React Typescript

1. Initialisation
   1. Why TypeScript + React?

There are several reasons why developers may choose to use React with TypeScript:

Type safety: TypeScript provides a way to add type annotations to JavaScript code, which can help catch errors at compile-time instead of runtime. This can make development and maintenance of a React application more efficient.

Intellisense: TypeScript provides Intellisense, which is a type of code completion that makes it easier to write and understand code. This can help developers write code faster and reduce the number of errors.

Better documentation: TypeScript's type annotations can serve as a form of documentation for the code, making it easier for other developers to understand how the application works.

Improved scalability: TypeScript's static typing allows for better scalability and maintainability of large codebases, it also enables better tooling and editor support making it easier to navigate and refactor large codebases.

Better third-party library support: Many popular JavaScript libraries have built-in TypeScript definitions, which makes it easier to use them in a TypeScript codebase.

Many people say :

* Typescript is just frustrating most of the time
* I have to write a lot more code than | normally do and the compiler keeps complaining. | simply don't know what to do about those errors
* It doesn't seem to be productive for me

But you have to look at typescript as an initial investment that pays off in the long run ,once I get the hang of it I’ll realize that the benefit I get outweigh the additional time we spend on it

* 1. About this course
* You'll learn with examples, how to use TypeScript with React
* Itis not a course on TypeScript itself
* You have to know the fundamentals of React
* Watch the React playlist on this channel
* TypeScript for Beginners in 50 minutes

1. Getting started
   1. Configure React with typescript

Create React app provides a typescript we can use :

* npx create-react-app react-typescript --template typescript

HELPS US TO CONFIGURED TYPESCRIPT IN OUR REACT PROJECT

In React with typescript the component are defined in a tsx file not a js file

1. Component Props
   1. Typing props

App.tsx

function App() {

  return (

    <div className="App">

      <Greet name='Vishwas'/>

    </div>

  );

}

Greet.tsx

type GreetProps={

    name: string

}

export const Greet = (props:GreetProps) => {

  return (

    <div>

      <h2>Welcome {props.name}! you have 10 unread messages</h2>{" "}

    </div>

  );

};

* 1. Basic Props

String Number Boolean Obj Array

App.tsx

function App() {

  const personName={

    first:'Bruce',

    last:'Wayne'

  }

  const nameList=[

    {

      first:'Bruce',

      last:'Wayne'

    },{

      first:'Clark',

      last:'Kent'

    },{

      first:'Princess',

      last:'Diana'

    },{

      first:'Jhown',

      last:'Snow'

    }

  ]

  return (

    <div className="App">

      <Greet name='Vishwas' messageCount={20} isLoggedIn={false}/>

      <Person name={personName}/>

      <PersonList names={nameList}/>

    </div>

  );

}

export default App;

Greet.tsx

type GreetProps={

    name: string

    messageCount:number

    isLoggedIn:boolean

}

export const Greet = (props:GreetProps) => {

  return (

    <div>

      <h2>{props.isLoggedIn ? `Welcome ${props.name}! you have ${props.messageCount} unread messages`:'Welcome Guest'

    }</h2>{" "}

    </div>

  );

};

Person.tsx

type PersonProps={

    name:{

        first:string

        last:string

    }

}

export default function Person(props:PersonProps) {

  return (

    <div>{props.name.first} {props.name.last}</div>

  )

}

PersonList.tsx

type PersonListProps = {

  names: {

    first: string;

    last: string;

  }[];

};

export default function PersonList(props: PersonListProps) {

  return (

    <div>

      {props.names.map((name) => {

        return(

            <h2 key={name.first}>{name.first} {name.last}</h2>

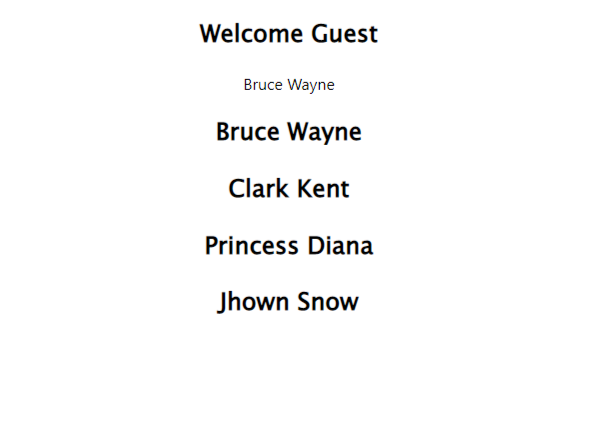
        );

      })}

    </div>

  );

}



* 1. Advanced Props

App.tsx

function App() {

  return (

    <div className="App">

      {/\* Union of string literals as our Type \*/}

      <Status status="loading" />

{/\* if we dont use 'loading' or 'success' or 'error' in the status value it will not work\*/}

{/\* Child is a string \*/}

      <Heading>PlaceHolder text</Heading>

      {/\* Child is a react component \*/}

      <Oscar>

        <Heading>PlaceHolder text</Heading>

      </Oscar>

      {/\* Optional types \*/}

      {/\* SomeTimes it might so happen that a component prop doesn't haved to be passed \*/}

      <Greet name='Vishwas' messageCount={20} isLoggedIn={true}/>

      <Greet name="Vishwas"  isLoggedIn={true} />

    </div>

  );

}

**{/\* Union of string literals as our Type \*/}**

Status.tsx

type StatusProps = {

    status: 'loading' | 'success' | 'error'

  }

  export const Status = (props: StatusProps) => {

    let message

    if (props.status === 'loading') {

      message = 'Loading...'

    } else if (props.status === 'success') {

      message = 'Data fetched successfully!'

    } else if (props.status === 'error') {

      message = 'Error fetching data'

    }

    return <h2>Status - {message}</h2>

  }

if we dont use 'loading' or 'success' or 'error' in the status value it will not work

{/\* Child is a string \*/}

Heading.tsx

type HeadingProps = {

  children: string;

};

export function Heading(props :HeadingProps) {

  return <div>{props.children}</div>;

}

{/\* Child is a react component\*/}

type OscarProps = {

    children: React.ReactNode;

  };

export const Oscar=(props: OscarProps)=>{

    return <div>{props.children}</div>

}

{/\* Child is a react component\*/}

Greet.tsx

type OscarProps = {

    children: React.ReactNode;

  };

export const Oscar=(props: OscarProps)=>{

    return <div>{props.children}</div>

}

{/\* Optional types\*/}

Greet.tsx

type GreetProps = {

  name: string;

  // Optionnal props

  messageCount?: number;

  isLoggedIn: oolean;

};

export const Greet = (props: GreetProps) => {

  // message count to 0 par default if we don’t have messageCount because messageCount is Optional props

  const { messageCount = 0 } = props;

  return (

    <div>

      <h2>

        {props.isLoggedIn

          ? `Welcome ${props.name}! you have ${messageCount} unread messages`

          : “Welcome Guest”}

      </h2>{“ “}

    </div>

  );

};

SomeTimes it might so happen that a component prop doesn’t haved to be passed

* 1. Event Props

In this paragraph let’s look at typing event props there are a lot of events that we can work with but in this paragraph, we’re going to focus on the two most frequently used event as props

The Click Event and The Change Event

App.tsx

function App() {

  return (

    <div className="App">

      <Button

        handleClick={(event, id) => console.log("Button Clicked", event, id)}

      />

{/\* Typically with the  input eelements the component would need two props the input value

and the onChange handler \*/}

      <Input value='' handleChange={(e)=>console.log(e)}/>

    </div>

  );

}

export default App;

Button.tsx

type ButtonProps = {

  handleClick: (event: React.MouseEvent<HTMLButtonElement>, id: number) => void;

};

export const Button = (props: ButtonProps) => {

  return (

    <button onClick={(event) => props.handleClick(event, 1)}>Click</button>

  );

};

Input.tsx

type InputProps = {

  value: string;

  handleChange: (event: React.ChangeEvent<HTMLInputElement>) => void;

};

export const Input = (props: InputProps) => {

  return (

    <input type="text" value={props.value} onChange={props.handleChange} />

  );

};

* 1. Style Props

App.tsx

<Container styles={{ border: "1px solid black", padding: "1 rem" }} />

Container.tsx

type ContainerProps={

    styles:React.CSSProperties

}

export const Container=(props:ContainerProps)=>{

    return <div style={props.styles}>Text content goes here </div>

}

* 1. Props Types and Tips

Destructuring props

type InputProps = {

  value: string;

  handleChange: (event: React.ChangeEvent<HTMLInputElement>) => void;

};

//Destructuring props

export const Input = ({value,handleChange}:InputProps) => {

  return (

    <input type="text" value={value} onChange={handleChange} />

  );

};

Exporting Types

When working on large components with multiple types we often want to move the types into a separate file and import them where necessary

Person.types.tsx

export type PersonProps={

    name:{

        first:string

        last:string

    }

}

Person.tsx

import { PersonProps } from "./Person.types"

export default function Person(props:PersonProps) {

  return (

    <div>{props.name.first} {props.name.last}</div>

  )

}

1. UseState, UseReducer, UseContext, UseRef
   1. UseState
      1. **For simple values**

We don’y really have to explicitly type the useState hook type inference will take care of everything for simple values Example:

import { useState } from 'react'

export const LoggedIn = () => {

  const [isLoggedIn, setIsLoggedIn] = useState(false)

  const handleLogin = () => {

    setIsLoggedIn(true)

  }

  const handleLogout = () => {

    setIsLoggedIn(false)

    // setIsLoggedIn(0) we can do that because 0 is not boolean

  }

  return (

    <div>

      <button onClick={handleLogin}>Login</button>

      <button onClick={handleLogout}>Logout</button>

      <div>User is {isLoggedIn ? 'logged in' : 'logged out'}</div>

    </div>

  )

}

* + 1. UseState for Future

In this example we will learn how to use Union types in React with useState and we will see the intelligence of using typescript (use ? automatically)

import { useState } from "react";

type AuthUser = {

  name: string;

  email: string;

};

export const User = () => {

  const [user, setUser] = useState<AuthUser | null>(null);

  //It allows us use AuthUser type and the null type becuse the user can be null

  //    or auth User

  const handleLogin = () => {

    setUser({

      name: "Vishwas",

      email: "vishwas@example.com",

    });

  };

  const handleLogout = () => {

    setUser(null);

  };

  return (

    <div>

      <button onClick={handleLogin}>Login</button>

      <button onClick={handleLogout}>Logout</button>

      {/\* intteligence helps us tu use ? Automaticaly in our object \*/}

      <div>User name is {user?.name}</div>

      <div>User name is {user?.email}</div>

    </div>

  );

};

* + 1. useState Type assertion

Type assertion is something I could potentially come across in a code base

import { useState } from "react";

type AuthUser = {

  name: string;

  email: string;

};

export const User = () => {

  const [user, setUser] = useState<AuthUser>({} as AuthUser);

  //It allows us use AuthUser type and the null type becuse the user can be null

  //    or auth User

  const handleLogin = () => {

    setUser({

      name: "Vishwas",

      email: "vishwas@example.com",

    });

  };

//   const handleLogout = () => {

//     setUser(null);

//   };

// Just An example tu understand Type assertion

  return (

    <div>

      <button onClick={handleLogin}>Login</button>

      {/\* <button onClick={handleLogout}>Logout</button> \*/}

      {/\* intteligence helps us tu use ? Automaticaly in our object \*/}

      <div>User name is {user?.name}</div>

      <div>User name is {user?.email}</div>

    </div>

  );

};

* 1. useReducer

If we have complex state logic where the next state depends on the previous state useReducer is preferable in this video we will learn how to type a useReducer Hook

import { useReducer } from 'react'

type CounterState = {

  count: number

}

type CounterAction={

    type:string,

    payload:number

}

const initialState = { count: 0 }

function reducer(state: CounterState, action: CounterAction) {

  switch (action.type) {

    case 'increment':

      return { count: state.count + action.payload }

    case 'decrement':

      return { count: state.count - action.payload }

    case 'reset':

      return initialState

    default:

      return state

  }

}

export const Counter = () => {

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

  return (

    <>

      Count: {state.count}

      <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>

        Increment 10

      </button>

      <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>

        Decrement 10

      </button>

    </>

  )

}

UseReducer Strict

In the previous Example we have no error is good but we can have a small problem in the future if

We wil add a reset action and we will see the problem so the solution is in this example:

import { useReducer } from 'react'

type CounterState = {

  count: number

}

type UpdateAction = {

  type: 'increment' | 'decrement'

  payload: number

}

type ResetAction = {

  type: 'reset'

}

  // Or we can do

// type UpdateAction = {

//     type: string

//     payload: number

//   }

//   type ResetAction = {

//     type: string

//   }

type CounterAction = UpdateAction | ResetAction

const initialState = { count: 0 }

function reducer(state: CounterState, action: CounterAction) {

  switch (action.type) {

    case 'increment':

      return { count: state.count + action.payload }

    case 'decrement':

      return { count: state.count - action.payload }

    case 'reset':

      return initialState

    default:

      return state

  }

}

export const Counter = () => {

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

  return (

    <>

      Count: {state.count}

      <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>

        Increment 10

      </button>

      <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>

        Decrement 10

      </button>

      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>

    </>

  )

}

* 1. UseContext

We create a folder within this folder we have created a few files which we need for our example ,we are going to consider using react context with a theme providing themes to our components is a very common use case of the context API and we will see how to do it with typescript.

Theme.ts

export const theme = {

    primary: {

      main: '#3f51b5',

      text: '#fff'

    },

    secondary: {

      main: '#f50057',

      text: '#fff'

    }

  }

ThemeContext.tsx

import { createContext } from "react";

import { theme } from "./theme";

type ThemeContextProviderProps = {

  children: React.ReactNode;

};

export const ThemeContext = createContext(theme)

export const ThemeContextProvider = ({

    children

  }: ThemeContextProviderProps) => {

    return <ThemeContext.Provider value={theme}>{children}</ThemeContext.Provider>

  }

App.tsx

<ThemeContextProvider>

        <Box />

      </ThemeContextProvider>

Box.tsx

import { useContext } from "react"

import { ThemeContext } from "./ThemeContext"

export const Box = () => {

    const theme=useContext(ThemeContext)

  return (

    <div style={{backgroundColor:theme.primary.main,color:theme.primary.text}}>ThemeContext</div>

  )

}

Example2:

UseContext Future Value

In this Example we will learn how to deal with typescript and the useContext hook with a future value

UserContext.tsx

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

export type AuthUser = {

  name: string;

  email: string;

};

type UserContextType = {

  user: AuthUser | null;

  setUser: React.Dispatch<React.SetStateAction<AuthUser | null>>;

};

type UserContextProviderProps = {

  children: React.ReactNode;

};

export const UserContext = createContext<UserContextType | null>(null);

export const UserContextProvider = ({ children }: UserContextProviderProps) => {

  const [user, setUser] = useState<AuthUser | null>(null);

  return (

    <UserContext.Provider value={{ user, setUser }}>

      {children}

    </UserContext.Provider>

  );

};

UserContext.tsx

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

export type AuthUser = {

  name: string;

  email: string;

};

type UserContextType = {

  user: AuthUser | null;

  setUser: React.Dispatch<React.SetStateAction<AuthUser | null>>;

};

type UserContextProviderProps = {

  children: React.ReactNode;

};

export const UserContext = createContext<UserContextType | null>(null);

export const UserContextProvider = ({ children }: UserContextProviderProps) => {

  const [user, setUser] = useState<AuthUser | null>(null);

  return (

    <UserContext.Provider value={{ user, setUser }}>

      {children}

    </UserContext.Provider>

  );

};

App.tsx

<UserContextProvider>

        <User/>

      </UserContextProvider>

user.tsx

import { useContext } from "react";

import { UserContext } from "./UserContext";

export const User = () => {

  const userContext = useContext(UserContext);

  const handleLogin = () => {

    if (userContext) {

      userContext.setUser({

        name: "Vishwas",

        email: "vishwas@example.com"

      });

    }

  };

  const handleLogout = () => {

    if (userContext) {

        userContext.setUser(null);

      }

  };

  return (

    <div>

      <button onClick={handleLogin}>LogIn</button>

      <button onClick={handleLogout}>LogOut</button>

      <div>User name is {userContext?.user?.name}</div>

      <div>User email is {userContext?.user?.email}</div>

    </div>

  );

};

A context always has to be created outside the component where as it future valie will always be set inside a component

* 1. UseRef

In this paragraph we will see the final hook which is the useRef Hookn

When it comes to seRef there are primarily two scenarios

* As a read-only ref for a dom element
* As a mutable value which can behave like an instance variable

DomRef.tsx

import { useEffect, useRef } from "react"

export const DomRef = () => {

    const inputRef=useRef<HTMLInputElement>(null)

    useEffect(()=>{

        if(inputRef.current){

        inputRef.current?.focus()

        }

    },[])

  return (

    <div>

        <input type='text' ref={inputRef}/>

    </div>

  )

}

MutableRef.tsx

import { useState, useRef, useEffect } from 'react'

export const MutableRef = () => {

  const [timer, setTimer] = useState(0)

  const interValRef = useRef<number | null>(null)

  const stopTimer = () => {

    if (interValRef.current) {

      window.clearInterval(interValRef.current)

    }

  }

  useEffect(() => {

    interValRef.current = window.setInterval(() => {

      setTimer(timer => timer + 1)

    }, 1000)

    return () => {

      stopTimer()

    }

  }, [])

  return (

    <div>

      HookTimer - {timer} -

      <button onClick={() => stopTimer()}>Stop Timer</button>

    </div>

  )

}

1. Class component

It might so happen that you’re working on a project that’s mean in development for a few years now and we’re going to come across class-based components

Class/Counter.tsx

import { Component } from 'react'

type CounterProps = {

  message: string

}

type CounterState = {

  count: number

}

/\*\* The count value is 5 \*/

export class Counter extends Component<CounterProps, CounterState> {

  state = {

    count: 0

  }

  handleClick = () => {

    this.setState(prevState => ({ count: prevState.count + 1 }))

  }

  render() {

    return (

      <div>

        <button onClick={this.handleClick}>Increment</button>

        {this.props.message} {this.state.count}

      </div>

    )

  }

}

App.js

<Counter message="test" />

1. Props
   1. Component Prop

Auth/Login.tsx

export const Login = () => {

    return <div>Please login to continue</div>

  }

Auth/Profile.tsx

export type ProfileProps={

    name:string

}

export const Profile = ({name}:ProfileProps) => {

    return <div>Private Profile component. Name is {name} </div>

  }

Auth/Private.tsx

import { Login } from "./Login";

import { ProfileProps } from "./Profile";

type PrivateProps = {

  isLoggedIn: boolean;

  Component: React.ComponentType<ProfileProps>;

};

export const Private = ({ isLoggedIn, Component }: PrivateProps) => {

  if (isLoggedIn) {

    return <Component name='Houssam'/>;

  } else {

    return <Login />;

  }

};

App.tsx

 <Private isLoggedIn={true} Component={Profile} />

* 1. Generic Prop
* In this paragraph we will talk about generics which is a typescript feature that is really useful when building react Components
* What we need is a way to tell Typescript that the type of items and the onClick handler item can Vary and generics are the way to that
* So Generic helps to avoid code duplication when we need to multiple types to be handled but at the same time

List.tsx

type ListProps<T> = {

    items: T[]

    onClick: (value: T) => void

  }

  export const List1 = <T extends string | number > ({

    items,

    onClick

  }: ListProps <T>) => {

    return (

      <div>

        <h2>List of items</h2>

        {items.map((item,index) => {

          return (

            <div key={index} onClick={() => onClick(item)}>

            {item}

            </div>

          )

        })}

      </div>

    )

  }

  export const List = <T extends { id: number }>({

    items,

    onClick

  }: ListProps<T>) => {

    return (

      <div>

        <h2>List of items</h2>

        {items.map(item => {

          return (

            <div key={item.id} onClick={() => onClick(item)}>

              {item.id}

            </div>

          )

        })}

      </div>

    )

  }

App.tsx

 <List1 items={[1, 2, 3]} onClick={(item) => console.log(item)} />

      <List

        items={[

          { id: 1, firstName: "Houssam", lastName: "Baaloul" },

          { id: 1, firstName: "Houssam", lastName: "Baaloul" },

          { id: 1, firstName: "Houssam", lastName: "Baaloul" },

        ]}

        onClick={(item) => console.log(item)}

      />

* 1. Restricting Props

We can restrict a component prop with typescript making use of the never type

Restruction/RunderNumber.tsx

type RandomNumberPropsNotGod={

    value: number

    isPositive: boolean

    isNegative?: boolean

    isZero?: boolean

}

  export const RandomNumberNotGod = ({

    value,

    isPositive,

    isNegative,

    isZero

  }: RandomNumberPropsNotGod) => {

    return (

      <div>

        {value} {isPositive && 'positive'} {isNegative && 'negative'}{' '}

        {isZero && 'zero'}

      </div>

    )

  }

  // Solutions

  type RandomNumberType = {

    value: number

  }

  type PositiveNumber = RandomNumberType & {

    isPositive: boolean

    isNegative?: never

    isZero?: never

  }

  type NegativeNumber = RandomNumberType & {

    isNegative: boolean

    isPositive?: never

    isZero?: never

  }

  type Zero = RandomNumberType & {

    isZero: boolean

    isPositive?: never

    isNegative?: never

  }

  type RandomNumberProps = PositiveNumber | NegativeNumber | Zero

  export const RandomNumber = ({

    value,

    isPositive,

    isNegative,

    isZero

  }: RandomNumberProps) => {

    return (

      <div>

        {value} {isPositive && 'positive'} {isNegative && 'negative'}{' '}

        {isZero && 'zero'}

      </div>

    )

  }

App.tsx

<RandomNumberNotGod value={2} isPositive isNegative isZero />

      {/\* <RandomNumber value={3} isPositive isNegative isZero /> \*/}

      {/\* Solution \*/}

      <RandomNumber value={3} isNegative />

1. Template Literals and Exclude

Template literals helps us to create many strings by making use of union and exclude helps us to exclude some strings.

Templateliterals/Toast.tsx

/\*\*

 \* Position prop can be one of

 \* "left-center" | "left-top" | "left-bottom" | "center" | "center-top" |

 \* "center-bottom" | "right-center" | "right-top" | "right-bottom"

 \*/

//Toast notification component

type HorizontalPosition = "left" | "center" | "right";

type VerticalPosition = "top" | "center" | "bottom";

type ToastProps = {

  position:

    | Exclude<`${HorizontalPosition}-${VerticalPosition}`, "center-center">

    | "center";

};

// Exclude pour enlenver center-center

export const Toast = ({ position }: ToastProps) => {

  return <div>Toast Notification Position - {position}</div>;

};

App.tsx

<Toast position="center-bottom" />

1. Wrapping HTML Elements

html/Button.tsx

/\*\*In this video we will learn how to wrap html elements and

 \* create custom components which can be used in react application

 \* if we're building a design sysytem or even a regular react app

 \* without relying on a UI compnent library it is common to create

 \* basic element like buttons and inputs with our own styling

 \*

 \*/

type ButtonProps = {

  variant: "primary" | "secondary";

  children: string;

} & Omit<React.ComponentProps<"button">, "children">;

//Omit takes an object type and remove the specified properties

//Omit helps us to tell TypeScript  to leave out the children type

// from the html element type

// Omitting props is somethig we will come across when building

// design system or component from scratch

export const CustomButton = ({ variant, children, ...rest }: ButtonProps) => {

  return (

    <button className={`class-with-${variant}`} {...rest}>

      {children}

    </button>

  );

};

Html/Input.tsx

type InputProps = React.ComponentProps<"input">;

export default function input(props: InputProps) {

  return <input {...props} />;

}

App.tsx

<CustomButton variant="primary">Primary Button</CustomButton>

1. Polymorphic Components

App.tsx

{/\* We want this text component to behave like different html

       elements bas on as prop Such a component is call Polymorphic componenet

       \*/}

<Text as='h1' size="lg">Heading</Text>

    <Text as='p' size="lg">Paragraph</Text>

      <Text as='div' size="lg" color="secondary">

        LAbel

      </Text>

Polymorphic/Text.tsx

/\*\*

 \* It is something we don't need unless we're building a

 \* component library or a design system for our project at work

 \*

 \*/

type TextOwnProps = {

  size?: "sm" | "md" | "lg";

  color?: "primary" | "secondary";

  children: React.ReactNode;

  as?: React.ElementType;

};

export const Text = ({ size, color, children, as }: TextOwnProps) => {

  const Component = as || "div";

  return <Component className={`class-with-${size}-${color}`}>{children}</Component>;

};

Advend code if we want to add htmFor props in Text.tsx

Une image contenant texte

Description générée automatiquementUne image contenant texte

Description générée automatiquement

1. Course Summary

Une image contenant table

Description générée automatiquement