flex items-center
      ↪  justify-center ">
        <h1>404 Not Found</h1>
      </div>
    ),
  },
]);

export default function App() {
  return (
    <>
      <RouterProvider router={routes} />{" "}
      {/* Using RouterProvider to provide the routing context to the
      ↪  application */}
    </>
  );
}

// Explain the Part of Nested Routes:
- In this example, the `/projects` route has nested routes defined within
↪  it. When a user navigates to `/projects`, the `Projects` component will
↪  be rendered. Additionally, if the user navigates to `/projects/react`,
↪  the `<h1>React Projects</h1>` element will be displayed within the
↪  `Projects` component.
- This allows you to create a structured routing system where specific
↪  components can be rendered based on the URL path, enabling a more
↪  organized and modular approach to building your React application.
```

```
- children prop is used to define nested routes within the parent route.
↪  Each child route can have its own path and element, allowing for a
↪  hierarchical structure in your routing configuration.
```

- To render nested routes within a parent component, use the `Outlet` component from `react-router-dom`. Place `<Outlet />` in your parent component where you want the child route elements to appear. This enables the nested route content to be displayed dynamically based on the current path.

Projects.jsx File:

```
import React from "react";
import { Link, Outlet } from "react-router-dom";

function Projects() {
  return (
    <>
      <h1 className="text-center">Projects</h1>
      <div className="flex justify-between items-center">
        <div className="w-1/4 bg-teal-200">
          <Link className="block" to="react">
            React{" "}
          </Link>
          <Link className="block" to="angular">
            Angular{" "}
          </Link>
          <Link className="block" to="flutter">
            Flutter{" "}
          </Link>
          <Link className="block" to="desktop">
            Desktop{" "}
          </Link>
        </div>
        <div className="w-3/4 bg-purple-200">
          <Outlet />
          {/* The Outlet component will render the child routes defined in
          ↪  the router configuration */}
        </div>
      </div>
    </>
  );
}

export default Projects;
// The Links Should be Like :
- http://localhost:5173/projects/desktop
- http://localhost:5173/projects/react
- http://localhost:5173/projects/angular
- http://localhost:5173/projects/flutter
```

- The `<Outlet />` acts as a placeholder for the matched child route components.
- **Nested routes are ideal when you have a persistent layout (such as a fixed**

navbar or sidebar) and want only part of the page content to change based on the current route.

- In this scenario, the main layout remains visible, while the content area updates according to the nested route.
- This approach keeps your UI consistent and makes it easy to manage shared layout components across different pages.

---

15 Right Way To Handle The Navbar & Your LAYOUT First & Render Other Pages Inside It

15.1 Creating a Layout Component

Layout should always be the root component for your application's UI structure. It acts as the persistent wrapper that contains elements you want visible on every page (such as the Navbar, Footer, or Sidebar). All other pages and routes are rendered inside this Layout using the `<Outlet />` component. This ensures a consistent look and feel across your app, with only the main content area changing as users navigate between routes.

Step 1: Define Your Layout Structure

- Decide what should be persistent across all pages (e.g., Navbar, Footer, Sidebar).
- The Layout component will wrap these shared UI elements and provide a place for nested page content.

Example: Basic Layout Component

```jsx
// Layout.jsx
import React from "react";
import Navbar from "./Navbar"; // Import your Navbar component
import Footer from "./Footer"; // (Optional) Import your Footer component
import { Outlet } from "react-router-dom"; // For rendering nested routes

export default function Layout() {
  return (
    <>
      <Navbar />
      <main>
        <Outlet /> {/* Nested route content will be rendered here */}
      </main>
      {/* <Footer /> Uncomment if you have a footer */}
    </>
  );
}
```

Key Points:

- Place `<Navbar />` and `<Footer />` (if any) inside the Layout so they appear on every page.
- Use `<Outlet />` from `react-router-dom` as a placeholder for child route components.

- Style your layout as needed (e.g., add containers, padding, or background).
- Layout.jsx File:

```jsx
import React from "react";
import Navbar from "../Navbar/Navbar";
import Home from "../Home/Home";
import { Outlet } from "react-router-dom";

function Layout() {
  return (
    <>
      <Navbar />

      <div className="flex">
        <div className="w-1/4">
          <h1 className="m-20">Side BBar</h1>
          <h1 className="m-20">Side BBar</h1>
        </div>
        <div className="w-3/4">
          <Outlet />
        </div>
      </div>

      <div className="footer">
        <h1>I'm Footer</h1>
        <h1>I'm Footer</h1>
        <h1>I'm Footer</h1>
        <h1>I'm Footer</h1>
      </div>
    </>
  );
}

export default Layout;
```

- App.jsx File:

```jsx
import React from "react";
import Home from "./Components/Home/Home";
import About from "./Components/About/About";
import Gallery from "./Components/Gallery/Gallery";
import Parent from "./Components/Parent/Parent";
import Child from "./Components/Child/Child";
import Navbar from "./Components/Navbar/Navbar"; // Importing the Navbar
↪  component
import { createBrowserRouter, RouterProvider } from "react-router-dom"; //
↪  Importing RouterProvider and createBrowserRouter from
↪  react-router-dom

import "./App.css"; // Importing the CSS file for styling (App.css)
import Projects from "./Components/Projects/Projects";
```

```
import Layout from "./Components/Layout/Layout";

const Routes = createBrowserRouter([
  {
    path: "",
    element: <Layout />,
    children: [
      { index: true, element: <Home /> }, // Default route that renders
↪   Home component when the path is empty
      // Uncomment the line below if you want to set Home as the default
       ↪  route
      // This will render Home when the path is exactly "/"
      // If you want to set Home as the default route, you can use the
       ↪  index property
      // { index: true, element: <Home /> },

      // { path: "", element: <Home /> },
      { path: "home", element: <Home /> },

      { path: "about", element: <About /> },
      { path: "gallery", element: <Gallery /> },
      { path: "parent", element: <Parent /> },
      { path: "child", element: <Child /> },
      {
        path: "projects",
        element: <Projects />,
        children: [
          { path: "react", element: <h1>React Projects</h1> },
          { path: "Angular", element: <h1>Angular Projects</h1> },
          { path: "Flutter", element: <h1>Flutter Projects</h1> },
          { path: "Desktop", element: <h1>Desktop Projects</h1> },
        ],
      },
      {
        path: "/*",
        element: (
          <div className="h-screen bg-purple-200 flex items-center
           ↪  justify-center ">
            <h1>404 Not Found</h1>
          </div>
        ),
      },
    ],
  }, // Root route that renders the Layout component
]);

// const routes = createBrowserRouter([
//   { path: "/", element: <Home /> },
//   { path: "/home", element: <Home /> },
```

```
//   { path: "/about", element: <About /> },
//   { path: "/gallery", element: <Gallery /> },
//   { path: "/parent", element: <Parent /> },
//   { path: "/child", element: <Child /> },
//   {
//     path: "/projects",
//     element: <Projects />,
//     children: [
//       { path: "react", element: <h1>React Projects</h1> },
//       { path: "Angular", element: <h1>Angular Projects</h1> },
//       { path: "Flutter", element: <h1>Flutter Projects</h1> },
//       { path: "Desktop", element: <h1>Desktop Projects</h1> },
//     ],
//   },
//   {
//     path: "/*",
//     element: (
//       <div className="h-screen bg-purple-200 flex items-center
↪  justify-center ">
//         <h1>404 Not Found</h1>
//       </div>
//     ),
//   },
// ]);

export default function App() {
  return (
    <>
      <RouterProvider router={Routes} />{" "}
      {/* Using RouterProvider to provide the routing context to the
       ↪  application */}
    </>
  );
}
```

Note about `index: true`:

- The `index: true` property in the route configuration indicates that this route should be rendered when the parent route's path is matched exactly (e.g., when navigating to `/`).
- This is useful for setting a default route that renders a specific component (like `Home`) when the user visits the root path of your application.
- If you want to set `Home` as the default route, you can use `index: true` in the route configuration. This way, when users navigate to the root path (`/`), they will see the `Home` component without needing to specify the path explicitly.

Summary:
The Layout component acts as the main wrapper for your app, ensuring consistent UI elements (like navigation) are always visible, while the main content changes based on the current route.