Tuesday, April 11, 2023 1:57 PM

React with typeScript

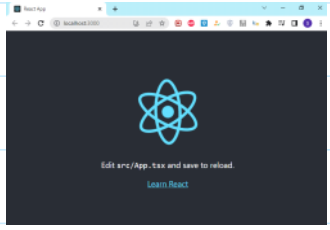
* with static type checking ⇒ helps to learn about potential bug than heding to browser and looking at the issue itself
* Describe the shape of an object hence providing better auto complete.
* makes maintanance & refactoring of large code base with much ease

To create react-app with typeScript:

> npx create-react-app App_Name --template typescript

> cd App_Name

> npm start / Yarn start ⇒ To start the project



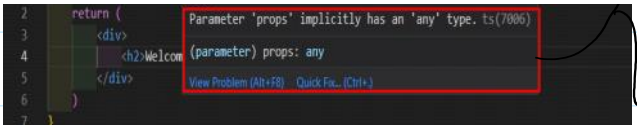
Type Inference

* Typescript infers types of variable when there is no explict information available in form of type annotations.
* The process of determining the appropriate types for expression based on how they are used is called Type Inference.
  ↳ above code shows Jsx.Element specifying JSX type

Typing Props in typescript:

* Provides components ⇒ a variety of values and function & can be used as object keys in components
* TypeScript validates type at compile time & PropTypes at runtime
* PropTypes are validator that can be used to make sure the data received is valid

Typescript is referring to any structure (Any type)



any structure ("Any" type)

↳ Specify type of props.

How to specify Prop Types.

* first Identify what type of prop is being passed → In this case String

```
<div>
    <h2>Welcome {props.name}</h2>
</div>
```

↳ String prop data

* Then specify the prop type (object: String) Define Structure

```
type GreetProps = {
    name: string
}
```

*

```
export const Greet = (props: GreetProps) => {
    return (
        <div>
            <h2>Welcome {props.name}</h2>
        </div>
    )
}
```

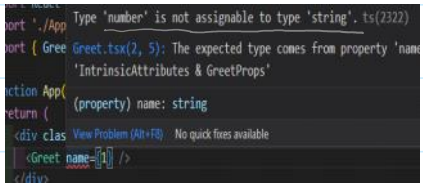
lastly specify the type name

↳ props: GreetProps.

Advantage of Using these extra code?

* Auto Complete the prop name

* When Invoking any other data type rather than String, typescript automatic point out



Types OR INTERFACE

Use types on building applications

↳ Use Interfaces on building libraries

Prop types


```
type GreetProps = {
    name: string,
    messageCount: number,
    isLoggedIn: boolean,
}
```

* Typescript notifies on missing props.

Props type as Object: → Create a seperate folder for Components.

How object type is Refactored !

```tsx
function App() {

  const personName = {
    first: 'Sahaj',
    last: 'Shakya',
  }

  return (
    <div className="App">
      <Greet name="Sahaj" messageCount={0} isLoggedIn={false} />
      <Person name={personName}/>
    </div>
  );
}

export default App;
```

Props passed as object

Invokes other component

```tsx
type PersonProps = {
    name: {
        first: string,
        last: string,
    }
}

export const Person = (props: PersonProps) => {
  return (
    <div>{props.name.first} and {props.name.last}</div>
  )
}
```

Object type specified for props

Defining Props

Props type as list

```tsx
const nameList =  [
  {
    first: 'Sahaj',
    last: 'Shakya',
  },
  {
    first: 'Sahaj',
    last: 'Shakya',
  },
  {
  first: 'Sahaj',
  last: 'Shakya',
}
]

return (
  <div className="App">
    <PersonList names={nameList}/>
  </div>
```

List

Pass as props

```tsx
type PersonListProps = {
    names: {
        first: string,
        last: string,
    }[]
}

export const PersonList = (props: PersonListProps) => {
  return (
      <div>
        { props.names.map((name) => {
            return (
                <h2 key={name.first}>{name.first} {name.last} </h2>
            )
        })
        }
      </div>
  )
}
```

Structure of list

Specified as list/Array

Props pass

Seperating & mapping

Prop String Validation

```tsx
return (
  <div className="App">
    <Greet name="Sahaj" messageCount={0} isLoggedIn={false} />
    <Person name={personName}/>
    <PersonList names={nameList}/>
    <Status status='asd' />
  </div>
);
}
```

```tsx
type StatusProps = {
    status: string
}

export const Status = (props: StatusProps) => {
    let message;
    if (props.status === 'loading') {
        message = 'Loading...';
    }
    else if (props.status === 'success') {
        message = 'Data Fetched Sucessfully';
    }
    else if (props.status = 'error') {
        message = 'Error fetching data'
    }
    return (
        <div>
            <h2>Status - {message} </h2>
        </div>
    )
}
```

Accept any String

* Here String type can be anything.
* To fix these => Specific validation of String can be specified on props.

```tsx
type StatusProps = {
    status: 'loading' | 'success' | 'error'
}
```

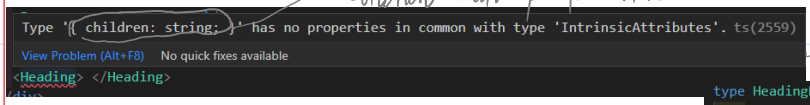
String can be specified on props.

Doesn't match any

Throws error on run time

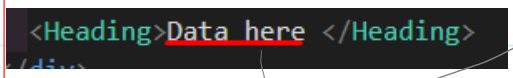
String data validation

Children props data

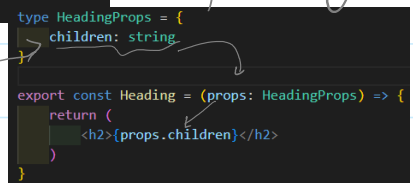


Solution auto specifies here.

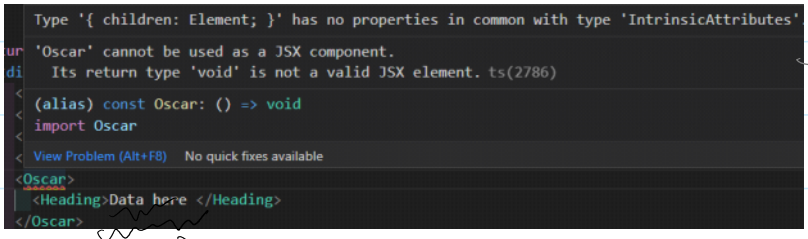
Just make the prop type as String



Pass as children



Passing components as props in typescript



What is type of React Component?

↳ various types are there

one of type is

children: React.ReactNode

Invoked

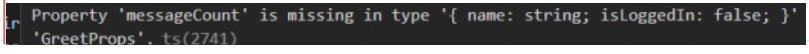
ReactNode type

Pass through as props

```
type OscarProps = {
    children: React.ReactNode
}

export const Oscar = (props: OscarProps) => {
    return(
        <div>
            {props.children}
        </div>
    )
}
```

When Component doesn't have to be passed?



In such cases, where no

```
Property 'messageCount' is missing in type '{ name: string; isLoggedIn: false; }'
'GreetProps'. ts(2741)
Greet.tsx(3, 5): 'messageCount' is declared here.
(alias) const Greet: (props: GreetProps) => JSX.Element
import Greet
View Problem (Alt+F8)  Quick Fix... (Ctrl+.)
<Greet name="Sahaj"  isLoggedIn={false} />
```

In such cases, where no msgCount needs to be passed, Component can be invoked without passing that props

↳ No messageCount is Sent

In such case ⇒ to tell typescript for excluding the props ⇒ Just add ? on type

```
type GreetProps = {
    name: string,
    messageCount?: number,
    isLoggedIn: boolean,
}
```

```
<Greet name="Sahaj"  isLoggedIn={false} />
```

Inform typescript to ignore the props

Event props: (Pass event as props)

x Event handler is a callback routine that operates asynchronously once an event takes place.

Basic Syntax → `type Name_of_Props = { eventHandler_func_Name: () => Return_type }`

type of event

`const eventHandler_fun_Name = () => { }`

```
<Button
  handleClick={() => {alert('Clicked')
}}
/>
```

function callback()

Main function

Main Place

```
type ButtonProps = {
    handleClick: (event: React.MouseEvent<HTMLButtonElement>) => void
}
export const Button = (props: ButtonProps) => {
    return(
        <div>
            <button onClick={props.handleClick}>Click Here</button>
        </div>
    )
}
```

Prop Callback Name

Specify event type (Button Click)

no return type

Callback()

```
<Button
  handleClick={(event, id) => {console.log('Clicked', event, id)
}}
/>
```

Pass event & number

```
type ButtonProps = {
    handleClick: (event: React.MouseEvent<HTMLButtonElement>, id: number) => void
}
export const Button = (props: ButtonProps) => {
    return(
        <div>
            <button onClick={(event) => props.handleClick(event, 1)}>Click Here</button>
        </div>
    )
}
```

Event type

Id type.

Callback with event & id

Handling Input Event

```
<Input
  value='aa' handleChange={(event) => console.log(event)}
/>
```

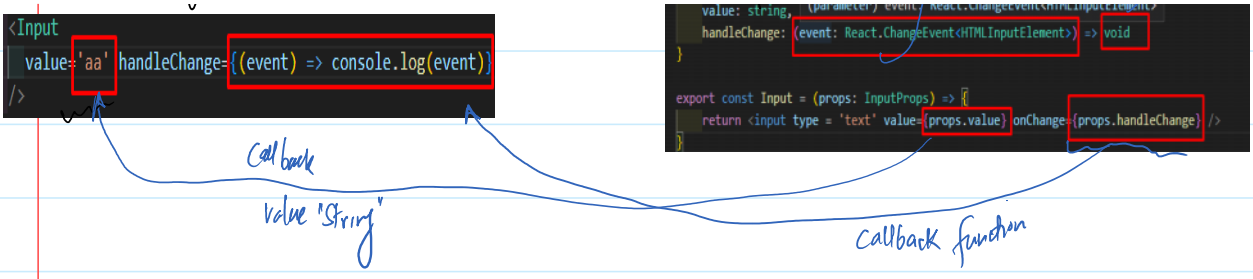
type of value

OnChange Event

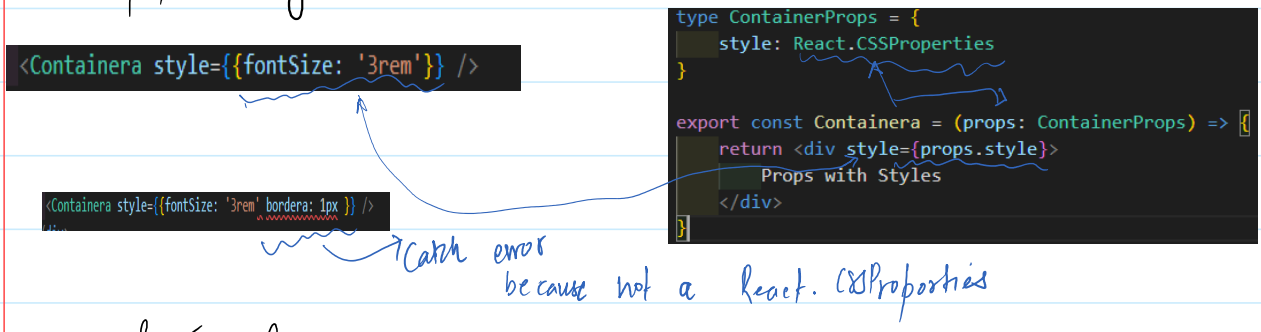
Return type

```
type InputProps = {
    value: string,
    handleChange: (parameter) event: React.ChangeEvent<HTMLInputElement>
}
```

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



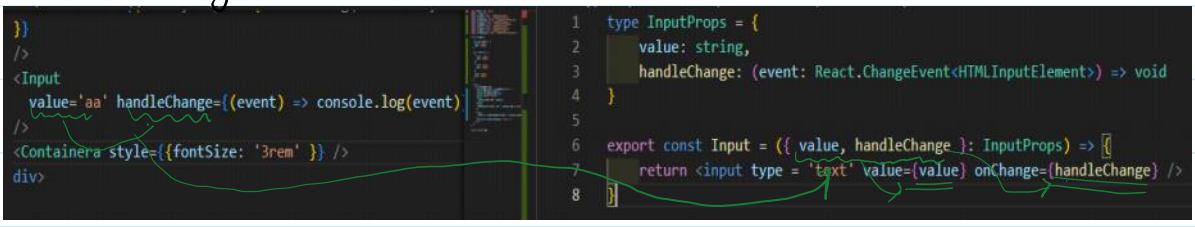
Style props Passing:



Types & Type for props

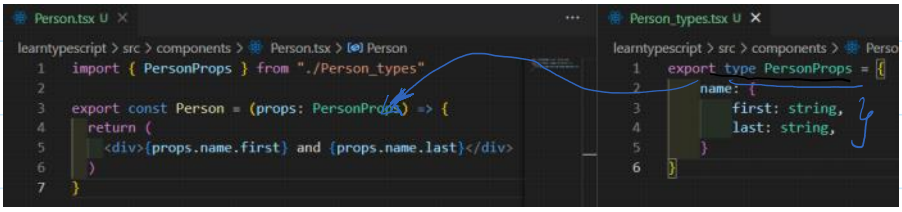
1) Destructuring

* creating new variable by extracting some values from stored data in objects or arrays.



2) Exporting types.

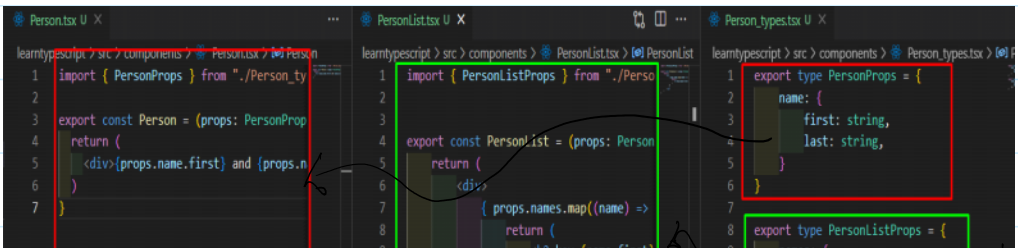
* creating seperate file for assigning types.



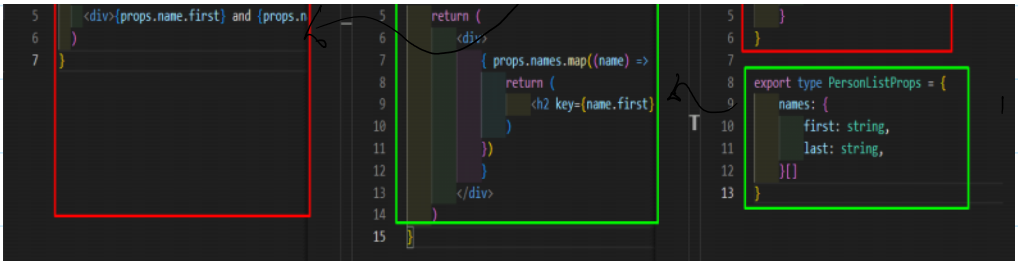
Exporting types on seperate files

3) Reusing types

* extract types and use it on multiple places



* Can be used on multiple places =>



places ⇒ Reusability & remove duplications.

UseState Hooks:

* Create a new folder "state" inside component folder

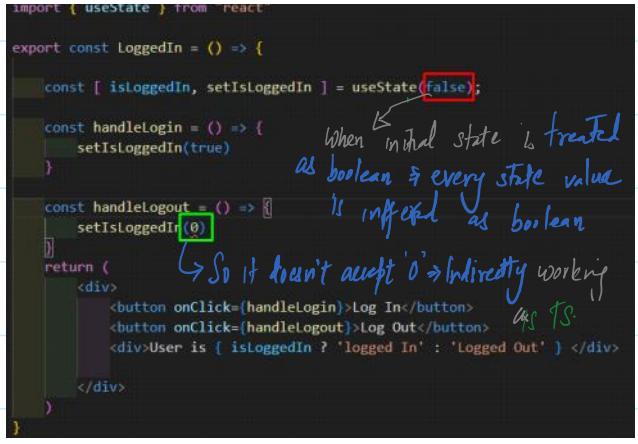
Step1: Import useState() from react hooks.
```
import { useState } from "react"
```

Step 2: Create a state variable with initial values.
```
const [ state, setState ] =
useState('Inital_Val');
```

Step3: Render the text based on state values.
```
{state ? 'Condition1': 'Condition2'}
```

Step 4: Create a handler function to change the respective State variable.
```
const handleerFunction = () => {
    setState(Change_Conditions)
}
```

But where is typescript? ⇓ This is plane Javascript.



* In Javascript, both false & '0' is treated as boolean expression but In typescript, once initial value is treated as false then it won't accept '0' as a boolean false expression ⇒ Indirectly type inferencing as a boolean type for True & false.

What if Initial State variable for useState() is unknown?

* when Initial State variable value is only known in future?

eg: When a user visits a website ⇒ User is not logged In by default.

i.e Initial value is null/none: Common thing is to set the initial State value as null.

```
import { useState } from "react"
```

```tsx
import { useState } from "react"

type AuthUser = {
    name: String,
    email: String,
}

export const User = () => {

    const [ user, setUser ] = useState(null);

    const handleLogin = () => {
        setUser({
            name: 'Sahaj',
            email: 'Sahaj@wsahaj.com',
        })
    }
```

Initially the user value is set to null that's why typescript doesn't let the state values to be assigned to other type

How to fix such issue?

* Explicitly specify the hook type & not rely on inference:

Include angle bracket (<>) after usestate hook & set value as null or other type

```tsx
import { useState } from "react"
type AuthUser = {
    name: String,
    email: String,
}
export const User = () => {
    const [ user, setUser ] = useState<AuthUser | null>(null);
    const handleLogin = () => {
        setUser({
            name: 'Sahaj',
            email: 'Sahaj@wsahaj.com',
        })
    }
    const handleLogout = () => {
        setUser(null)
    }
    return (
        <div>
            <button onClick={handleLogin}>Log In</button>
            <button onClick={handleLogout}>Log In</button>
            <div>User name is {user?.name}</div>
            <div>User email is {user?.email}</div>
        </div>
    )
}
```

Add a anglebracket with val can be either null or state value

Optional chaining operator (?) because user can be null

→ Optional chaining is automatically assigned ⇒ bc user can be null (so that typescript doesn't throws an error.

<AuthUser | null>(null)

AuthUser → If value; null → Null; (null) → Initial value

Type Assertion in typeScript

* Allows user to set the type of value and let compiler to not infer it. It has no effect during runtime whereas typecasting has effect on runtime

for eg: * Some type the useState value cannot be null ⇒ In such case type Assertion can be used (Telling compiler not to make the state value null)

```tsx
import { useState } from "react"
type AuthUser = {
    name: String,
    email: String,
}
export const User = () => {
    const [ user, setUser ] = useState<AuthUser>( {} as AuthUser );
    const handleLogin = () => {
        setUser({
            name: 'Sahaj',
            email: 'Sahaj@wsahaj.com',
        })
    }
    return (
        <div>
            <button onClick={handleLogin}>Log In</button>
            <div>User name is {user?.name}</div>
            <div>User email is {user?.email}</div>
        </div>
    )
}
```

Empty obj as Authtype

→ Type Assertion

* forcing compiler to not change the state type
* Syntax similar to useState for future state but forcing the initial & final state to be auth type

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

be auth type

* Type Assertion should only be used if the developer is sure that the state value will not be modified (force it to be specified by user).

UseReducer (Complex state operation where next state depends upon previous State)

```
import { useReducer } from "react";

const initalState = {
    count: 0,
};

function reducer(state,action) {
    switch(action.type) {
        case 'increment':
            return { count: state.count + action.payload };
        case 'decrement':
            return { count: state.count - action.payload };
        default:
            return state;
    }
}

export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState)
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

Initial State ⇒ count set to 0
updates it
action 0+10,
current state
data
High level state & action
0-10,
Array destructor to get individual state & update dispatch function
return
what action?
data

* The useReducer() Hook returns the current state & dispatch method.
* The dispatch function updates the state to different value & trigger a re-render

useReducer(reducer, initalArg, init)

* specify how state is updated.
* take state and action as arguments and returns next state

Value from which initial state is calculated.

How initial value is Calculated from

depends on next `init`

* initializer function that should return initial state.
* if not specified ⇒ initial state is set to initialArg.
* else set to result of calling init(initialArg)

* Returns two exact values

ie `const [state, dispatch] = useReducer(reducer, initalstate)`

state

During first run, it is set to init(initialArg)

dispatch (Doesn't Returns)

* updates the state to different value and initialize/ trigger a re-render.

React will set the next state to result of calling the reducer function.

ie `const [state, dispatch] = useReducer(reducer, { count: 0 });`

```
import { useReducer } from "react";
type CounterState = {
    count: number,
}
```

Here count state is number

```tsx
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type CounterAction = {
    type: string,
    payload: number
}
const initalState = {
    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 };

        default:
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

Here count state is number

type is string

returns back a counter (numerical) state back.

Here;

count => numerical (number) state

⇓

so type is set as number

Action => has currently two types

action type → string

payload → number

So set to both string & number.

More Strick Action type

```tsx
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type CounterAction = {
    type: string,
    payload: number
}
const initalState = {
    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 };

        default:
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

So what is the problem? ↳ Action set to string

Can accept any string not strictly specified which actions.

```tsx
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type CounterAction = {
    type: string,
    payload: number
}
const initalState = {
    count: 0,
};
function reducer(state: CounterState, action: CounterAction) {
    switch(action.type) {
        case 'reset':
            return { count: state.count + action.payload };

        case 'decrement':
            return { count: state.count - action.payload };

        default:
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

When set to any other string value, there is no error at runtime & compile time but logical error.

⇓ By default, it gets triggered.

How to solve it? => use temperal literal => to specify strick action type.

```tsx
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type CounterAction = {
    type: `increment` | `decrement`,
    payload: number
}
const initalState = {
    count: 0,
};
function reducer(state: CounterState, action: CounterAction) {
    switch(action.type) {
        case 'reset':
            return { count: state.count + action.payload };

        case 'decrement':
            return { count: state.count - action.payload };

        default:
            return state;
    }
}
```

only accepts either increment or decrement

error type caught.

```tsx
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type CounterAction = {
    type: `increment` | `decrement`,
    payload: number
}
const initalState = {
    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 };

        default:
            return state;
    }
}
export const Counter = () => {
```

Explicitly Defined

```
        default:
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

```
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    return (
        <>
            Count: { state.count }
            <button onClick={() => dispatch({ type: 'increment', payload: 10 })}>Increment 10</button>
            <button onClick={() => dispatch({ type: 'decrement', payload: 10 })}>Decrement 10</button>
        </>
    )
}
```

Adding a reset button:

```
import { useReducer } from "react";
type CounterState = {
    count: number,
}
type UpdateAction = {
    type: 'increment' | 'decrement',
    payload: number,
}
type ResetAction = {
    type: 'reset'
}
type CounterAction = UpdateAction | ResetAction

const initalState = {
    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 initalState;

        default:
            return state;
    }
}
export const Counter = () => {
    const [ state, dispatch ] = useReducer(reducer, initalState);
    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>
        </>
    )
}
```

Modern way instead of traditional way

Set payload action as it is.

Set reset action type as null

type CounterAction = UpdateAction | ResetAction → can be null / reset

const initalState = { count: 0, }; → can have default payload state type.

Similar to writing count: state, action

((action.payload) | 0)

* If reset button is added the value should be set to zero
  ie {type: 'reset' | payload: 0}
  But in case of complex state it's not ok.
* other way is to make the payload accept null value

type CounterAction = {
  type: 'increment' | 'decrement' | 'reset',
  payload? : number
}

but in case of type script ⇒ it should only accept either num or zero at instant which can be solved by traditional JS way

ie case 'increment'
  return {count: state.count + (action.payload) (0) }
  any value ← Add

UseContext for TypeScript:

* React Context is a way to maintain State Globally. It can be use together with useState() hook.

Step 1: Create Context using createContext

```
const UserContext = createContext( )
```

Step 2: Wrap the tee of Component that needs state Context, ie wrap child Component in the ContextProvider and supply state value.

```
function Component1 ( ) {
    const [ user, setUser ] = useState ('User1');
    return (
    ) <UserContext.Provider value = { user }>
```

```
function Component2 ( ) {
    const user = useContext (UserContext);
    return (
```

```
return (
  <UserContext.Provider value={user}>
    <h1>{`Hello ${user} again`}</h1>
    <Component2 user={user} />
  </UserContext.Provider>
);
```

Now component2 will have access of user too.

```
return (
  <h1>{`Hello ${user} again`}</h1>
)
```

Using TypeScript:

Step 1: Create context

```
import { createContext } from "react";
export const ThemeContext = createContext(theme);
```

Step 2: Define provider Component & ProviderComponent type

```
type ThemeConextProviderProps = {
  children: React.ReactNode
}
export const ThemeContextProvider = ( {children}:
ThemeConextProviderProps ) => {
  return <ThemeContext.Provider value={theme}>{ children }
</ThemeContext.Provider>
}
```

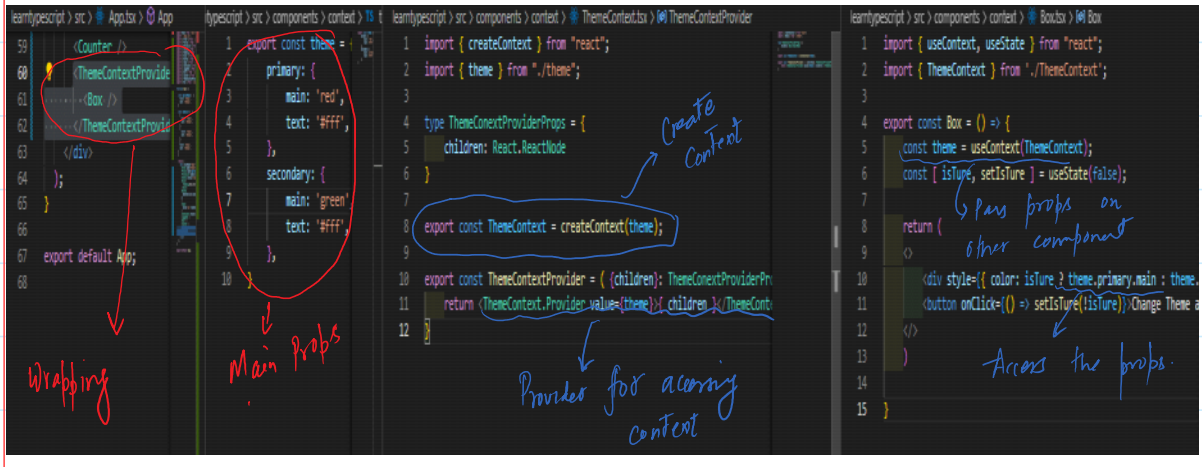
Props destructure

Render provider prop children

Step3: Wrap the other component with the provider

```
<ThemeContextProvider>
      <Box />
 </ThemeContextProvider>
```

Step4: Use the created Context on other Components.



Use Context for future values? (If future Context type is unknown)

Step1: Create a new context

Step2: Create the component the creates the context value

```
type  UserContextProviderProps = {
    children: React.ReactNode
}
export const UserContextProvider =
({ children }:UserContextProviderProps) => {}
```

* A user should be login & logout of the app. If login => Context should

hold the user type

* So Context should be a function that is able to login or logout as well as type AuthUser if login { export type AuthUser: { name: String }

In case of JS

```
}

type

}

expo

Unreachable code detected. ts(7027)
Unreachable code. eslint(no-unreachable)
Type '{ user: AuthUser | null; setUser: React.Dispatch<
assignable to type 'null'. ts(2322)
index.d.ts(339, 9): The expected type comes from proper
'IntrinsicAttributes & ProviderProps<null>'
(property) React.ProviderProps<null>.value: null
View Problem (Alt+F8)  Quick Fix... (Ctrl+.)
<UserContext.Provider value={{ user,setUser }}>
    {children}
</UserContext.Provider>

}

export const UserContext = createContext(null);
```

Initial the value is set to null

But there is some value with some type

TypeScript throws a type Error

Initially set to null

Step3: Specify the type for the context value

```
import { useState, createContext } from "react"
export type AuthUser = {
    name: string,
    email: string
}
type  UserContextProviderProps = {
    children: React.ReactNode
}
type UserContextType = {
    user: AuthUser | null,
    setUser: React.Dispatch<React.SetStateAction<AuthUser | null>>
}
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>

}
```

user value can be null or any

Could be UserContext type or null

Initially null

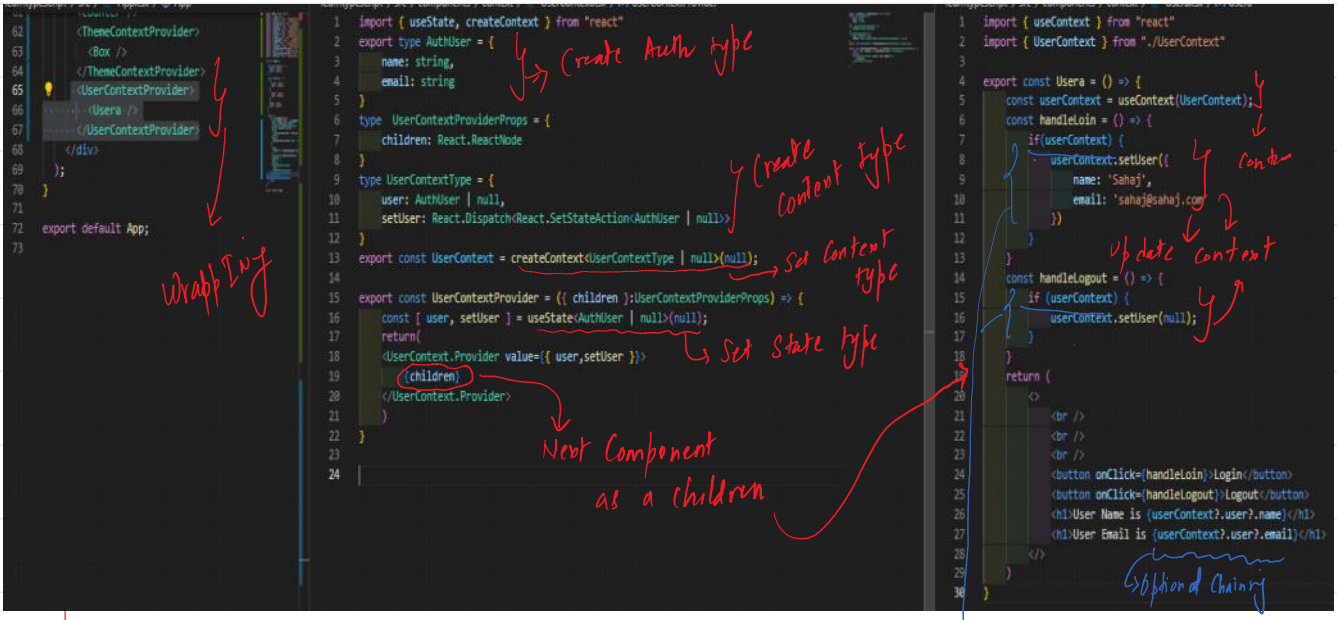
→ In detail, Specify the context type

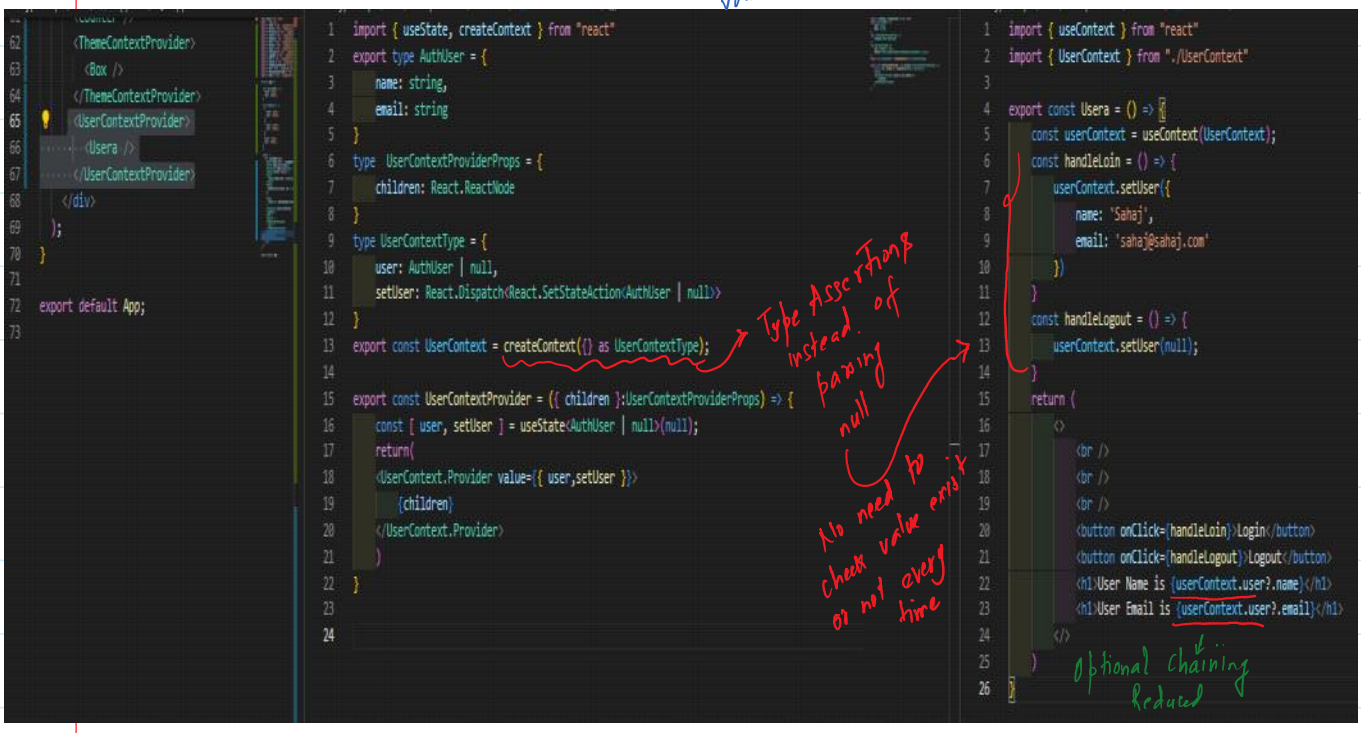
⇓

The type should be AuthUser or null but initally null

Step 4: wrapp it with UserContext Provider

```
<UserContextProvider>
    <Usera />
</UserContextProvider>
```





Useref In case of typeScript

DomRef

```
import { useEffect, useRef } from "react";

export const DomRef = () => {
    const InputRef = useRef<HTMLInputElement>(null!);
    useEffect(() => {
        InputRef.current.focus()
    },[])
    return (
        <div>
            <input type='text' ref={InputRef} />
        </div>
    )
```

type of ref
Remove optional chaing

Mutable Ref

```
import { useState, useEffect, useRef } 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>
    <input type='text' ref={InputRef} />
</div>
)
}
```

* for Dom ref => Specify the Dom ref Type

```
        stopTimer()
    }
},[])
return (
    <div>
        HookTimer= {timer}
        <button onClick={() => stopTimer()}>Stop Timer</button>
    </div>
)
}
```

↳ Specify the mutable Ref Type

Component as prop in TypeScript

prop

```
<Private isLoggedIn={true}
component={Profile} />
```

↳ Component as a prop

```
import { Login } from './Login';
import { ProfileProps } from './Profile';
type PrivateProps = {
    isLoggedIn: boolean,
    component: React.ComponentType<ProfileProps>
}
export const Private = ({ isLoggedIn, component:Component }: PrivateProps) => {
    if (isLoggedIn) {
        return <Component name='Sahaj'/>
    }
    else {
        return <Login />
    }
}
```

↳ type
↳ what component type
↳ Component: {Profile}
↳ type string
Indirectly return of Profile name.

```
export type ProfileProps = {
    name: string
}
export const Profile = ({name}: ProfileProps) => {
    return <div>Private Profile Component Name is {name}</div>
}
```

↳ Component props accepts string

Generics

* Generics allows creating 'type variable' which can be used to create classes, function & type alias that doesn't need to be explicitly defined.
    ↳ Done for reuseability of code

```
<Private isLoggedIn={true} component={Profile} />
{/* Generics */}
<List
  items={['A','B','C']}
  onClick={(item) => alert(item)}
/>
</div>
);
}

export default App;
```

↳ string array
↳ call back function no return

```
type ListProps = {
    items: string[],
    onClick: (value: string) => void
}
export const List = ({ items, onClick }: ListProps ) => {
    return (
        <div>
            <h2>List of Items</h2>
            {items.map ((item,index) => {
                return (
                    <div key={index} onClick={() => onClick(item)}>
                        {item}
                    </div>
                )
            })}
        </div>
    )
}
```

* upto here no problem

If user needs to invoke the component with numeric array?

One Soln is to accept both string & number type in prop type

```
type ListProps ={
    items: string[] | number[]
}
```

```
items={[
onClick=
/>
<List
items={[1,2,3]}
onClick={(item) => alert(item)}
/>
```

Type 'number' is not assignable to type 'string'. ts(2322)
Type 'number' is not assignable to type 'string'. ts(2322)
View Problem (Alt+F8) No quick fixes available

What is the problem still?
↳ only accepts array of number and strings

How to Solve?

```
items={[
    {
        first:'Sahaj',
        last: 'Shakya',
    },
]}
```

```
    },
]}
```

How to Solve?

↳ way to tell type-script that type of items and onClick handler can vary.

↳ How? using Generics.

How to Add Generics?

Step1: Add Generics to the propstyle.

```
type ListProps<T> = {
    items: T[]
    onClick: (value: string) => void
}
export const List = ({ 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>
    )
}
```

General convention is to add <T> which is an acronym for type but it's user defined, ie can be <Generics>

Step2: Specify what type of Generic?

```
type ListProps<T> = {
    items: T[],
    onClick: (value: T) => void
}
export const List = <T extends {}>({ items, onClick }: ListProps<
    return (
        <div>
            <h2>List of Items</h2>
            {items.map ((item,index) => {
                return (
                    <div key={index} onClick={() => onClick(item)
                        {item}
                    </div>
                )
            })}
        </div>
    )
}
```

↳ least restriction passing Generics.

Here "T" is a type Parameter is an object as a generic type within the function to operate on an object of any type that extends or implements the empty object type {} which ensures that only object type can be used as type argument for "T" when invoking it

Item error can be converted into ReactNode type (compatible with JSON syntax)

↳ one of the easy way To solve it is to convert it into json

{item} => {JSON.stringify(item)}

Restricting Props.

```
<RandomNumber
value={10}
isPositive={true}
isNegative={false}
isZero={false} />
```

while invoking only need isPositive or isNegative or isZero

But TypeScript will throw an error if

```
type RandomNumberProps= {
    value: number,
    isPositive: boolean,
    isNegative: boolean,
    isZero: boolean
}
export const RandomNumber = ({
    value,
    isPositive,
    isNegative,
    isZero,
}:RandomNumberProps)  => {
    return (
        <div>
            {value} {isPositive && 'positive'} {isNegative && 'Negative'} {isZero && 'zero'}
        </div>
    )
}
```

But TypeScript will throw an error if any one gets missed

How to solve it? ⇒ Restrict the props

```ts
type RandomNumberType = {
    value: number
}
type PositiveNumber = RandomNumberType & {
    isPostive: boolean,
    isNegative?: never,
    isZero?: never,
}
type NegativeNumber = RandomNumberType & {
    isNegative: boolean,
    isPostive?: never,
    isZero?: never,
}
type Zero = RandomNumberType & {
    isZero: boolean,
    isPostive?: never,
    isNegative?: never,
}
type RandomNumberProps = PositiveNumber | NegativeNumber | Zero;

export const RandomNumber = ({
    value,
    isPostive,
    isNegative,
    isZero,
}:RandomNumberProps)  => {
    return (
        <div>
            {value} {isPostive && 'positive'} {isNegative && 'Negative'} {isZero && 'zero'}
        </div>
    )
}
```

Always same

Different

Different

Different

Any value only

Temperal literals (create many string by means of unions)

```ts
type HorizontalPosition = 'left' | 'center' | 'right';
type VerticalPosition = 'top' | 'center' | 'bottom';
type Position = `${HorizontalPosition}-${VerticalPosition}`
```

```
(property) position: "center-center" | "center-top" | "center-bottom" | "left-center" | "left-top" | "left-bottom" | "right-center" | "right-top" | "right-bottom"
```

```ts
export const Toast = ({ position }: ToastProps) => {
    return <div>
        Toast Notification Position {position}
    </div>
}
```

```
<Toast position='left-center' />
```

Using temperal literal provides auto combination of position

There's a slight problem

Is for center, the position would be center-center

How to Resolve it? ⇒ use exclude

Caught

```
<Toast position='center-center' />
```

```
<Toast position='center' />
```

```ts
type HorizontalPosition = 'left' | 'center' | 'right';
type VerticalPosition = 'top' | 'center' | 'bottom';

type ToastProps = {
    position: Exclude<`${HorizontalPosition}-${VerticalPosition}`, 'center-center'> | 'center'
}

export const Toast = ({ position }: ToastProps) => {
    return <div>
        Toast Notification Position {position}
    </div>
}
```

OR

only this combo

Exclude

Wrapping HTML elements

In react, it's common to create a custom HTML elements with its own styling.

Sending classname as prop

```
<CustomButton variant='primary' />
```

```ts
type ButtonProps = {
    variant='primary' | 'secondary'
}
```

Sending classname as prop

<CustomButton variant='primary'/> invoking with className as a prop

variant='primary' | 'secondary'

export const CustomButton = ({variant}: ButtonProps) => {
return <button classname={`class-with-${variant}`}>Hello</button>

Sending large amount of prop data with children

type to pass every attributes of button

```
type ButtonProps = {
    variant: 'primary' | 'secondary'
} & React.ComponentProps<'button'>

export const CustomButton = ({ variant, children, ...rest }: ButtonProps) => {
    return <button className={`class-with-${variant}`} {...rest} >{children}</button>
}
```

spread operator to pass any attribute of button html type

<CustomButton variant='primary'>
Custom Button</CustomButton>

children → Advantage of this is that any thing, variable, tags can be invoked as children

ie <CustomButton> Hello → Children </CustomButton>

accepts any react Node.

```
(parameter) children: React.ReactNode

export const CustomButton = ({ variant, children, ...rest }: ButtonProps) => {
    return <button className={`class-with-${variant}`} {...rest} >{children}</button>
}
```

How to Restrict it?

→ Restrict children to accept string only using Omit <WHAT_TO_PASS, WHAT_TO_OMIT>

```
    last: 'Shakya',
  },
]}
This JSX tag's 'children' prop expects type 'string' which requires multiple children, but only a single child was provided. ts(2745)

(alias) const CustomButton: ({ variant, children, ...rest }: ButtonProps) => JSX.Element
import CustomButton

View Problem (Alt+F8)   No quick fixes available

<CustomButton variant='primary'><div>Hello</div></CustomButton>
<CustomInput type='text' placeholder='Custom Input'/>
```

```
1  type ButtonProps = {
2      variant: 'primary' | 'secondary';
3      children: string
4  } & Omit<React.ComponentProps<'button'>, 'children'>
5
6  export const CustomButton = ({ variant, children, ...rest }: Butt
7      return <button className={`class-with-${variant}`} {...rest}
8  }
```

Pass OMIT

→ only accepts String as children

```
{/* Template Literals and Exclude */}
<Toast position='center' />
{/* Wrapping HTML Elements */}
<CustomButton variant='primary'>Custom Button</CustomBu
<CustomInput type='text' placeholder='Custom Input'/>
```

```
1  type InputProps = React.ComponentProps<'input'>
2  export const CustomInput = (props: InputProps) => {
3      return <input {...props} />
4  }
```

every props treated as React Component

Extract Component Prop types (Reuse prop type with no access of that prop type)

Syntax

export const Name_of_Component = (props: React.ComponentProps<typeof Name_of_Component_To_extract_from>) => {
return (

... {props.name_of_props_from_where_it's_extracted}

```
<Greet
name="Sahaj"  isLoggedIn={false}
/>
<CustomComponent name="Sahaj"
isLoggedIn={false}/>
```

Same type as Greet

```
import { Greet } from "../Greet"
export const CustomComponent = (props: React.ComponentProps<typeof Greet>) => {
    return (
        <div>
            {props.name}
        </div>
    )
}
```

(Extract prop type of Greet

Polymorphic Components

- Component that can be rendered with different container element/node.
- Specify which React element → to use for it's root
  - eg: MaterialUI → <Typography> → ability to render as any HTML element for root node.

App.tsx M

learntypescript > src > App.tsx > App

```
136            <CustomComponent name="Sahaj" isLoggedIn={false}/>
137            <br />
138            {/* Polymorphic Components */}
139            <Test size='lg'>Heading</Test>
140            <Test size='md'>Paragraphs</Test>
141            <Test size='sm' color='secondary'>Heading</Test>
142        </div>
143    );
144  }
145
146  export default App;
```

Test.tsx U

learntypescript > src > components > polymorphic > Test.tsx > TextProps

```
1   import React from 'react'
2   type TextProps = {
3       size?: 'sm' | 'md' | 'lg'
4       color?: 'primary' | 'secondary'
5       children: React.ReactNode
6   }
7   export const Test = ({ size, color, children }: TextProps) => {
8       return (
9           <div className={`class-with=${size}-${color}`}>{children}
10      )
11  }
12
```

On adding/passing extra prop data, typescript will throw error

<Test test="test" size='lg'> Heading </Test>

- → behave like different html element based on test prop
  - → Such property is known as Polymorphic

How to do?

Step 1: → Just add the test property within the component (optional prop)

```
            <CustomComponent name="Sahaj" isLoggedIn={false}
            <br />
            {/* Polymorphic Components */}
            <Test size='lg'>Heading</Test>
            <Test size='md'>Paragraphs</Test>
            <Test size='sm' color='secondary'>Heading</Test>
        </div>
    );
}

export default App;
```

```
1   import React from 'react'
2   type TextProps = {
3       size?: 'sm' | 'md' | 'lg'
4       color?: 'primary' | 'secondary'
5       children: React.ReactNode
6       test?: string
7   }
8   export const Test = ({ size, color, children, test }: TextProps) => {
9       const Component = test || 'div'
10      return (
11          <div className={`class-with=${size}-${color}`}>{children}</div>
12      )
13  }
14
```

optional type

add/pass as props

→ make it behave like html element

Step 2: Specify valid Html type for test (React.ElementType)

```tsx
<CustomButton variant='primary'>Custom Button</C
<CustomInput type='text' placeholder='Custom Inp
{/* Extracting Component Prop Types */}
<CustomComponent name="Sahaj" isLoggedIn={false}
<br />
{/* Polymorphic Components */}
<Test test='h1' size='lg'>Heading</Test>
<Test size='md'>Paragraphs</Test>
<Test size='sm' color='secondary'>Heading</Test>
</div>
);
}

export default App;
```

```tsx
import React from 'react'
type TextProps = {
    size?: 'sm' | 'md' | 'lg'
    color?: 'primary' | 'secondary'
    children: React.ReactNode
    test?: React.ElementType
}
export const Test = ({ size, color, children, test }: TextProps) => {
    const Component = test || 'div'
    return (
        <div className={`class-with=${size}-${color}`}>{children}</div>
    )
}
```

will accept any html type & attributes

Suppose if another Html tag needs to be passed/invoked through it

```tsx
<Test test='label' size='lg' htmlFor='someId'>Heading</Test>
```

throw error as it's not mentioned/type not mentioned

(How?

↳ treat the test prop as generic element (that accepts any)

```tsx
import React from 'react'
type TextOwnProps<E extends React.ElementType> = {
    size?: 'sm' | 'md' | 'lg'
    color?: 'primary' | 'secondary'
    children: React.ReactNode
    test?: E
}
type TestProps<E extends React.ElementType>
    = TextOwnProps<E> &
        Omit<React.ComponentProps<E>, keyof TextOwnProps<E>>
export const Test = <E extends React.ElementType='div'>
    ({ size, color, children, test }: TestProps<E>) => {
    const Component = test || 'div'
    return (
        <div className={`class-with=${size}-${color}`}>{children}</div>
    )
}
```

Any Html element

treat the prop type as Generic prop

treat the test prop as part of Generic by

↳ itself as Generic prop type

↳ children also a react type ie to avoid duplications

Explicitly assign

↳ Every type as Generic type

Create Custom React Hook (Reuse method inside component)

```tsx
import React, { useEffect, useState } from 'react';
export const Application: React.FunctionComponent = () => {
    const [data, setData] = useState<any[]>([]);
    useEffect(() => {
        fetch('https://jsonplaceholder.typicode.com/todos')
            .then((res) => res.json())
            .then((data) => {
                setData(data);
            })
    },[])
    return (
        <div>
            Data Received
            <hr />
            {
                data.length > 0 ?
                    data.map((item, index) => {
                        return (
                            <div key={index}>
                                <p>
                                    Item Id: <strong>{item.id}</strong>
                                    <br />
                                    Completed: <strong>{item.completed ? 'Yes' : 'No'}</strong>
                                    <br />
                                    title: <strong>{item.title}</strong>
                                    <br />
                                    User Id: <strong>{item.userId}</strong>
                                </p>
                                <hr />
                            </div>
                        )
```

when every part of code is in one particular component

↳ * treat fetch as Sep hook

* treat display part as Sep component

```
                    </p>
                    <hr />
                </div>
            )
        })
        :
        <p>Loading Data</p>
    }
</div>
)
}
```

* treat display part as sep component

accepts url which is string

```
import React, { useEffect, useState } from 'react';
import { useFetch } from './hooks/useFetch';
export const Application: React.FunctionComponent = () => {
    const data = useFetch('https://jsonplaceholder.typicode.com/todos')
    return (
        <div>
            Data Received
            <hr />
            {
                data.length > 0 ?
                    data.map((item, index) => {
                        return (
                            <div key={index}>
                                <p>
                                    Item Id: <strong>{item.id}</strong>
                                    <br />
                                    Completed: <strong>{item.completed ?
                                    <br />
                                    title: <strong>{item.title}</strong>
                                    <br />
                                    User Id: <strong>{item.userId}</stro
                                </p>
                                <hr />
                            </div>
                        )
                    })
                    :
                    <p>Loading Data</p>
            }
        </div>
    )
}
```

↳ string data

```
import React, { useEffect, useState } from 'react'
export function useFetch(url: string) {
    const [data, setData] = useState<any[]>([]);
    useEffect(() => {
        fetch(url)
            .then((res) => res.json())
            .then((data) => {
                setData(data);
            });
    }, [url]);

    return data;
}
```

Advantage?

* By creating useFetch as seperate hook any component can invoke it with an API fetch request and get the API data

* useHook() ⇒ reusability for api data call.

Interface:

↳ defines property, methods, events which are member of interface

↳ only has declaration of members.

Let consider an Object

```
var Person = {
    FName: "ABC",
    LName: "BCD",
    sayHi: () => { return "Hi" }
};
```

⇒ Interface for this would be

```
{
    fName: String,
    LName: String,
    SayHi() => String,
}
```

Syntax ⇒ interface INTERFACE_NAME {

}

```
import React, { useEffect, useState } from 'react';
import { useFetch } from './hooks/useFetch';

interface TodoItem {
    id: number,
    userId: number,
    title: string,
    completed: boolean
}

export const Application: React.FunctionComponent = () => {
```

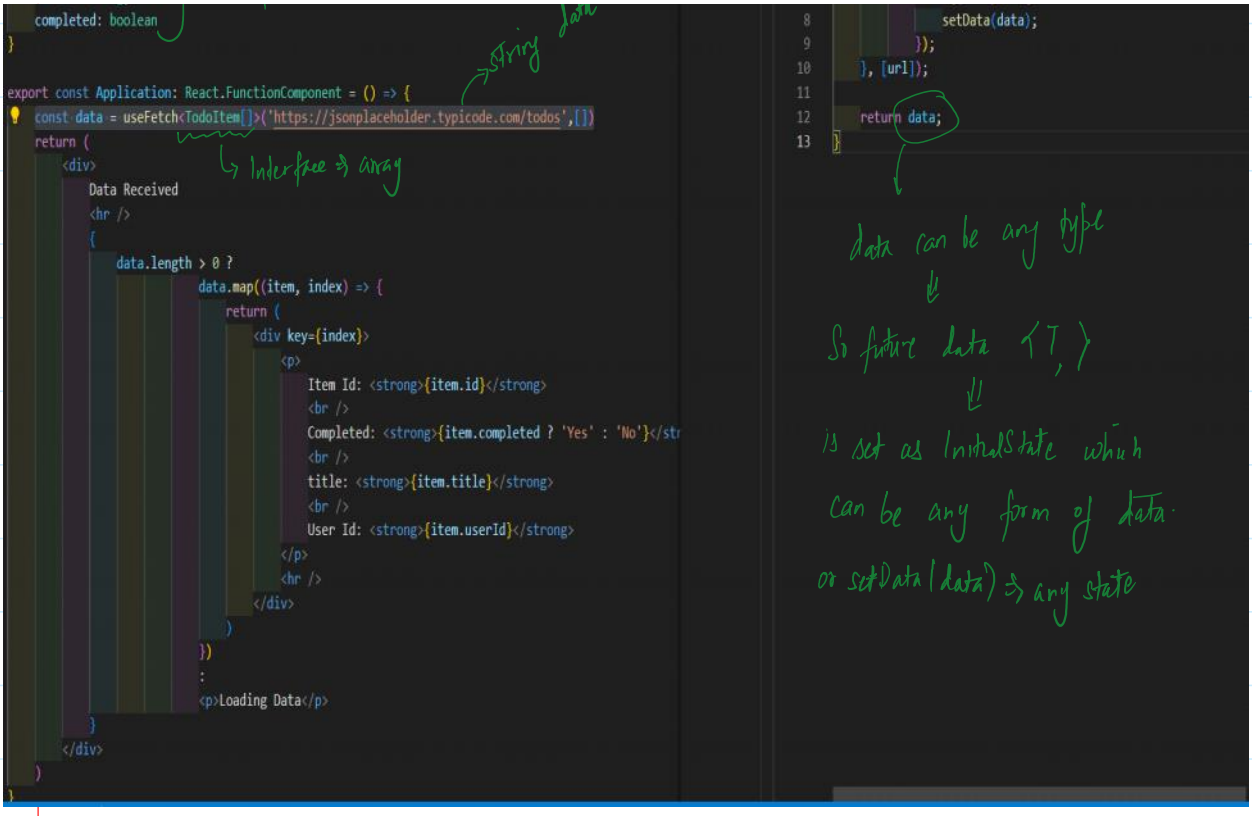
↳ Created Interface for returned data type

↳ String data

Generic type defined

future value as T generic type

```
import React, { useEffect, useState } from 'react';
export function useFetch<T,>(url: string, initialState: T) {
    const [data, setData] = useState<T>(initialState);
    useEffect(() => {
        fetch(url)
            .then((res) => res.json())
            .then((data) => {
                setData(data);
            });
    }, [url]);
```



x By using generic type parameter 'T' ⇒ useFetch() becomes more flexible and reusable. allowing it to work with any type of data

x useFetch<T,>(url: string, initialState: T) ⇒ url here is always string
↳ but initialState data can be any