ReactJS Basics

Target & Objective!

At the end of this PPT you will be able **to build a frontend in ReactJS** and **connect with any backend for data transfer & interaction**

Quick review of some JavaScript Basic features

- Spreading
- Destructuring objects
- Rest parameter
- Map, Filter, Reduce, etc.

Description of React Basics and working with components

- In this chapter we are going to cover:
  1. How react works
  2. JSX in react
  3. Components in ReactJS
  4. Outputting dynamic data and working with expressions
  5. Passing data via props
  6. The concept of composition of components (children props, connecting multiple components)

Short History about React (ReactJS)

- It was created by **Jordan Walke**, who was a **software engineer** at **Facebook**.
- It was initially developed and maintained by Facebook and was later **used in its products like WhatsApp & Instagram**.
- Facebook developed React in **2011** in its newsfeed section, but it was released to the public in the month of **May 2013**.

Description of React

- **It is commonly used for building web applications**
- It is particularly used for **creating dynamic** and **interactive user interfaces**.
- It allows developers to **create reusable UI components** and **manage the state** of these components efficiently
- It is a **component-based architecture**
- Key word:
  - **reusable components**

Some other key points about React

- **Component-Based**
  - React encourages a component-based architecture, where **UIs are built using reusable components.**
  - **Each component encapsulates its own logic** and **UI**, making it easier to manage and maintain large-scale applications.
- **Virtual DOM**
  - React utilizes a **virtual DOM** (Document Object Model) to efficiently update the UI.
  - Instead of directly manipulating the browser's DOM, **React creates a lightweight representation of the DOM in memory**, compares it with the previous state, and applies only the necessary updates to the actual DOM, resulting in better performance.

Some other key points about React

- **JSX Syntax**
  - React **uses JSX** (JavaScript XML), an extension of JavaScript syntax that allows developers to write HTML-like code within JavaScript.
  - JSX makes it easier to write and visualize UI components, as it closely resembles HTML markup.
- **Unidirectional Data Flow**
  - React follows a **unidirectional data flow pattern**, where **data flows down from parent components to child components via props.**
  - This helps maintain a clear and predictable data flow within the application, making it easier to debug and reason about.
- **Declarative**
  - **React promotes a declarative programming style**, where developers describe the desired UI state and React takes care of updating the DOM to match that state.

How react works

- It creates a **virtual representation of the user interface,** called the **virtual DOM**
- The **virtual DOM** is a lightweight, in-memory representation of the actual DOM that is used to compute the changes that need to be made to the real DOM
- When the **state of a component changes**, React updates the **virtual DOM** and calculates the minimal set of changes that need to be made to update the actual DOM. This process is called "**reconciliation**".
- React uses a **unidirectional data flow model**, where data flows from parent components to child components through "**props**".
- **Props** are **read-only** and are **used to pass data and behaviour from parent components to child components**. When a child component needs to communicate back to its parent, it does so through "callbacks" or "events".

React Environment Setup

- **Pre-requisite**
  - Installation of NodeJS
- **Method 1: Using create-react-app (CRA command)**
- **Method 2: Using webpack and babel**
- **Method 3: Using Vite build tool**

Installation of NodeJS Method 1: Using create-react-app (CRA command)

- **Step 1**: Navigate to the folder where you want to create the project and open it in terminal
- **Step 2**: In the terminal of the application directory type the following command
  - **npx create-react-app** (*app name here*)
- **Step 3**: Navigate to the newly created folder using the command
  - **cd** (*app name here*)
- **Step 4:** To run this application type the following command in terminal
  - **npm start**

Installation of NodeJS Method 2: Using webpack and babel

- To setup a react development environment using **webpack** and **babel** is a long process and we have to import each package and create setup files ourselves.
- We have to create the setup using '**npm init -y**' command and then import the necessary packages in the folder and then install react using the command.
  - **npm i react react-dom**
- To install the **necessary packages** in our project use the command
  - **npm i webpack webpack-cli @babel/core @babel/preset-env@babel/preset-react babel-loader html-webpack-plugin webpack-dev-server - -save-dev**

Installation of NodeJS Method 3: Using Vite build tool

- **Step 1**: Navigate to the folder where you want to create the project and open it in terminal
- **Step 2:** In the terminal of the application directory type the following command.
  - npm **create vite@latest** (*app name here*)
- **Step 3:** Select the React Framework and then variant as JavaScript from options.
- **Step 4:** Navigate to the newly created folder using the command.
  - **cd** (*app name here*)
- **Step 5:** Use the below command in terminal to install all required dependencies.
  - npm **install**
- **Step 6:** To run the application use the following command in terminal.
  - npm **run dev**

JSX in react

- JSX stands for **JavaScript XML**.
- JSX **allows us to write HTML in React**.
- JSX makes it easier to write and add HTML in React.
- The **JSX elements** you write are actually compiled down to JavaScript by a tool called **Babel**.
- **Babel** is a "**transpiler**," a made-up word that means it transforms code into valid ES5 JavaScript that all browsers can understand.
- Each JSX element becomes a function call, where its arguments are its attributes ("**props**") and its contents ("**children**").

JSX in react

```
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);
```

Components in React

- React is all about **components**.
- Components are **independent and reusable bits of code.**
- They serve the same purpose as **JavaScript functions**, but work in isolation and return HTML.
- It is advisable to start component's name with an **upper-case letter**.
- React application is made up of multiple components
- Each **component has its own logic and controls.**
- **Components are reusable** which help you to maintain the code when working on larger scale projects.

Components in React

- React component provides **below functionalities**:
  - **Initial rendering of the user interface**.
  - **Management and handling of events.**
  - **Updating the UI when the internal state is changed**.
- Types of React components
  - **Function component** – Uses plain JavaScript function.
  - **Class component** – Uses ES6 class.

Function component

```
import React from 'react';

function Todo() {
    return <h2>This a Todo app</h2>
}
```

Class component

```
import React from 'react';

class Todo extends React.Component {
    render() {
        return <h2>This a Todo app</h2>
    }
}
```

Homework-Individual work

- Key differences between **React components (functional & class components)**

Rendering React Component

```
import './App.css';
import Todo from './components/Todo';
function App() {
  return (
    <div className="App">
      {/** Rendering a Component */}
      <Todo />
    </div>
  );
}

export default App;
```

Outputting dynamic data and working with props and expressions.

```jsx
import './App.css';
import Todo from "./components/Todo";
function App() {
  return (
    <div className="App">
      {/* This is how to render a component*/}
      <Todo title="Washing the body." start="2024-03-31" end="2024-04-15" personnel={3}/>
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

function Todo(props) {
    const currentDate = new Date().toDateString();
    const taskTitle = props.title;
    const taskStartDate = new Date(props.start).toDateString();
    const taskEndDate = new Date(props.end).toDateString();
    const numberPersonnel = props.personnel;
    const isTooLate = taskStartDate < currentDate;

    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {taskTitle}</li>
                <li>Task start: {taskStartDate}</li>
                <li>Task end: {taskEndDate}</li>
                <li>Number Personnel: {numberPersonnel}</li>
            </ol>
            {isTooLate ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

React props

- **React component props** are used **to pass data from component to component**.
- The word props is basically a shorter term for **properties**.
- **Immutable Nature**: Props are immutable. That means child components cannot modify the props passed to them by their parent components. **They are read-only**.
- Props provide a way to make your React components **dynamic** and **reusable**.
- They allow you to pass data down the component tree, making it easy to build complex UIs composed of smaller, self-contained components.

Props naming rules

- Use descriptive and **meaningful names.**
- Use **camelCase.**
- Be consistent.
- **Avoid reserved words.**
- **Avoid abbreviations** and **acronyms.**
- Use **consistent naming** for related props.
- **Avoid name collisions.**

React properties supports attribute's value of different types.

- React properties supports attribute's value of different types:
  - **String**
  - **Number**
  - **Date time**
  - **Array**
  - **List**
  - **Objects**

Parent, child components and prop destructuring

```jsx
import React from 'react';
import ChildComponent from './ChildComponent';

/**  Parent Component: this defines two variables (name and age)
and passes them as props to the child component.*/
function ParentComponent() {
    const name = 'John Doe';
    const age = 25;

    return (
        <div>
            <h1>Parent Component</h1>
            {/* render the ChildComponent and pass
            the name and age props to it using JSX syntax */}
            <ChildComponent name={name} age={age} />
        </div>
    );
}

export default ParentComponent;
```

```jsx
import React from 'react';

// Child Component
function ChildComponent(props) {
    /** receive the props as an argument and
     * destructure them to access their values
     */
    const { name, age } = props;

    return (
        <div>
            <h2>Child Component</h2>
            {/* use these prop values */}
            <p>Name: {name}</p>
            <p>Age: {age}</p>
        </div>
    );
}
/** Note that props are read-only and
cannot be modified within the child component.
 to update the prop values, go in the parent component's state
 and pass updated values as props */
export default ChildComponent;
```

Inline style in React

```jsx
import React from 'react';

function Todo(props) {
    const { title, start, end, personnel, isStarted = new Date() > new Date(start) } = props;
    const headingStyle = {
        color: 'red',
        fontSize: '24px',
        fontWeight: 'bold',
        marginBottom: '10px',
    };
    const paragraphStyle = {
        color: 'blue',
        fontSize: '16px',
    };

    return (
        <div>
            <h1 style={headingStyle}>Tasks</h1>
            <ol style={{ color: "white", fontSize: 18, background: "black" }}>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            {isStarted ?
            (<p style={paragraphStyle}>This task was started.</p>) :
            (<p style={paragraphStyle}>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

React Spread Props

- In React,
  - you can use the **spread operator (...) to pass props from one component to another**.
- This technique is commonly known as "**spreading props**" or "**prop spreading**."
- It allows you **to pass all the props of one component to another without explicitly specifying each prop individually**.

React Spread Props

Without props spread

```jsx
import './App.css';
import Todo from "./components/Todo";
function App() {
  return (
    <div className="App">
      {/* This passes individual prop to the child component*/}
      <Todo title="Washing the body." start="2024-03-31" end="2024-04-15" personnel={3}/>
    </div>
  );
}

export default App;
```

With props spread

```jsx
import './App.css';
import Todo from "./components/Todo";
function App() {

  const tasks = {
    title: "Washing the body.",
    start: "2024-03-31",
    end: "2024-04-15",
    personnel: 3
  }

  return (
    <div className="App">
      {/* React Spread Props: This spreads all the props from the tasks object
      and passes them individually as props to the ChildComponent (Todo).*/}

      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

1st way to receive React Spread Props

With props spread in Parent component

```
import './App.css';
import Todo from "./components/Todo";
function App() {

  const tasks = {
    title: "Washing the body.",
    start: "2024-03-31",
    end: "2024-04-15",
    personnel: 3
  }

  return (
    <div className="App">
      {/* React Spread Props: This spreads all the props from the tasks object
      and passes them individually as props to the ChildComponent (Todo).*/}

      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

Receive props in Child component

```
import React from 'react';

function Todo({ title, start, end, personnel }) {

    const taskStartDate = new Date(start);
    const isTooLate = new Date(start) < new Date();
    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {taskStartDate.toDateString()}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            {isTooLate ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

2nd way to receive React Spread Props

With props spread in Parent component

```jsx
import './App.css';
import Todo from "./components/Todo";
function App() {

  const tasks = {
    title: "Washing the body.",
    start: "2024-03-31",
    end: "2024-04-15",
    personnel: 3
  }

  return (
    <div className="App">
      {/* React Spread Props: This spreads all the props from the tasks object
      and passes them individually as props to the ChildComponent (Todo).*/}

      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

Receive props in Child component

```jsx
import React from 'react';

function Todo(tasks) {
    const { title, start, end, personnel } = tasks;

    const taskStartDate = new Date(start);
    const isStarted = new Date(start) < new Date();
    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {taskStartDate.toDateString()}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            {isStarted ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

Composition of components
(connecting multiple components)

- This is a practice of **building complex user interfaces** by combining smaller, reusable components together.
- Composition allows you to create a **hierarchy of components**, where parent components can encapsulate and manage the **behavior** and **state** of their child components.
- It's a fundamental concept in React development and **encourages a modular approach to building UIs**.

```jsx
import React from 'react';

const Todo = ({ title, start, end, personnel }) => {
    return (
        <div>
            <Header title={`Today's task is ${title}`} />
            <Body start={start} end={end} personnel={personnel} />
            <Footer start={start} />
        </div>
    );
}

const Header = ({ title }) => <h1>{title}</h1>

const Body = ({ start, end, personnel }) => {
    return (
        <ol>
            <li>Task start: {start}</li>
            <li>Task end: {end}</li>
            <li>Number Personnel: {personnel}</li>
        </ol>
    );
}

const Footer = ({ start }) => {
    const isStarted = new Date(start) < new Date();
    return (
        <div>
            {isStarted ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

Composition of components

Rest parameter

- In React, you can use the **rest parameter** to collect all the remaining props that are not explicitly defined in your component's prop types.
- This can be useful when you want to pass down all props to a child component without explicitly naming them.

Rest parameter

```jsx
import './App.css';
import Todo from "./components/Todo";
function App() {

  const tasks = {
    title: "Washing the body.",
    start: "2024-03-31",
    end: "2024-04-15",
    personnel: 3
  }

  return (
    <div className="App">
      {/* React Spread Props: This spreads all the props from the tasks object
      and passes them individually as props to the ChildComponent (Todo).*/}

      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

Receive some props and keep others in one rest object in Child component

```jsx
import React from 'react';

const Todo = ({ title, ...rest }) => {

    return (
        <div>
            <h1>Task title: {title}</h1>
            {/* collect all the remaining props that are not explicitly defined
            in your component's prop types.  */}
            <p>Other Props: {JSON.stringify(rest)}</p>
        </div>
    );
}

export default Todo;
```

React props with Default Value

title is commented intentionally to ensure its absence

```
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    // title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

Set title=No title

```
import React from 'react';

const Todo = ({ title, ...rest }) => {

    return (
        <div>
            <h1>Task title: {title||"No title"}</h1>
            {/* collect all the remaining props that are not explicitly defined
            in your component's prop types.  */}
            <p>Other Props: {JSON.stringify(rest)}</p>
        </div>
    );
}

export default Todo;
```

How to pass Components as Props

```jsx
import './App.css';
import { Todo, Body } from "./components/Todo";

function App() {
  const tasks = [
    {
      title: 'Making bread for kids',
      start: "2024-04-04 6:00",
      end: "2024-04-05 9:00",
      personnel: 1,
      component: Body //component as a prop
    }, {
      title: 'Cleaning the house',
      start: "2024-04-04 9:00",
      end: "2024-04-05 9:30",
      personnel: 1,
      component: Body //component as a prop
    }
  ]

  return (
    <div className="App">
      <Todo tasks={tasks} />
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

const Todo = ({ tasks }) => {
    return (
        <div>
            {tasks.map((task, index) => {
                const { title, start, end, personnel, component: BodyComponent } = task;
                return (
                    <div key={index}>
                        <BodyComponent title={title} start={start} end={end} personnel={personnel} />
                    </div>
                );
            })}

        </div>
    );
}

const Body = ({ title, start, end, personnel }) => {
    return (
        <ol>
            <li> {title}</li>
            <li>Task start: {start}</li>
            <li>Task end: {end}</li>
            <li>Number Personnel: {personnel}</li>
        </ol>
    );
}
export { Todo, Body };
```

Description of React state and working with events

- In this chapter we are going to cover:
  1. Listening to events and working with event handlers
  2. Working with state
  3. Listening to user input
  4. Working with multiple states
  5. Updating state that depends on previous state
  6. Handling form submission
  7. Adding two-way binding
  8. Child-to-parent component communication(bottom-up)
  9. Lifting the state up
  10. Controlled vs uncontrolled components, stateless vs stateful components

Listening to events and working with event handlers

- **Attach an Event Handler**
  - In your component, you can attach an event handler to a particular element using the **on<EventName>** attribute.
  - For example, to listen to a button click event, you can attach the **onClick** event handler.

Listening to events and working with event handlers

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

const Todo = (tasks) => {
    const { title, start, end, personnel } = tasks;
    const isStarted = new Date(start) < new Date();

    const findTaskDuration = () => {
        // Convert milliseconds to days
        const taskDuration = (new Date(end) - new Date(start)) / (1000 * 60 * 60 * 24);
        console.log(taskDuration)
    };


    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            <button onClick={findTaskDuration}>Check task duration</button>
            {isStarted ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

Listening to events and working with event handlers with parameter

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

const Todo = (tasks) => {
    const { title, start, end, personnel } = tasks;
    const isStarted = new Date(start) < new Date();

    // Passing arguments
    const findTaskDuration = (taskSupervisor) => {
        const taskDuration = (new Date(end) - new Date(start)) / (1000 * 60 * 60 * 24);
        console.log(`${title} takes ${taskDuration} days. To be supervised by: ${taskSupervisor}`)
    };

    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            <button onClick={findTaskDuration("John")}>Check task duration</button>
            {isStarted ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

Listening to events and working with event handlers (event object)

- **Event Object**
  - React provides the event object as the first argument to event handlers.
  - You can access properties of the event object, such as **target.value** for input elements.

Listening to events and working with event handlers (event object)

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

const Todo = (tasks) => {
    const { title, start, end, personnel } = tasks;
    const isStarted = new Date(start) < new Date();

    const handleChange = (event) => {
        console.log(event.target.value)
    };

    return (

        <div>
            <h1>Tasks</h1>
            <input type="search" onChange={handleChange} />
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>Number Personnel: {personnel}</li>
            </ol>
            {isStarted ? (<p>This task was started.</p>) : (<p>This task isn't yet started.</p>)}
        </div>
    );
}

export default Todo;
```

Working with state

- In React, **state** is a JavaScript object that **represents the internal data of a component**.
- It **allows you to store and manage dynamic information** that can be updated and affect the rendering of your components.
- To initialize state in a component, you can use the **useState hook**.
- **Import it from the react package** and call it within your component function.
- The **useState hook returns an array** with two elements:
  1. the current state value and
  2. a function to update the state.

Working with state

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

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

const Todo = (tasks) => {
    const { title, start, end, personnel } = tasks;
    const [personnelNun, setPersonnel] = useState(personnel);

    const incrementPersonnel = () => {
        setPersonnel(personnelNun + 1);
    };

    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>New number of personnel: {personnelNun}</li>
            </ol>
            <button onClick={incrementPersonnel}>Increment Personnel</button>
        </div>
    );
}

export default Todo;
```

Working with state

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

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

const Todo = (tasks) => {
    const [taskTitle, setPersonnel] = useState(tasks.title);

    const updateTaskName = () => {
        setPersonnel("Cooking for kids");
    };

    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {taskTitle}</li>
                <li>Task start: {tasks.start}</li>
                <li>Task end: {tasks.end}</li>
                <li>New number of personnel: {tasks.personnelNun}</li>
            </ol>
            <button onClick={updateTaskName}>Change Task title</button>
        </div>
    );
}

export default Todo;
```

Working with state

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

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

const Todo = (tasks) => {
    const { title, start, end } = tasks;
    const [date, delayTask] = useState(new Date(start));

    const incrementStartDate  = () => {
        const newDate = new Date(date);
        newDate.setDate(newDate.getDate() + 1);
        delayTask(newDate);
    };

    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {title}</li>
                <li>Task start: {start}</li>
                <li>Task end: {end}</li>
                <li>New task start: {date.toDateString()}</li>
            </ol>
            <button onClick={incrementStartDate }>Procrastinate</button>
        </div>
    );
}

export default Todo;
```

Listening to user input

```jsx
import './App.css';
import  Todo  from "./components/Todo";

function App() {
  const tasks = {
    title: 'Making bread for kids',
    start: "2024-04-04 6:00",
    end: "2024-04-05 9:00",
    personnel: 1,
  }

  return (
    <div className="App">
      <Todo {...tasks} />
    </div>
  );
}

export default App;
```

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

const Todo = (tasks) => {
    const [taskTitle, setTitle] = useState(tasks.title);

    const setTaskName = (event) => {
        setTitle(event.target.value);
    };

    return (
        <div>
            <h1>Tasks</h1>
            <input type="text" value={taskTitle} onChange={setTaskName} />
            <ol>
                <li>Task title: {taskTitle}</li>
                <li>Task start: {tasks.start}</li>
                <li>Task end: {tasks.end}</li>
                <li>New number of personnel: {tasks.personnel}</li>
            </ol>
        </div>
    );
}

export default Todo;
```

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

const Todo = () => {
    const [taskTitle, setTitle] = useState("");
    const [taskPersonnel, setPersonnel] = useState(0);
    const [taskStart, setStart] = useState("");
    const [taskEnd, setEnd] = useState("");

    const setTaskTitle = (event) => {
        setTitle(event.target.value);
    };
    const setTaskPersonnel = (event) => {
        setPersonnel(event.target.value);
    };
    const setTaskStart = (event) => {
        setStart(event.target.value);
    };
    const setTaskEnd = (event) => {
        setEnd(event.target.value);
    };

    return (
        <div>
            <h1>Tasks</h1>
            <input type="text" value={taskTitle} onChange={setTaskTitle} />
            <input type="number" value={taskPersonnel} onChange={setTaskPersonnel} />
            <input type="date" value={taskStart} onChange={setTaskStart} />
            <input type="date" value={taskEnd} onChange={setTaskEnd} />
            <ol>
                <li>Task title: {taskTitle}</li>
                <li>Task start: {taskPersonnel}</li>
                <li>Task end: {taskStart}</li>
                <li>Task personnel: {taskEnd}</li>
            </ol>
        </div>
    );
}

export default Todo;
```

Working with multiple states

Working with multiple states

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

const Todo = () => {
    const [task, setTask] = useState({
        title: "",
        personnel: 0,
        start: "",
        end: ""
    });

    const handleChange = (event, key) => {
        setTask({ ...task, [key]: event.target.value });
    };

    return (
        <div>
            <h1>Tasks</h1>
            <input type="text" value={task.title} onChange={(e) => handleChange(e, 'title')} />
            <input type="number" value={task.personnel} onChange={(e) => handleChange(e, 'personnel')} />
            <input type="date" value={task.start} onChange={(e) => handleChange(e, 'start')} />
            <input type="date" value={task.end} onChange={(e) => handleChange(e, 'end')} />
            <ol>
                <li>Task title: {task.title}</li>
                <li>Task personnel: {task.personnel}</li>
                <li>Task start: {task.start}</li>
                <li>Task end: {task.end}</li>
            </ol>
        </div>
    );
}

export default Todo;
```

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

const Todo = () => {
    const [task, setTask] = useState({
        title: "",
        personnel: 0,
        start: "",
        end: ""
    });

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

    return (
        <div>
            <h1>Tasks</h1>
            <input type="text" name="title" value={task.title} onChange={handleChange} />
            <input type="number" name="personnel" value={task.personnel} onChange={handleChange} />
            <input type="date" name="start" value={task.start} onChange={handleChange} />
            <input type="date" name="end" value={task.end} onChange={handleChange} />
            <ol>
                <li>Task title: {task.title}</li>
                <li>Task personnel: {task.personnel}</li>
                <li>Task start: {task.start}</li>
                <li>Task end: {task.end}</li>
            </ol>
        </div>
    );
}

export default Todo;
```

Working with multiple states

Updating state that depends on previous state

You can safely update state that depends on the previous state and avoid any potential race conditions or conflicts when handling concurrent state updates.

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

const Todo = (tasks) => {
    const [personnelNun, setPersonnel] = useState(tasks.personnel);

    const incrementPersonnel = () => {
        setPersonnel((prevCount) => prevCount + 1);
    };
    const decrementPersonnel = () => {
        setPersonnel((prevCount) => prevCount - 1);
    };

    const resetPersonnel = () => {
        setPersonnel(0);
    };
    return (

        <div>
            <h1>Tasks</h1>
            <ol>
                <li>Task title: {tasks.title}</li>
                <li>Task start: {tasks.start}</li>
                <li>Task end: {tasks.end}</li>
                <li>Task personnel: {personnelNun}</li>
            </ol>
            <button onClick={incrementPersonnel}>Increment Personnel</button>
            <button onClick={decrementPersonnel}>Decrement Personnel</button>
            <button onClick={resetPersonnel}>Reset Personnel</button>
        </div>
    );
}

export default Todo;
```

Handling form submission

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

const Todo = () => {
    const [task, setTask] = useState({ title: "", personnel: 0, start: "", end: "" });

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

    const handleSubmit = async (event) => {
        event.preventDefault();
        try {
            const response = await fetch('your-backend-api-url', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(task)
            });
            if (response.ok) {
                setTask({ title: "", personnel: 0, start: "", end: "" });//Clearing data is optional
            } else {
                console.error('Failed to submit task');
            }
        } catch (error) {
            console.error('Error submitting task:', error);
        }
    };

    return (
        <div>
            <h1>Tasks details form</h1>
            <form onSubmit={handleSubmit}>
                <input type="text" name="title" value={task.title} onChange={handleChange} placeholder="Task title" />
                <input type="number" name="personnel" value={task.personnel} onChange={handleChange} placeholder="Task personnel" />
                <input type="date" name="start" value={task.start} onChange={handleChange} />
                <input type="date" name="end" value={task.end} onChange={handleChange} />
                <button type="submit">Submit</button>
            </form>
        </div>
    );
}

export default Todo;
```

Handling form submission

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

const Todo = () => {
    const [task, setTask] = useState({ title: "", personnel: 0, start: "", end: "", email: "" });
    const inputTypes = ['text', 'number', 'date', 'date']; // Add any other input types you want to support

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

    const handleSubmit = async (event) => {
        event.preventDefault();
        try {
            const response = await fetch('http://localhost:3002/api/v1/tasks/', {
                method: 'POST',
                headers: { 'Content-Type': 'application/json' },
                body: JSON.stringify(task)
            });
            if (response.ok) {
                console.log('Task submitted successfully');
                setTask({ title: "", personnel: 0, start: "", end: "" });
            } else console.error('Failed to submit task');
        } catch (error) {
            console.error('Error submitting task:', error);
        }
    };

    return (
        <div>
            <h1>Tasks</h1>
            <form onSubmit={handleSubmit}>
                {Object.entries(task).map(([key, value], index) => (
                    <input key={key} type={inputTypes[index % inputTypes.length]} name={key} value={value} onChange={handleChange} placeholder={`Task ${key}`} />
                ))}
                <button type="submit">Submit</button>
            </form>
        </div>
    );
}

export default Todo;
```

It’s time for Projects

- In your projects, continuously improve them. Make sure below features are used
  - Back-end in nodeJs+Express
  - DB in Postgres or MongoDB
  - Front-end in React
- Come and find me in the office for marking

React two-way binding

- In React, two-way binding refers to the **synchronization of data between a form input and its corresponding state value.**
- It allows changes made to the input field to be reflected in the state value and vice versa.
- **Refer to examples in previous slides:**
  - We set the value attribute of each input field to its corresponding value in the task state, enabling two-way binding. This means that changes in the input fields will update the state automatically.
  - We maintain the **handleChange** function to update the state when changes occur in the input fields, ensuring that changes in the state also reflect in the input fields.

```jsx
import React, { useState } from 'react';
import axios from 'axios';
const Todo = () => {
    const labels = ['Task Title', 'Personnel', 'Start Date', 'End Date']; // Add corresponding labels for input fields
    const inputTypes = ['text', 'number', 'datetime-local', 'datetime-local']; // Add any other input types you want to support
    const [task, setTask] = useState({ title: "", personnel: 0, start: "", end: "" });//Define form field
    const handleChange = (event) => {
        const { name, value } = event.target;
        setTask(prevState => ({ ...prevState, [name]: value }));
    };
    const handleSubmit = async (event) => {
        event.preventDefault();
        try {
            const response = await axios.post('http://localhost:3002/api/v1/tasks/', task, {
                headers: { 'Content-Type': 'application/json' }
            });
            if (response.status === 200 || response.status === 201) {
                console.log('Task submitted successfully');
                setTask({ title: "", personnel: 0, start: "", end: "" });
            } else console.error('Failed to submit task');
        } catch (error) {
            console.error('Error submitting task:', error);
        }
    };
    return (
        <div>
            <h1>Tasks</h1>
            <form onSubmit={handleSubmit}>
                {Object.entries(task).map(([key, value], index) => (
                    <div key={key}>
                        <label htmlFor={key}>{labels[index]}</label>
                        <input type={inputTypes[index % inputTypes.length]} name={key} value={value} onChange={handleChange} id={key} placeholder={`Enter ${labels[index]}`} />
                    </div>
                ))}
                <button type="submit">Submit</button>
            </form>
            <ol>
                {Object.entries(task).map(([key, value], index) => (
                    <li key={key}>{labels[index]}: {value}</li>
                ))}
            </ol>
        </div>
    );
}
```

React two-way binding

Child-to-parent component communication(bottom-up)

- **Normally,** in React, the communication between components is data **from parent to child**.
- Also, there can be a communication between child to parent components (**bottom-up)**
- it involves passing data or functions (**callback**) from child components to parent components.
- This is typically achieved by passing functions as props from the parent to the child components,
- and then invoking these functions in the child components to send data back to the parent.

Child-to-parent component communication(bottom-up)

Parent component

```jsx
import './App.css';
import Todo from "./components/Todo";
import React, { useState } from 'react';


const App = () => {
  const [dataFromChild, setDataFromChild] = useState('');

  const handleDataFromChild = (data) => {
    // Handle data received from child component
    setDataFromChild(data);
  };

  return (
    <div>
      <h1>Parent Component</h1>
      <p>Data from Child Component: {dataFromChild}</p>
      <Todo sendDataToParent={handleDataFromChild} />
    </div>
  );
};

export default App;
```

Child component

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

const Todo = ({ sendDataToParent }) => {
    const [taskTitle, setTaskTitle] = useState('');

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

    const sendDataToParentHandler = () => {
        // Send data to parent component
        sendDataToParent(taskTitle);
    };

    return (
        <div>
            <h2>Child Component</h2>
            <input type="text" value={taskTitle} onChange={handleChange} />
            <button onClick={sendDataToParentHandler}>Send Data to Parent</button>
        </div>
    );
};

export default Todo;
```

Lifting the state up

- **Lifting state up in React is a pattern** where you move the state of a component higher up in the component tree to a common ancestor, **typically to a parent component**.
- This is useful when multiple components need access to the same state or when the state affects multiple components.
- By lifting state up, you make the state accessible to multiple components and ensure that the state stays consistent across them.
- This approach helps ensure consistency and synchronization of shared data across the application
- In a nutshell, **this is often done to maintain a single source of truth for the shared data**.

Lifting the state up

Parent component

```jsx
import './App.css';
import Todo from "./components/Todo";
import React, { useState } from 'react';

const App = () => {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  const decrementCount = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h1>This is a parent Component</h1>
      <p>Count: {count}</p>
      <Todo count={count} incrementCount={incrementCount} decrementCount={decrementCount} />
    </div>
  );
};

export default App;
```

Child component

```jsx
import React from 'react';

const Todo = ({ count, incrementCount, decrementCount }) => {
    return (
        <div>
            <h2>This is a child Component</h2>
            <p>Count: {count}</p>
            <button onClick={incrementCount}>Increment</button>
            <button onClick={decrementCount}>Decrement</button>
        </div>
    );
};

export default Todo;
```

Controlled vs uncontrolled components

- **Controlled components** are components whose **values are controlled by React state**
- **Uncontrolled components** are components whose **values are managed by the DOM**, rather than by React state.

Stateless vs Stateful components

- **Stateless components**, also known as **functional components**, ~~**do not manage**~~ their own state.
- **Stateful components,** also known as **class components** or **components with hooks**, manage their own state.

Description of how to Render Lists and conditional content

- In this chapiter we are going to cover:
  1. Rendering list of data
  2. Using stateful list
  3. Understanding keys
  4. Out putting conditional content
  5. Adding Conditional Return Statements
  6. Adding Dynamic Styles

Rendering list of data

- In React, **rendering a list of data typically involves mapping over an array of items** and rendering individual components for each item.
- For each item, we create a new <li> element and provide a unique key prop using the index parameter.
- **It's important to assign a unique key to each item to help React efficiently update and reconcile the list when changes occur.**

Rendering list of data

```jsx
import './App.css';
import Todo from "./components/Todo";


function App() {
  return (
    <div className="App">
      <h1>Tasks</h1>
      <Todo />
    </div>
  );
}

export default App;
```

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

const Todo = () => {
    const [tasksList, setTasksList] = useState([]);
    const fetchTasks = async () => {
        try {
            const response = await fetch('http://localhost:3002/api/v1/tasks/');
            if (response.ok) {
                const data = await response.json();
                setTasksList(data); // Update tasksList state with fetched data
            } else console.error('Failed to fetch tasks');
        } catch (error) {
            console.error('Error fetching tasks:', error);
        }
    };
    useEffect(() => {
        fetchTasks(); // Fetch tasks when component mounts
    }, []); // Empty dependency array ensures this effect runs only once
    return (
        <div>
            <ul>
                {tasksList.map(taskItem => (
                    <li key={taskItem.id}>{taskItem.title}</li>
                ))}
            </ul>
        </div>
    );
}
export default Todo;
```

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

const Todo = () => {
  const [items, setItems] = useState([]);

  const addItem = () => setItems([...items, "New Item"]);
  /**the _ communicates that the argument is intentionally ignored.
  Normal: const newArray = array.filter((item, index) => condition);*/
  const removeItem = (index) => setItems(items.filter((_, i) => i !== index));

  return (
    <div>
      <h1>Stateful List Example</h1>
      <button onClick={addItem}>Add Item</button>
      <ul>
        {items.map((item, index) => (
          <li key={index}>
            {item}
            <button onClick={() => removeItem(index)}>Remove</button>
          </li>
        ))}
      </ul>
    </div>
  );
};

export default Todo;
```

Using
stateful list

Understanding keys

- Using keys correctly ensures that React can perform efficient updates and maintain component state accurately, especially when dealing with dynamic lists or components that frequently change.
- It's essential to choose stable and unique keys that represent the identity of each element in the list
- Role of keys in a list:
  - React uses keys to differentiate between elements and identify which items have changed, been added, or been removed.
  - Using stable keys helps React recognize elements and maintain component state correctly across updates
  - When an element is added, removed, or re-ordered within a list, React can identify which specific elements have changed by comparing their keys, rather than re-rendering the entire list.
  - React uses keys to determine whether to update an existing component, remove a component, or insert a new component when rendering lists

Out putting conditional content

- In React, **you can output conditional content** by **using JavaScript expressions inside JSX**.
- There are several ways to conditionally render content based on certain conditions.
  - Using the **ternary operator** (condition ? true : false)
  - Using **logical AND (&&) operator**
  - Using **if statements** outside of JSX
  - Using conditional rendering with **switch statement**

Out putting conditional content

Using the ternary operator (condition ? true : false)

```
import axios from 'axios';

const Login = async () => {
    try {
        const response = await axios.post('http://localhost:3001/api/v1/login/',
            { username: "kagabo@gmail.com", password: "Kagabo@123" },
            { withCredentials: true },
            { headers: { 'Content-Type': 'application/json' } }
        );
        const message = response.data.success ? "Login succeed" : "Login failled"
        console.log(message)
    } catch (error) {
        console.error('Error submitting user:', error);
    }
    return (
        <div>
            <h1>Using the ternary operator (condition ? true : false)  </h1>
        </div>
    );
}
export default Login;
```

Out putting conditional content

Using logical AND (&&) operator

```
import axios from 'axios';

const Login = async () => {
    try {
        const response = await axios.post('http://localhost:3001/api/v1/login/',
            { username: "kagabo@gmail.com", password: "Kagabo@123" },
            { withCredentials: true },
            { headers: { 'Content-Type': 'application/json' } }
        );
        const message = response.data.success && "Login succeed "
        console.log(message)
    } catch (error) {
        console.error('Error submitting user:', error);
    }
    return (
        <div>
            <h1>Using logical AND (&&) operator</h1>
        </div>
    );
}
export default Login;
```

Out putting conditional content

Using if statements outside of JSX

```
import axios from 'axios';

const Login = async () => {
    try {
        const response = await axios.post('http://localhost:3001/api/v1/login/',
            { username: "kagabo@gmail.com", password: "Kagabo@123" },
            { withCredentials: true },
            { headers: { 'Content-Type': 'application/json' } }
        );
        if (response.data.success) {
            console.log('Login Login succeed');

        } else {
            console.log('Login failed!');
        }
    } catch (error) {
        console.error('Error submitting user:', error);
    }
    return (
        <div>
            <h1>Using if statements outside of JSX</h1>
        </div>
    );
}
export default Login;
```

Out putting conditional content

Using conditional rendering with switch statement

```
const handleSubmit = async (event) => {
  event.preventDefault();
  try {
    const response = await axios.post('http://localhost:3001/api/v1/login/', user, { withCredentials: true }, {
      headers: { 'Content-Type': 'application/json' }
    });
    if (response.data.success) {
      switch (response.data.user.privilege) {
        case 1:
          navigate("/admin-dashboard");
          break;
        case 0:
          navigate("/user-dashboard");
          break;
        default:
          navigate("/guest-dashboard");
          break;
      }
    } else {
      console.log('Login failed! Please check your credentials.');
    }
  } catch (error) {
    console.error('Error submitting user:', error);
  }
};
```

Adding Conditional Return Statements

```
function Greeting({ isLoggedIn }) {
    if (isLoggedIn) {
        return <h1>Welcome back!</h1>;
    } else {
        return <h1>Please log in.</h1>;
    }
}
```

Adding Dynamic Styles

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

const Login = () => {
    const [backgroundColor, setBackgroundColor] = useState("green");
    function handleButtonClick(params) {
        setBackgroundColor("red")
    }
    return (
        <div style={{ color: backgroundColor }}>
            Hello, I am here in green. Click to turn me in red
            <button onClick={handleButtonClick}>Change Color</button>
        </div>
    );
}
export default Login;
```

Description of how to debug a react app

- Homework: **In a group of six prepare a presentation on bellow topics**
  - Understanding React Error Messages
  - Analyzing Code Flow & Warnings
  - Working with Breakpoints
  - Using the React DevTools

Description of how to work with fragments, portals and refs

- In this chapter we are going to cover:
  1. JSX Limitations & Workarounds
  2. Creating a Wrapper Component
  3. React Fragments
  4. Working with Portals
  5. Working with refs
  6. Controlled vs Uncontrolled Components

JSX Limitations & Workarounds

- **Limitation**: JSX does not allow conditional statements like if-else directly inside JSX.
  - **Workaround**: Use ternary operator or logical && operator for conditional rendering.
- **Limitation**: Inline styles in JSX can become cumbersome for complex styles.
  - **Workaround**: Extract complex styles into separate JavaScript objects or use CSS-in-JS libraries.
- **Limitation**: HTML-like syntax can sometimes be confusing for developers new to React.
  - **Workaround**: Utilize JavaScript expressions and functions to handle logic instead of relying solely on JSX.

Creating a Wrapper Component

- In React, a wrapper component is a component that wraps around another component or a group of components:
  - Creating a wrapper component in React involves defining a new component that **renders its children within a specific structure or behavior**.
  - This wrapper component can be useful for applying **consistent styles**, **layout**, or **functionality** to multiple child components.
  - By wrapping components with a wrapper component, you can keep your code organized and make it easier to maintain and extend.

Example of a Wrapper component that adds a blue border to its children

```jsx
import './App.css';
import Wrapper from "./components/wrapper-component/Wrapper";

function App() {
  return (
    <div className="App">
      <Wrapper>
        <h1>Hello, I am wrapped</h1>
        <p>Hello, I am wrapped also</p>
      </Wrapper>
    </div>
  );
}

export default App;
```

```jsx
import React from 'react';

const Wrapper = ({ children }) => {
    return (
        <div style={{ border: '1px solid blue', padding: '10px' }}>
            {children}
        </div>
    );
}

export default Wrapper;
```

React Fragments

- You can think of a Fragment as a **lightweight container** that does not generate any additional markup.
- Fragments allow you to **group a list of children without adding an extra node to the DOM**.

Two ways to use fragments

```jsx
import React from 'react';

const MyComponent = () => {
    return (
        <>
            <h1>Hello</h1>
            <p>This is a paragraph.</p>
        </>
    );
};
export default MyComponent;
```

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

const MyComponent = () => {
    return (
        <Fragment>
            <h1>Hello</h1>
            <p>This is a paragraph.</p>
        </Fragment>
    );
};
export default MyComponent;
```

Working with Portals

- **Portals in React provide a way to render children components into a DOM node that exists outside the parent component's DOM hierarchy.**
- This allows you to render components into different parts of the DOM, such as **Modal dialogs, Tooltips and popovers, Dropdown menus, without affecting the structure of the parent component**.
- Homework:
  - Create a model of you choice.
  - Reference: https://semaphoreci.com/blog/react-portals

When to use React Portal?

- A few common use cases of React Portals are:
  - Modals or Dialog boxes
  - Tooltips
  - Widgets
  - Floating menus
  - Notifications
  - Pop-up messages
  - Lightboxes
  - Loaders
  - etc.

React working with refs

- In React, refs provide a way to directly interact with DOM elements or React components.
- Refs are useful for **accessing DOM elements**, **managing focus**, **triggering imperative animations**, and **integrating third-party libraries**

React working with refs: Input Focus

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

const RefTest = () => {
    const myRef = useRef(null);

    // Accessing the ref
    const handleClick = () => {
        myRef.current.focus();
    };

    return (
        <div>
            <input ref={myRef} type="text" />
            <button onClick={handleClick}>Focus Input</button>
        </div>
    );
};

export default RefTest;
```

React working with refs: Change colour

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

const RefTest = () => {
    const myRef = useRef(null);

    // Accessing the ref
    const handleClick = () => {
        myRef.current.style.backgroundColor = 'red';
    };

    return (
        <div>
            <div ref={myRef}>Hello, world!</div>
            <button onClick={handleClick}>Change Color</button>
        </div>
    );
};

export default RefTest;
```

Refs notes to remember

- Refs should be used sparingly and are generally considered a last resort when other approaches like props and state are insufficient.
- They can lead to code that's harder to understand and maintain, especially in larger applications.
- However, when used appropriately, refs can be a powerful tool for managing DOM interactions in React.
- **The coming slide indicates how you can submit a form using refs**

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

const RefTest = () => {
    const labels = ['Task Title', 'Personnel', 'Start Date', 'End Date'];
    const inputTypes = ['text', 'number', 'datetime-local', 'datetime-local'];

    // Create ref objects for each input field
    const titleRef = useRef(null), personnelRef = useRef(null), startRef = useRef(null), endRef = useRef(null);

    const handleSubmit = async (event) => {
        event.preventDefault();
        const task = {
            title: titleRef.current.value, personnel: personnelRef.current.value, start: startRef.current.value, end: endRef.current.value
        };

        try {
            const response = await axios.post('http://localhost:3002/api/v1/tasks/', task, {
                headers: { 'Content-Type': 'application/json' }
            });
            if (response.status === 200 || response.status === 201) {
                console.log('Task submitted successfully');
                // Clear input fields after submission
                titleRef.current.value = personnelRef.current.value = startRef.current.value = endRef.current.value = "";
            } else console.error('Failed to submit task');
        } catch (error) {
            console.error('Error submitting task:', error);
        }
    };

    return (
        <div>
            <h1>Tasks</h1>
            <form onSubmit={handleSubmit}>
                {labels.map((label, index) => (
                    <div key={index}>
                        <label htmlFor={label}>{label}</label>
                        <input type={inputTypes[index % inputTypes.length]} name={label} ref={index === 0 ? titleRef : index === 1 ? personnelRef : index === 2 ? startRef : endRef} id={label} placeholder={`Enter ${label}`} />
                    </div>
                ))}
                <button type="submit">Submit</button>
            </form>
        </div>
    );
}

export default RefTest;
```

Thank you!!!!