TYPESCRIPT INTO REACT JS

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

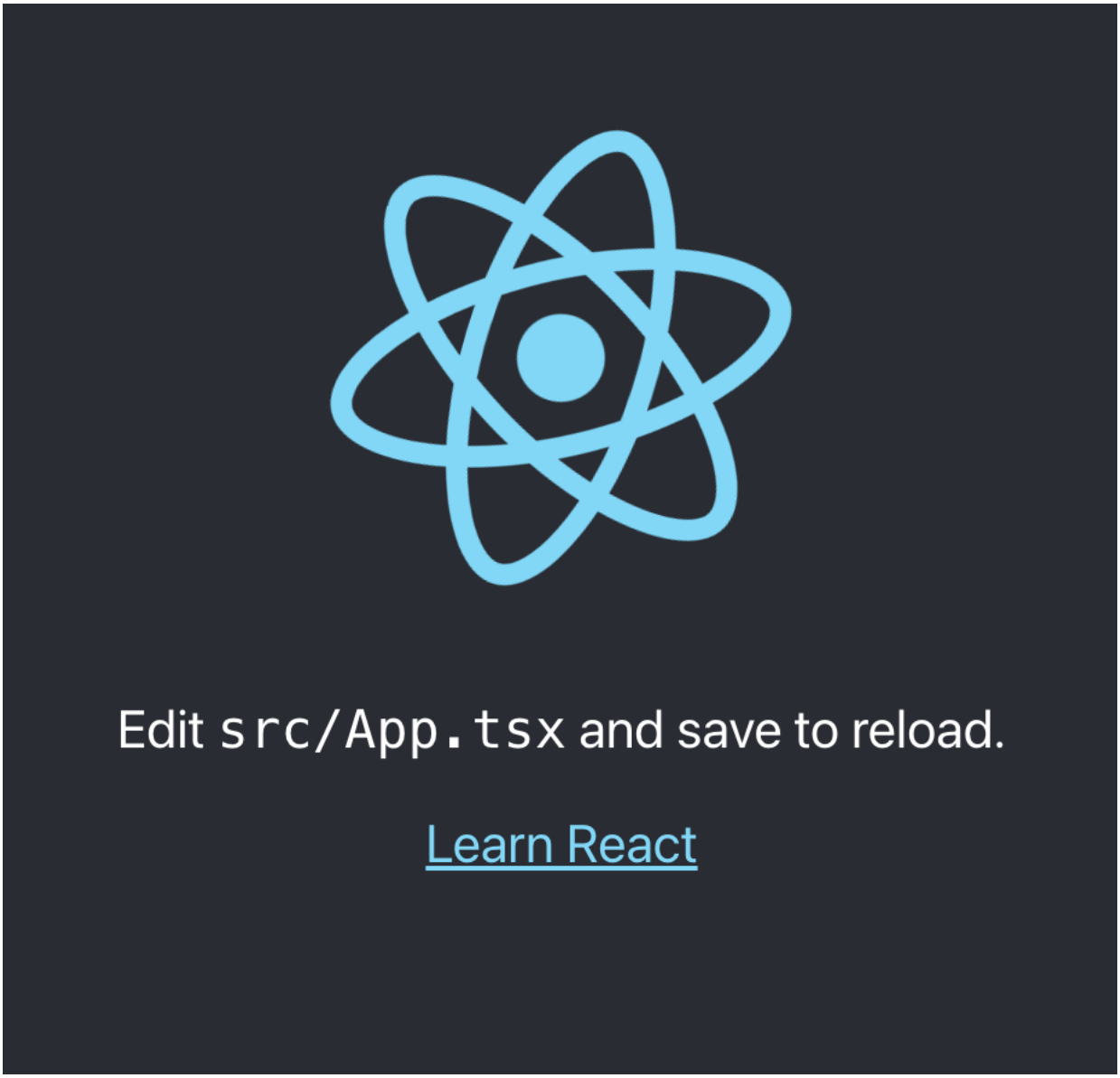
Intro and Installation

I hope you guys know React JS, TypeScript and JS very well or to an extent. This is a prerequisite to get the most out of this book. As the title says, it is a kutty book that describes the interesting facets of React with TypeScript. So, without reading much, let's get into the installation part.

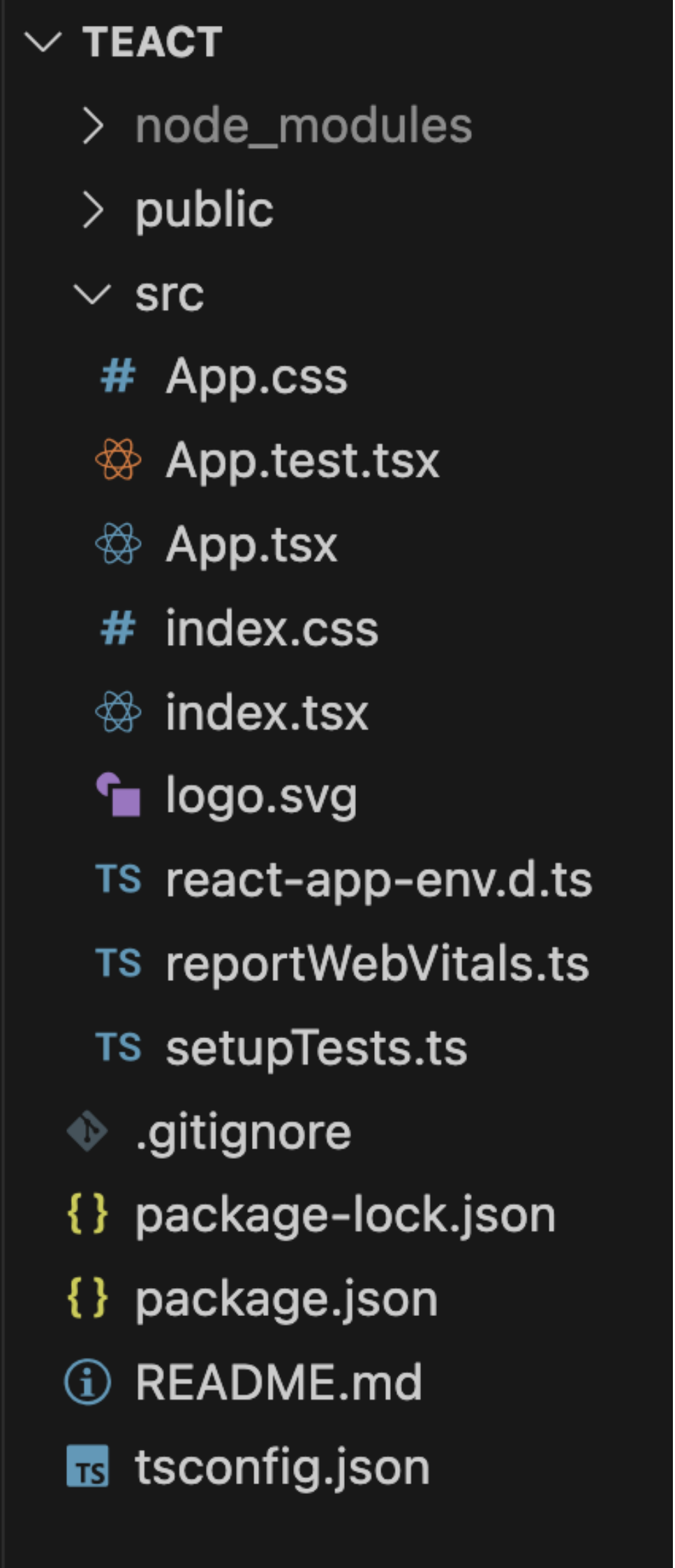
```
>> npx create-react-app teact --template typescript
```

The flag, "template" installs the react application with TypeScript configurations.

When we see the “Happy Hacking” message, it is the time to start. Get into our project folder, “teact” and type “npm start”.



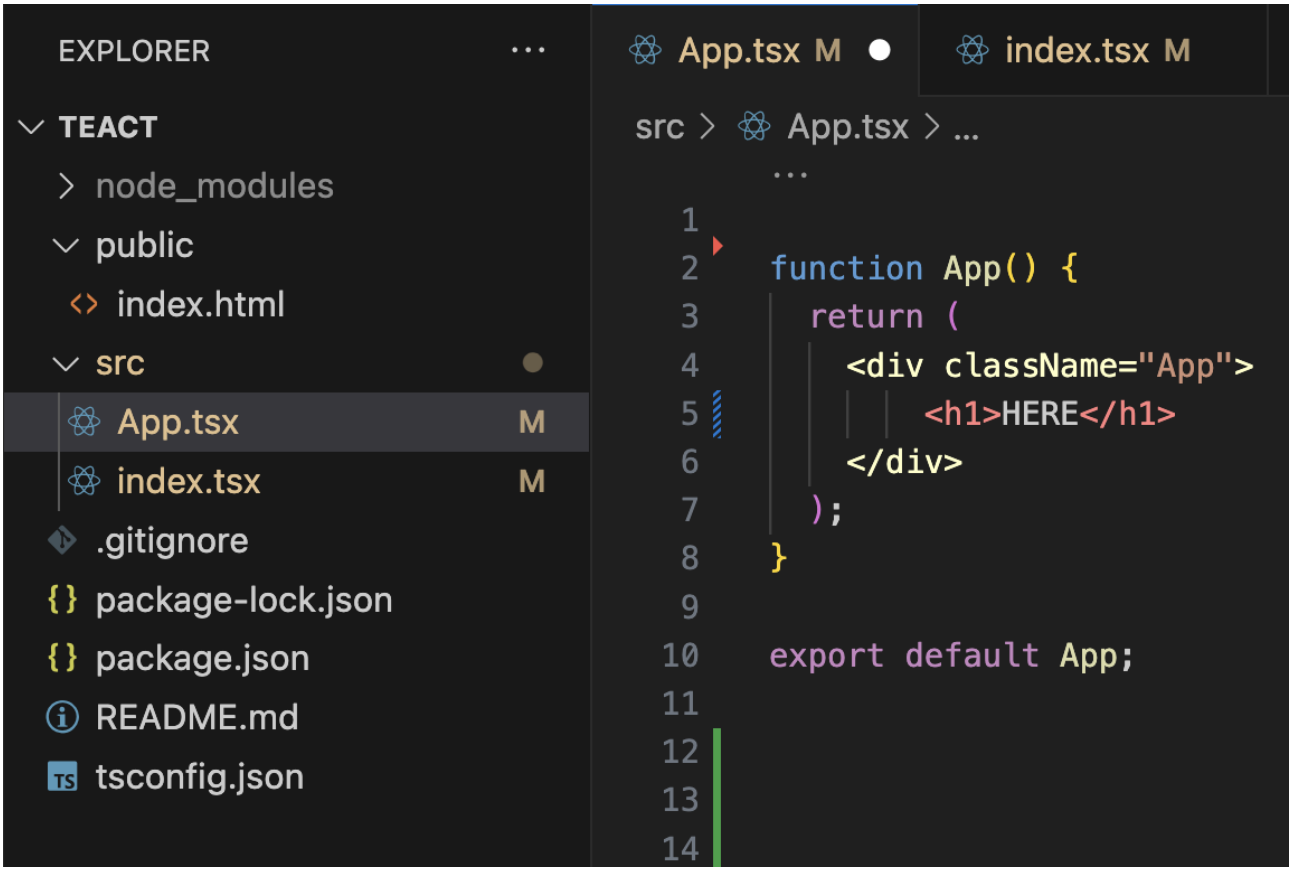
We can witness the rotating atom. But, there is little difference in the UI. It’s not App.jsx or App.js but App.tsx! Yes! It is the first facet of React when it is mingled with TypeScript. Let me code something now. All the files in our project folder are atypical!


TEACT
node_modules
public
src
App.css
App.test.tsx
App.tsx
index.css
index.tsx
logo.svg
react-app-env.d.ts
reportWebVitals.ts
setupTests.ts
.gitignore
package-lock.json
package.json
README.md
tsconfig.json

CASSOWARY

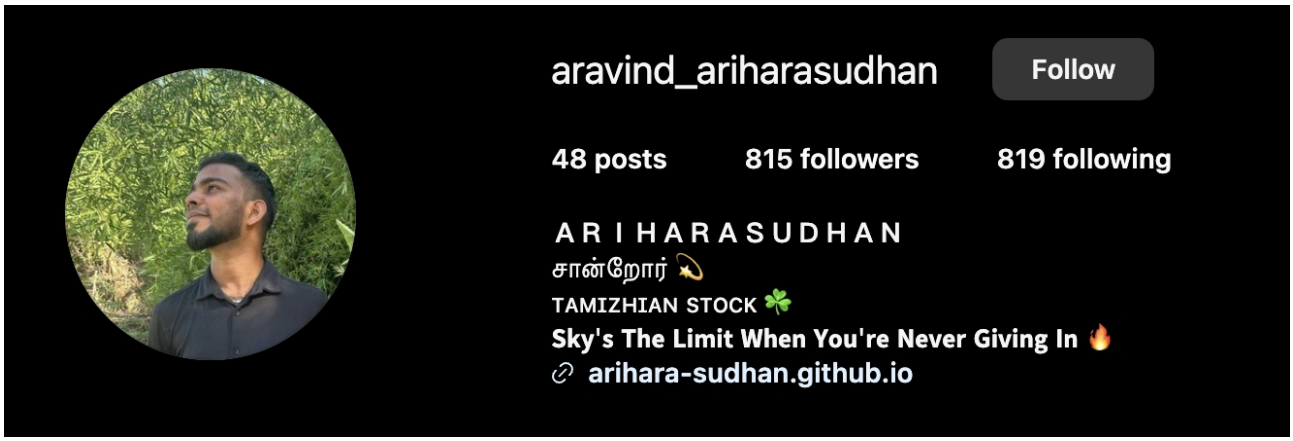
THE PROJECT

We'll be learning the stuff with a practical hands-on. Let me first clean the folder. I just want it to render a HERE message for now and it does.



HERE

Alright! I am going to create a page like the following.



Yes! It is instagram profile banner... (Don’t take it as a shameless self promotion). Let’s learn some basics and create our page.

COMPONENT TYPE

We can assign the FC (Functional Component) type to A Functional Component.

FUNCTIONAL COMPONENT: TYPED

```tsx
import React from "react";

const App: React.FC = () => {
  return (
    <div>
      <h1>I am ARI!</h1>
    </div>
  );
};

export default App;
```

It is not necessary to import React (In 17+ versions or React).

If you want to make it explicit, ah I mean clear to newbies, just write the type as FunctionComponent.

FUNCTIONAL COMPONENT: TYPED

```tsx
import React from "react";

const App: React.FunctionComponent = () => {
  return (
    <div>
      <h1>I am ARI!</h1>
    </div>
  );
};

export default App;
```

Here, we have to comprehend where it will be useful.

PROPS TYPE

Props can be strictly typed as shown below.

PROPS: TYPED

```tsx
interface AppProps {
  label: string;
  onClick: () => void;
}

const App: React.FC<AppProps> = (props) => {
  return <button onClick={props.onClick}>{props.label}</button>;
};

export default App;
```

While rendering, all the specified types must meet.

index.js

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

const root = ReactDOM.createRoot(
  document.getElementById('root') as HTMLElement
);
root.render(
  <React.StrictMode>
    <App onClick={()=>{alert("HELLO")}} label="CLICK"/>
  </React.StrictMode>
);
```

Missing props specified in type specifications will show squiggly lines. If I get rid of onClick in App rendering part, and <App label="CLICK"/> is the only thing left, The message will be like, Property “onClick” is missing in type “{label: string;}” but required in type “AppProps”. People usually use destructed props as the following.

App.js

```
const App: React.FC<AppProps> = ({onClick, label}) => {
   return <button onClick={onClick}>{label}</button>; };
```

We can directly set the type without constructs like interface.

App.js

```
const App: React.FC<{
  label: string;
  onClick: () => void;}> = ({onClick, label}) => {
  return <button onClick={onClick}>{label}</button>;
};

export default App;
```

We cannot have props without their types defined explicitly as of the current versions of TypeScript in JSX.

JSX TO TSX

But, there are some level of type inferences in cases like using an attribute which doesn't exist in a particular element.

```
const App: React.FC<AppProps> = ({onClick, label}) => {
  return <button href="https://ari.com" onClick={onClick}>{label}</button>;
};
```

The error message looks like, Type '{ children: string; href: string; onClick: () => void; }' is not assignable to type'DetailedHTMLProps<ButtonHTMLAttributes<HTMLButtonElement>,HTMLButtonElement>'.

Property 'href' does not exist on type 'DetailedHTMLProps<ButtonHTMLAttributes<HTMLButtonElement>,HTMLButtonElement>'. Did you mean 'ref'? I believe that You don’t like this message! TypeScript can deduce the types of variables as in Vanilla JS.

```
const App: React.FC<AppProps> = ({onClick, label}) => {
  let a = 23;
  a = 34;
  a = "Ari"; //Type 'string' is not assignable to type 'number'
  return <button onClick={onClick}>{label}</button>;
};
```

OPTIONAL PROPS

It's not a big deal for the people who know interface to define optional types in Interface. It helps in defining Optional Props.

OPTIONAL PROPS : ?

```tsx
interface AppProps {
  label: string;
  onClick?: () => void;
}

const App: React.FC<AppProps> = ({onClick, label}) => {
  return <button onClick={onClick}>{label}</button>;
};

export default App;
```

It is not required to pass the onClick props.

```tsx
<React.StrictMode>
  <App label="CLICK"/>
</React.StrictMode>
```

We can have default props too...

App.js

```tsx
interface AppProps {
  label: string;
  onClick?: () => void;
  disabled?: boolean;
}

const App: React.FC<AppProps> = ({ label, onClick, disabled = false }) => {
  return (
    <button onClick={onClick} disabled={disabled}>
      {label}
    </button>
  );
};

export default App;
```

If you like, you can also use the defaultProps property of the component object.

TYPE OF THE STATE

We can explicitly type the state as shown below.

STATE TYPE

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

const App: React.FC = () => {
  const [count, setCount] = useState<number>(0);
  return (
    <h1>{count}</h1>
  );
};

export default App;
```

If we don't have specified the type explicitly, it will be inferred.

```tsx
const App: React.FC = () => {
  const [count, setCount] = useState(0);
  return (
    <h1 onClick={()=>{ setCount("Ari") }}>{count}</h1>
  );
};
```

In the snippet given above, we are trying to set a string value to a number state. Since the type of the state is inferred, it will cause type mismatch as “Argument of type 'string' is not assignable to parameter of type 'SetStateAction<number>'.”

Let’s play with complex types.

```
type Bird = {
  name: string,
  age: number
}

const App: React.FC = () => {
  const [bird, setBird] = useState<null | Bird>(null);
  return (
    <h1>BIRD</h1>
  );
};
```

If we have to define an array over there in the state, there are two sort of syntaxes. One is the usual one we use! Meanwhile, the other one is what uses Array object.

```tsx
const App: React.FC = () => {
  const [nums, setNums] = useState<number[]>([1, 2, 3]);
  const [ages, setAges] = useState<Array<number>>([21, 22]);
  return (
    <h1>BIRD</h1>
  );
};
```

No issues if you like to use union of types.

```tsx
const App: React.FC = () => {
  const [nameOrAge, setNameOrAge] = useState<string | number>("Ari");
  return (
    <>
      <button onClick={()=> {setNameOrAge(21)}}>CLICK</button>
      {/*Argument of type 'boolean' is not assignable to parameter of type
       'SetStateAction<string | number>'*/}
      <button onClick={()=> {setNameOrAge(true)}}>CLICK</button>
    </>
  );
};
```

We can also type the most complex types and actions in useReducer hook.

COMPLEX STATES

```tsx
import React, { useReducer } from "react";

interface CounterState {
  count: number;
  step: number;
}

type CounterAction =
  | { type: "increment" }
  | { type: "decrement" }
  | { type: "setStep", payload: number };

const counterReducer = (state: CounterState, action: CounterAction): CounterState => {
  switch (action.type) {
    case "increment":
      return { ...state, count: state.count + state.step };
    case "decrement":
      return { ...state, count: state.count - state.step };
    case "setStep":
      return { ...state, step: action.payload };
    default:
      return state;
  }
};

const CounterComponent: React.FC = () => {
  const initialState: CounterState = { count: 0, step: 1 };
  const [state, dispatch] = useReducer(counterReducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <p>Step: {state.step}</p>
      <button onClick={() => dispatch({ type: "increment" })}>Increment</button>
      <button onClick={() => dispatch({ type: "decrement" })}>Decrement</button>
      <button onClick={() => dispatch({ type: "setStep", payload: 2 })}>Set Step to 2</button>
    </div>
  );
};

export default CounterComponent;
```

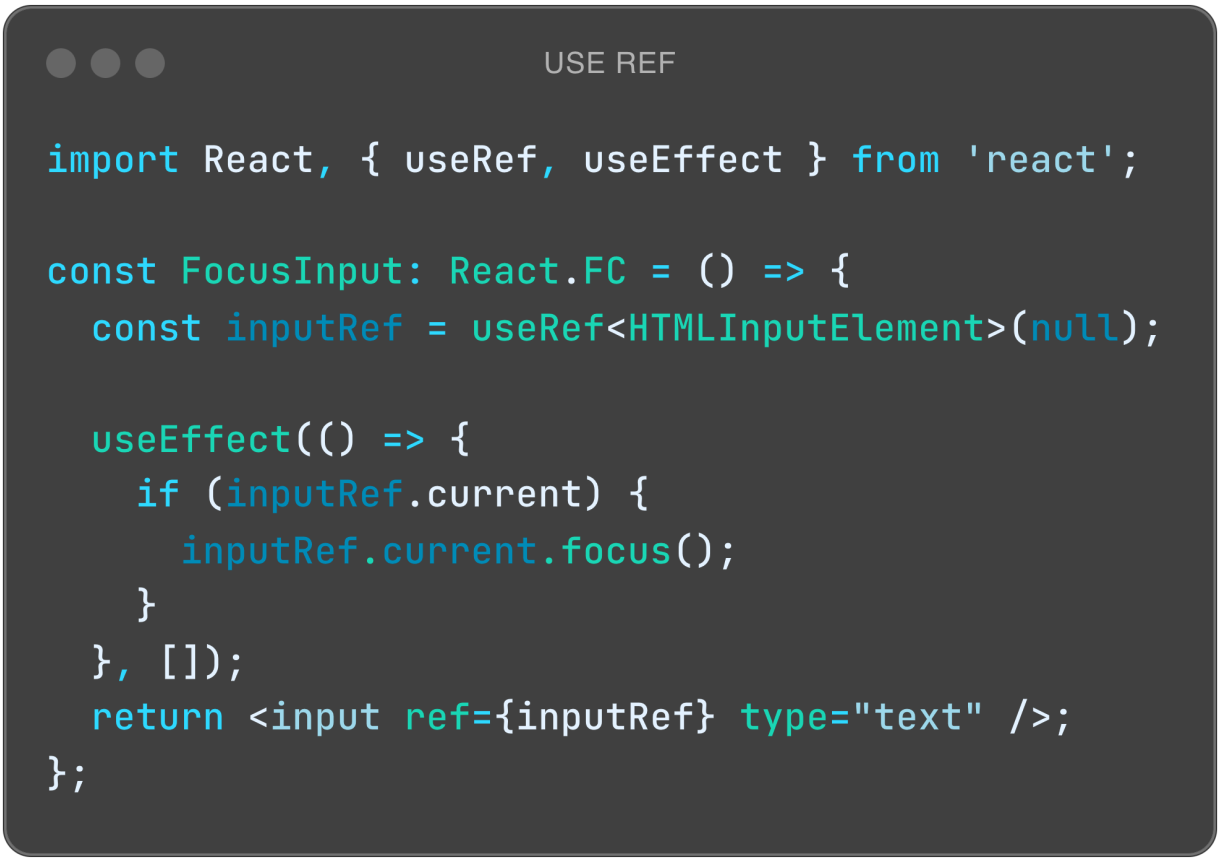
JUST USING THE TYPE

If the type is defined in useState or useReducer like state management hook, just use them in other hooks such as useEffect as it turns out.

USE EFFECT

```tsx
interface Props {
  title: string;
}
const VanakkamComponent = ({ title }: Props) => {
  useEffect(() => {
    document.title = title;
  }, [title]);
  return <div>{title}</div>;
};
```

Meanwhile, it’s quite fascinating with useRef hook.



Types such as HTMLInputElement in are predefined types that represent various DOM elements.

There are more like this...

DOM ELEMENTS: TYPES

```
const elementRef = useRef<HTMLElement | null>(null);
const inputRef = useRef<HTMLInputElement | null>(null);
const textareaRef = useRef<HTMLTextAreaElement | null>(null);
const buttonRef = useRef<HTMLButtonElement | null>(null);
const divRef = useRef<HTMLDivElement | null>(null);
const anchorRef = useRef<HTMLAnchorElement | null>(null);
const imgRef = useRef<HTMLImageElement | null>(null);
const selectRef = useRef<HTMLSelectElement | null>(null);
const formRef = useRef<HTMLFormElement | null>(null);
const canvasRef = useRef<HTMLCanvasElement | null>(null);
const videoRef = useRef<HTMLVideoElement | null>(null);
const svgRef = useRef<SVGElement | null>(null);
```

The types like these and other DOM element types can hold null when we explicitly define them to account for the possibility that the reference might not yet be assigned to an element.

It can also be used for normal types.

```tsx
const App: React.FC = () => {
  const dataRef = useRef<number>(1);
  const updateData = () => {
    dataRef.current = 'React';
  };
  return (
    <div>
      <button onClick={updateData}>Update Data</button>
    </div>
  );
};
```

And why not to use them for complex types?

```tsx
interface AriType {
  id: number;
  name: string;
}
const App: React.FC = () => {
  const dataRef = useRef<AriType>({name: "Aravind", id: 29});
  const updateData = () => {
    dataRef.current.name = 'Ari';
    dataRef.current.id = 77;
  };
  return (
    <div>
      <button onClick={updateData}>Update Data</button>
    </div>
  );
};
```

TYPING THE EVENTS

It is possible to type the events. TypeScript provides specific types for each kind of event, which should be used to ensure correct typing. The event handler function signature looks like the following:

EVENT HANDLING

```
const handleEvent = (event: React.EventType<ElementType>) => {
  // Handle the event down!!!!!!!!!
};
```

Following includes an example of defining an event handler.

EVENT TYPES

```tsx
const App: React.FC = () => {
  const handleClick = (e: React.MouseEvent<HTMLButtonElement>) => {
    alert("HEY BUDDY! BECOME MUDDY!")
  }
  return (
    <div>
      <button onClick={handleClick}>CLICK ME</button>
    </div>
  );
};

export default App;
```

There are some such Event Types.

EVENT TYPES

```
React.MouseEvent<T>: onClick, onMouseEnter, onMouseLeave
React.KeyboardEvent<T>: onKeyDown, onKeyUp
React.FocusEvent<T>: onFocus, onBlur
React.FormEvent<T>: onSubmit, onReset
React.ChangeEvent<T>: onChange
React.InputEvent<T>: onInput
AND SO ON
```

CONTEXT API

To create a context in TypeScript, we typically start by defining the shape of the context data and then create the context itself.

CONTEXT API

```tsx
import React, { createContext, ReactNode, useContext, useState } from 'react';

interface AppContextType {
    count: number;
    increment: () => void;
}

const AppContext = createContext<AppContextType | undefined>(undefined);
const AppProvider: React.FC<{ children: ReactNode }> = ({ children }) => {
    const [count, setCount] = useState<number>(0);
    const increment = () => {
        setCount((prev) => prev + 1);
    };
    return (
        <AppContext.Provider value={{ count, increment }}>
            {children}
        </AppContext.Provider>
    );
};

export { AppProvider, AppContext };
```

The context created can be used accordingly.

CONTEXT API

```
import React from 'react';
import { AppProvider } from './AppContext';
import Counter from './Counter';

const App: React.FC = () => {
    return (
        <AppProvider>
            <Counter />
        </AppProvider>
    );
};

export default App;
```

CONTEXT API

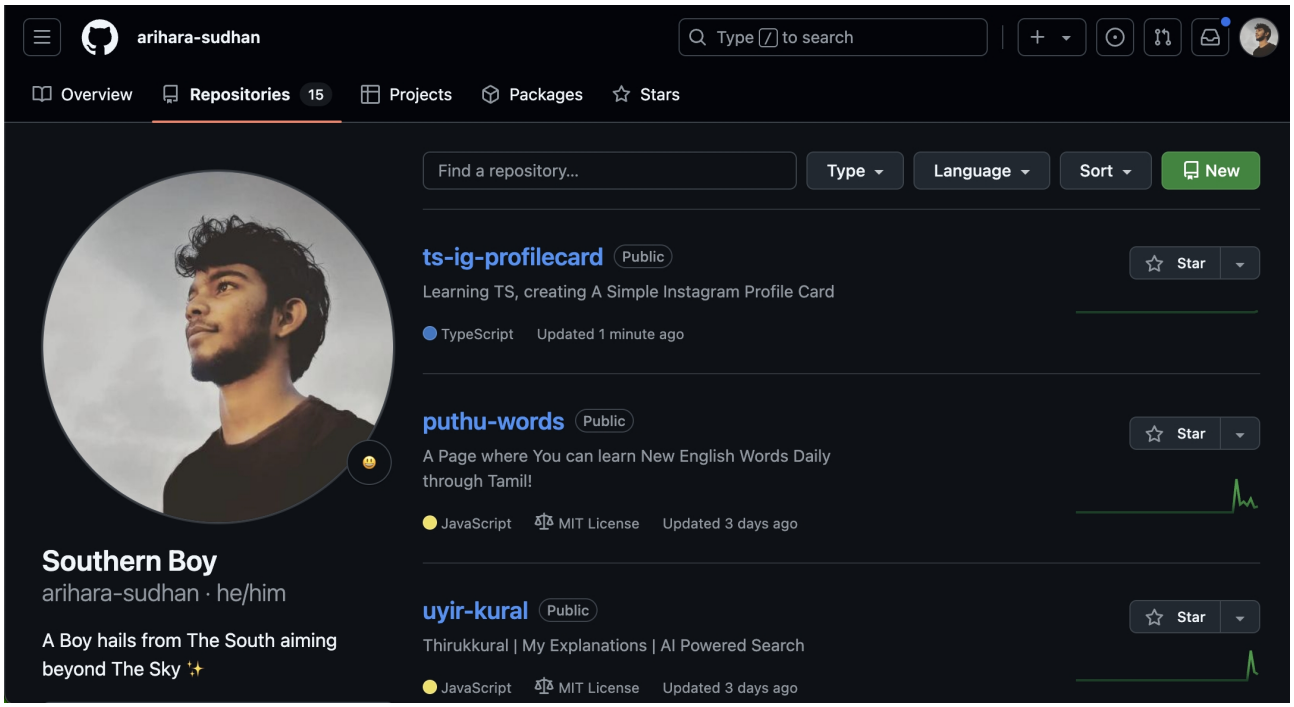
```
import React from 'react';
import { AppContext } from './AppContext';

const Counter: React.FC = () => {
    const context = useContext(AppContext);
    const { count, increment } = context;
    return (
        <div>
            <h1>Count: {count}</h1>
            <button onClick={increment}>Increment</button>
        </div>
    );
};

export default Counter;
```

THE PROJECT IS OUT

The Project I had a notion to create is created! Check it in my GitHub Repo:



You can check it here: https://github.com/arihara-sudhan/ts-ig-profilecard

Thank You from

CASSOWARI