

Day 6

Table of Contents:

- React Introduction
- Render HTML
- React Components
- JSX
- Class
- Props
- Events
- Conditionals
- Lists
- Forms
- Router

What is React?

React is a popular javascript library developed by facebook for building interfaces especially for building single page application. It allows developers to create reusable ui components that efficiently updates and renders as data changes.

How does React Work?

```

[1] Start / User interacts with the page
    |
    v
[2] React Component receives new data (state/props change)
    |
    v
[3] React renders a new Virtual DOM (in memory)
    |
    v
[4] React compares Virtual DOM with the previous Virtual DOM
    (This process is called "Diffing")
    |
    v
[5] React identifies the minimal set of changes (differences)
    |
    v
[6] React updates only those specific parts in the Real DOM
    |
    v
[7] Browser shows updated UI
    |
    v
[8] End / Wait for next interaction

```

React Render HTML:

React renders HTML to the web page by using a function called **createRoot()** and its method **render()**.

i) The createRoot Function:

The createRoot() function takes one argument, an HTML element.

The purpose of the function is to define the HTML element where a React component should be displayed.

```

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);

```

ii) The render Method:

The render() method is a built-in lifecycle method used only in class components in React.

It is responsible for returning the JSX (JavaScript XML) that defines the UI to be displayed on the screen.

But render where?

- There is another folder in the root directory of your React project, named "public". In this folder, there is an index.html file.
- You'll notice a single <div> in the body of this file. This is where our React application will be rendered.

index.html:

```
<body>
  <div id="root"></div>
</body>
```

index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const container = document.getElementById('root');
const root = ReactDOM.createRoot(container);
root.render(<p>Hello</p>);
```



iii) The HTML Code:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myelement = (
  <table>
    <tr>
      <th>Name</th>
    </tr>
    <tr>
      <td>John</td>
    </tr>
    <tr>
      <td>Elsa</td>
    </tr>
  </table>
);

const container = document.getElementById('root');
const root = ReactDOM.createRoot(container);
root.render(myelement);
```



iv) The Root Node:

The root node is the HTML element where you want to display the result.

It is like a container for content managed by React.

Note: It does NOT have to be a <div> element and it does NOT have to have the id='root':

Example:

The root node can be called whatever you like:

index.html:

```
<body>

  <header id="sandy"></header>

</body>
```

index.js:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const container = document.getElementById('sandy');
const root = ReactDOM.createRoot(container);
root.render(<p>Hallo</p>);
```



What is JSX?

- JSX stands for JavaScript XML.
- JSX allows us to write HTML in React.
- JSX makes it easier to write and add HTML in React.

Coding JSX:

- JSX allows us to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- JSX converts HTML tags into react elements.

Example 1: with JSX

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1>I Love JSX!</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Example 2: Without JSX

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = React.createElement('h1', {}, 'I do not use JSX!');

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Expressions in JSX:

- An expression is any piece of code that produces a value. With JSX you can write expressions inside curly braces { }.
- The expression can be a React variable, or property, or any other valid JavaScript expression.

JSX will execute the expression and return the result:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1>React is {5 + 5} times better with JSX</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Inserting a Large Block of HTML:

To write HTML on multiple lines, put the HTML inside parentheses:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
  <ul>
    <li>Apples</li>
    <li>Bananas</li>
    <li>Cherries</li>
  </ul>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



One Top Level Element:

- The HTML code must be wrapped in ONE top level element.
- So if you like to write two paragraphs, you must put them inside a parent element, like a div element.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
  <div>
    <h1>I am a Header.</h1>
    <h1>I am a Header too.</h1>
  </div>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



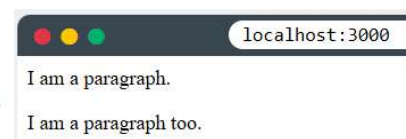
- Alternatively, you can use a "fragment" to wrap multiple lines. This will prevent unnecessarily adding extra nodes to the DOM.
- A fragment looks like an empty HTML tag: `<></>`.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = (
  <>
    <p>I am a paragraph.</p>
    <p>I am a paragraph too.</p>
  </>
);

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Elements Must be Closed:

JSX follows XML rules, and therefore HTML elements must be properly closed.

Note: Close empty elements with `</>`

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <input type="text" />;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Attribute `class` = `className`:

- The class attribute is a much used attribute in HTML, but since JSX is rendered as JavaScript, and the class keyword is a reserved word in JavaScript, you are not allowed

to use it in JSX.

- JSX solved this by using `className` instead. When JSX is rendered, it translates `className` attributes into `class` attributes.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const myElement = <h1 className="myclass">Hello World</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Conditions - if statements:

- React supports if statements, but not inside JSX.
- To be able to use conditional statements in JSX, you should put the if statements outside of the JSX, or you could use a ternary expression instead.

Option 1: Write if statements outside of the JSX code:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const x = 5;
let text = "Goodbye";
if (x < 10) {
  text = "Hello";
}

const myElement = <h1>{text}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



Option 2: Use ternary expressions instead:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

const x = 5;

const myElement = <h1>{(x) < 10 ? "Hello" : "Goodbye"}</h1>;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```



React Components:

- Components are independent and reusable bits of code. They serve the same purpose

as JavaScript functions, but work in isolation and return HTML.

- Components come in two types, Class components and Function components, in this tutorial we will concentrate on Function components.

Note: When creating a React component, the component's name MUST start with an upper case letter.

i) Class Component:

- A class component must include the extends React.Component statement. This statement creates an inheritance to React.Component, and gives your component access to React.Component's functions.
- The component also requires a render() method, this method returns HTML.

Example: Create a Class component called Car

```
class Car extends React.Component {  
  render() {  
    return <h2>Hi, I am a Car!</h2>;  
  }  
}
```

ii) Function Component:

A Function component also returns HTML, and behaves much the same way as a Class component, but Function components can be written using much less code, are easier to understand.

Example: Create a Function component called Car

```
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}
```

Rendering a Component:

- Now your React application has a component called Car, which returns an <h2> element.
- To use this component in your application, use similar syntax as normal HTML: <Car />

Example: Display the Car component in the "root" element:


```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car() {
  return <h2>Hi, I am a Car!</h2>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```



Props:

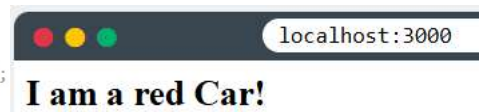
- Components can be passed as props, which stands for properties.
- Props are like function arguments, and you send them into the component as attributes.

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a {props.color} Car!</h2>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car color="red"/>);
```



Components in Components:

We can refer to components inside other components.

Example: Use the Car component inside the Garage component:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car() {
  return <h2>I am a Car!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my Garage?</h1>
      <Car />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```



Components in Files:

- React is all about re-using code, and it is recommended to split your components into separate files.
- To do that, create a new file with a .js file extension and put the code inside it.

Note: that the filename must start with an uppercase character.

Example: This is the new file, we named it "Car.js".

```
function Car() {  
  return <h2>Hi, I am a Car!</h2>;  
}
```

```
export default Car;
```

To be able to use the Car component, you have to import the file in your application.

Example: Now we import the "Car.js" file in the application, and we can use the Car component as if it was created here.

```
import React from 'react';  
import ReactDOM from 'react-dom/client';  
import Car from './Car.js';
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));  
root.render(<Car />);
```



React Props:

Props (short for properties) are a way to pass data from one component to another, like arguments to a function.

- React Props are like function arguments in JavaScript and attributes in HTML.
- To send props into a component, use the same syntax as HTML attributes:

Syntax:

```
<ComponentName propName="value" />
```

Then inside ComponentName, you access it as:

```
props.propName
```

Example: Parent Component (App.js)

```
import React from 'react';
import Greeting from './Greeting';

function App() {
  return (
    <div>
      <Greeting name="Sam" />
      <Greeting name="Chris" />
      <Greeting name="Alex" />
    </div>
  );
}

export default App;
```

Example: Child Component (Greeting.js)

```
import React from 'react';

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

export default Greeting;
```

Output:

```
Hello, Sam!
Hello, Chris!
Hello, Alex!
```

Example Structure (All in One File):

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand="Ford" />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Output:

Who lives in my Garage?

I am a Mustang!

React Events:

React events are just like HTML DOM events (like click, change, submit, etc.), but written in camelCase and handled using functions (not strings).

Adding Events:

- React events are written in camelCase syntax: onClick instead of onclick.
- React event handlers are written inside curly braces: onClick={shoot} instead of onclick="shoot()".

Example:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Football() {
  const shoot = () => {
    alert("Great Shot!");
  }

  return (
    <button onClick={shoot}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```



Passing Arguments:

To pass an argument to an event handler, use an arrow function.

Example: Send "Goal!" as a parameter to the shoot function, using arrow function:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Football() {
  const shoot = (a) => {
    alert(a);
  }

  return (
    <button onClick={() => shoot("Goal!")}>Take the shot!</button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Football />);
```



Event with Parameters:

What if you want to send a value when the button is clicked?

```
function MyButton() {
  function handleClick(name) {
    alert("Hello " + name);
  }

  return (
    <button onClick={() => handleClick("Sam")}>Click Me</button>
  );
}
```

Output: Clicking the button shows:

Hello Sam

Common React Events:

Event	Description
<code>onClick</code>	Button or element click
<code>onChange</code>	Input or form value change
<code>onSubmit</code>	Form submission
<code>onMouseOver</code>	Mouse hovers
<code>onKeyDown</code>	Key is pressed down

What is the Event Object in React?

When an event (like a click) happens in React, React automatically passes an event object to the event handler function.

This object contains information about the event, like:

- What element was clicked
- Mouse position
- Keyboard key pressed
- Form values, and much more!

Basic Syntax:

```
function handleClick(event) {  
  console.log(event); // event is the event object  
}
```

Example: onClick Event with Event Object

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MyButton() {
  function handleClick(e) {
    console.log(e);           // Full event object
    console.log(e.type);     // Event type: "click"
    console.log(e.target);   // Element that was clicked
    alert("Button clicked!");
  }

  return (
    <button onClick={handleClick}>
      Click Me
    </button>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyButton />);
```

Output:

- A button appears: "Click Me"
- When clicked:
 - Console shows the full event object
 - Also prints:

```
arduino

click
[object HTMLButtonElement]
```

- Alert pops up: "Button clicked!"

Useful Properties in Event Object:

Property	Description
<code>e.type</code>	Type of event (e.g., "click", "change")
<code>e.target</code>	The element that triggered the event
<code>e.target.value</code>	Value of input (for forms)
<code>e.preventDefault()</code>	Prevent default browser action
<code>e.stopPropagation()</code>	Stops bubbling to parent elements

React Conditional Rendering:

Conditional Rendering means displaying different content (components or elements) in the UI based on certain conditions, like if/else, user login status, data availability, or user actions.

i) Using if Statement:

We can use the if JavaScript operator to decide which component to render.

```

import React from 'react';
import ReactDOM from 'react-dom/client';

function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  if (isGoal) {
    return <MadeGoal/>;
  }
  return <MissedGoal/>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);

```

Output:

MISSED!

ii) Logical && Operator:

Another way to conditionally render a React component is by using the && operator.

Note: we use && when we want to show something only if a condition is true.

Example:


```

import React from 'react';
import ReactDOM from 'react-dom/client';

function Garage(props) {
  const cars = props.cars;
  return (
    <>
      <h1>Garage</h1>
      {cars.length > 0 &&
        <h2>
          You have {cars.length} cars in your garage.
        </h2>
      }
    </>
  );
}

const cars = ['Ford', 'BMW', 'Audi'];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage cars={cars} />);

```

Output:

Garage

You have 3 cars in your garage.

Explanation: If cars.length > 0 is equates to true, the expression after && will render.

iii) Ternary Operator:

Another way to conditionally render elements is by using a ternary operator.

Syntax:

```
condition ? true : false
```

Example:

```

import React from 'react';
import ReactDOM from 'react-dom/client';

function MissedGoal() {
  return <h1>MISSED!</h1>;
}

function MadeGoal() {
  return <h1>GOAL!</h1>;
}

function Goal(props) {
  const isGoal = props.isGoal;
  return (
    <>
      { isGoal ? <MadeGoal/> : <MissedGoal/> }
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Goal isGoal={false} />);

```

Output:

MISSED!

Explanation: Return the MadeGoal component if isGoal is true, otherwise return the MissedGoal component.

Summary:

Method	Usage Example
<code>if</code>	Best for more complex conditions before <code>return</code>
<code>? :</code>	Great for inline checks inside JSX
<code>&&</code>	Show something only if condition is true
<code>switch</code>	Rare; used for multiple options

React Lists:

- In React, a list is just a collection of items (like an array) that you want to display on the screen using JSX.

- In React, you will render lists with some type of loop. The `map()` array method is generally the preferred method.

Example: Let's render all of the cars from our garage:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = ['Ford', 'BMW', 'Audi'];
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <ul>
        {cars.map((car) => <Car brand={car} />)}
      </ul>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Output:

Who lives in my garage?

- I am a Ford
- I am a BMW
- I am a Audi

Note: When you run this code in your create-react-app, it will work but you will receive a warning that there is no "key" provided for the list items.

Keys:

- Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list.
- Keys need to be unique to each sibling. But they can be duplicated globally.

Example:

```

import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <li>I am a { props.brand }</li>;
}

function Garage() {
  const cars = [
    {id: 1, brand: 'Ford'},
    {id: 2, brand: 'BMW'},
    {id: 3, brand: 'Audi'}
  ];
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <ul>
        {cars.map((car) => <Car key={car.id} brand={car.brand} />)}
      </ul>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);

```

Who lives in my garage?

- I am a Ford
- I am a BMW
- I am a Audi

Summary:

Concept	Explanation
<code>map()</code>	Loops through array and returns JSX
<code>key</code>	Unique identifier for each element
<code>ul / li</code>	Used to show lists

React Forms:

A form is used to collect input from the user (like name, email, password, etc.).

In React, forms are slightly different from plain HTML because React uses something called state to handle input values.

i) Adding Forms in React:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function MyForm() {
  return (
    <form>
      <label>Enter your name:
        <input type="text" />
      </label>
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

Output:

Enter your name:

Note:

- This will work as normal, the form will submit and the page will refresh.
- But this is generally not what we want to happen in React.
- We want to prevent this default behavior and let React control the form.

ii) Handling Forms:

Form handling refers to managing data within a form when its values change or when the form is submitted. This process is important for collecting and manipulating data that users input.

Form Handling in HTML vs React:

HTML: Form data is usually handled by the DOM (Document Object Model), meaning the browser directly manages the form data.

React: Form data is handled by React components, where the data is stored in the component state and changes are managed via React's state management system.

Note:

- When the data is handled by the components, all the data is stored in the component state.
- You can control changes by adding event handlers in the onChange attribute.
- We can use the useState Hook to keep track of each inputs value and provide a "single

source of truth" for the entire application.

Example: Use the useState Hook to manage the input

```
import { useState } from "react";
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

  return (
    <form>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
    </form>
  )
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

Enter your name:

iii) Submitting Forms:

You can control the submit action by adding an event handler in the onSubmit attribute for the <form>:

Example: Add a submit button and an event handler in the onSubmit attribute.

```

import { useState } from "react";
import ReactDOM from 'react-dom/client';

function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  )
}

```

```

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

Enter your name:

iv) Multiple Input Fields:

- When you have multiple input fields in a form and want to handle them together, you can manage the form state more efficiently by using the **name** attribute for each field.
- This allows you to use a single state object to track the values of all input fields.
- We will initialize our state with an empty object {}.
- To access the fields in the event handler use the **event.target.name** and **event.target.value** syntax.
- To update the state, use square brackets [bracket notation] around the property name.

Example:

```

import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [inputs, setInputs] = useState({});

  const handleChange = (event) => {
    const name = event.target.name;
    const value = event.target.value;
    setInputs(values => ({...values, [name]: value}));
  }

  const handleSubmit = (event) => {
    event.preventDefault();
    console.log(inputs);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>Enter your name:
      <input
        type="text"
        name="username"
        value={inputs.username || ""}
        onChange={handleChange}
      />
    </label>
    <label>Enter your age:
    <input
      type="number"
      name="age"
      value={inputs.age || ""}
      onChange={handleChange}
    />
    </label>
    <input type="submit" />
  </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);

```

Output:

Enter your name: Enter your age:

v) Textarea:

In React, a `<textarea>` element can be controlled just like other form inputs (like `<input>`). The value of the `<textarea>` element can be controlled via React's state, making it a "controlled component."

Example:

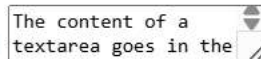
```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [textarea, setTextarea] = useState(
    "The content of a textarea goes in the value attribute"
  );

  const handleChange = (event) => {
    setTextarea(event.target.value)
  }

  return (
    <form>
      <textarea value={textarea} onChange={handleChange} />
    </form>
  )
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```

A screenshot of a web browser window. It shows a single text area containing the text "The content of a textarea goes in the value attribute". The text area has a standard scrollbar on the right side.

vi) Select:

In React, the selected value is defined with a value attribute on the select tag.

Example:

```
import { useState } from "react";
import ReactDOM from "react-dom/client";

function MyForm() {
  const [myCar, setMyCar] = useState("Volvo");

  const handleChange = (event) => {
    setMyCar(event.target.value)
  }

  return (
    <form>
      <select value={myCar} onChange={handleChange}>
        <option value="Ford">Ford</option>
        <option value="Volvo">Volvo</option>
        <option value="Fiat">Fiat</option>
      </select>
    </form>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyForm />);
```



React Router:

React Router allows you to define path (routes) that map to specific components. These components will render based on the url path. To set up routing in a React application, you typically use BrowserRouter, Routes, and Route components from react-router-dom.

Add React Router: npm i react-router-dom

Folder Structure:

To create an application with multiple page routes, let's first start with the file structure.

Within the src folder, we'll create a folder named pages with several files:

src\pages\ :

- Layout.js
- Home.js
- Blogs.js
- Contact.js
- NoPage.js

Example: Use React Router to route to pages based on URL.

index.js:

```

import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import Layout from "../pages/Layout";
import Home from "../pages/Home";
import Blogs from "../pages/Blogs";
import Contact from "../pages/Contact";
import NoPage from "../pages/NoPage";

export default function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Layout />}>
          <Route index element={<Home />} />
          <Route path="blogs" element={<Blogs />} />
          <Route path="contact" element={<Contact />} />
          <Route path="*" element={<NoPage />} />
        </Route>
      </Routes>
    </BrowserRouter>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<App />);

```

Output:

[Home](#)
[Blogs](#)
[Contact](#)

Home

Explanation:

i) BrowserRouter: This wraps your entire app and enables React router to keep track of the browser history, managing the navigation between different routes.

ii) Routes: This component is a container for all the individual routes.

iii) Route: Each route defines a path (URL) and the corresponding component to render. For example, when the user visits /about the About component will be rendered.

Note: Setting the path to * will act as a catch-all for any undefined URLs. This is great for a 404 error page.

Pages / Components:

- The Layout component has <Outlet> and <Link> elements.
- The <Outlet> renders the current route selected.
- <Link> is used to set the URL and keep track of browsing history.
- Anytime we link to an internal path, we will use <Link> instead of .
- The "layout route" is a shared component that inserts common content on all pages, such as a navigation menu.

Layout.js:

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

const Layout = () => {
  return (
    <>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/blogs">Blogs</Link>
          </li>
          <li>
            <Link to="/contact">Contact</Link>
          </li>
        </ul>
      </nav>

      <Outlet />
    </>
  )
};

export default Layout;
```

Home.js:

```
const Home = () => {
  return <h1>Home</h1>;
};

export default Home;
```

Blog.js:

```
const Blogs = () => {  
  return <h1>Blog Articles</h1>;  
};  
  
export default Blogs;
```

Contact.js:

```
const Contact = () => {  
  return <h1>Contact Me</h1>;  
};  
  
export default Contact;
```

NoPage.js:

```
const NoPage = () => {  
  return <h1>404</h1>;  
};  
  
export default NoPage;
```

NavLink and Link:

They are used to create navigation links in a React application but they have different purposes and behaviors.

i) Link:

Used to create a standard navigation links. It works similarly to an HTML anchor tag <a>, but instead of causing a full page reload, it allows React to handle navigation without refreshing the page.

ii) NavLink:

It is an enhanced version of Link that is used when you want to apply styles to the currently active link. This is especially useful for navigation menus where you want the current page's link to be visually distinct.

It automatically creates a class name called 'active', where you can style the active link.

Example:

Navbar.js

```
import React from "react";
import { NavLink } from "react-router-dom";
import "./styles.css";

const Navbar = () => {
  return (
    <nav className="navbar">
      <NavLink to="/" className="nav-link">
        Home
      </NavLink>
      <NavLink to="/about" className="nav-link">
        About
      </NavLink>
      <NavLink to="/services" className="nav-link">
        Services
      </NavLink>
      <NavLink to="/contact" className="nav-link">
        Contact
      </NavLink>
    </nav>
  );
};

export default Navbar;
```