React with TypeScript

https://github.com/peerberger/TypeScriptReact

What is TypeScript?

- TypeScript is like an extension of JavaScript.
- It adds new features to JS that help building bigger apps.
  Features like:
  - Strong types - so you can't change the type of variables, which means you can catch a lot of bugs without running the app, because the editor will tell you the type is wrong.
- Since TS is just an extension of JS, JS code is valid TS code (but not vice versa).

Creating a React project with TypeScript

- You can create a React project that uses TS with the normal npm command, you simply need to add the **--template typescript** flag.
  Ex:

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

Using TypeScript in React

- Let's create a new component named "TextField".
  Right now it looks like a regular js react component.
  Ex:

```
const TextField = () => {
    return (
        <div>
            <input />
        </div>
    );
}

export default TextField;
```

- Remember one of ts features is that it's strongly typed?
  Well, you can now define the component as a **Function Component** with a type from the react library.
  Ex:

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

const TextField : FC = () => {
```

```
    return (
        <div>
            <input />
        </div>
    );
}

export default TextField;
```

- One benefit of doing so, is that you can strongly define the properties the component is supposed to get.
  To do that, pass an object with the property names and types in the arrow brackets, as follows.
  Ex:

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

const TextField : FC<{ text: string }> = () => {
    return (
        <div>
            <input />
        </div>
    );
}

export default TextField;
```

- When you start having more and more properties, you can write it simpler by declaring the object as an **interface**, and then passing it instead.
  Ex:

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

interface Props {
    text: string
}

const TextField : FC<Props> = () => {
    return (
        <div>
            <input />
        </div>
    );
}
```

```
export default TextField;
```

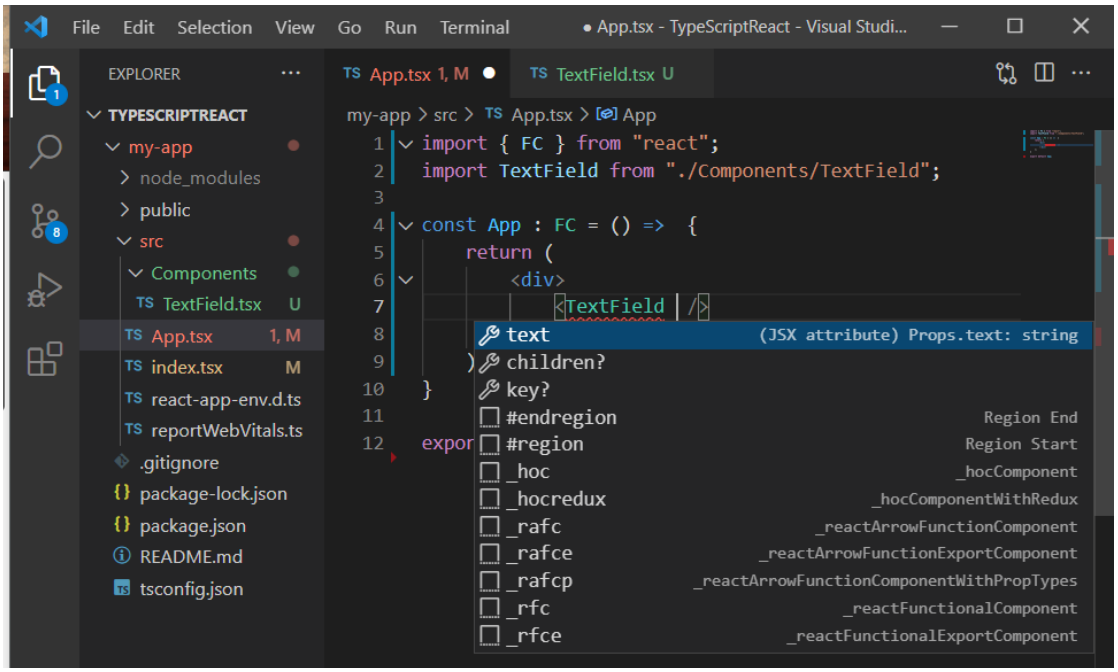
- What that does is that if you try to insert that component without the ‘text’ property, as follows, it will yell at you with a compilation error.
  Ex:

```
import { FC } from "react";
import TextField from "./Components/TextField";

const App : FC = () =>  {
    return (
        <div>
            <TextField />
        </div>
    );
}

export default App;
```

- Another perk is that (at least in vs code) you’ll get intellisense for the components properties!
  Ex:



- The other types you can have are as follows.
  Ex:

```
interface Props {
    text: string;
```

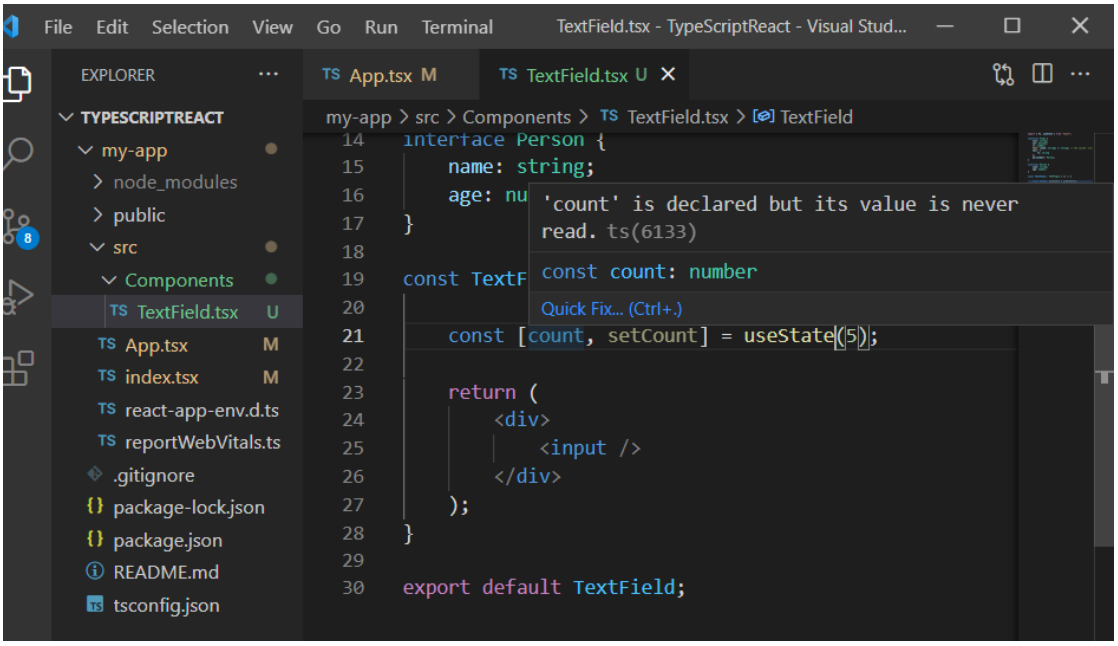
```
    ok: boolean;
    i: number;
    foo: (name: string) => string; // the second 'string' is the
return type
    obj: {
        f1: string
    };
    personObj: Person;
}

interface Person {
    name: string;
    age: number;
}
```

- You can also determine that a property is optional using ?.
  Ex:

```
interface Props {
    text: string;
    ok?: boolean;
    i?: number;
    foo?: (name: string) => string; // the second 'string' is the
return type
    obj?: {
        f1: string
    };
    personObj?: Person;
}
```

- Now let's see what you can do with react hooks and ts.
  First of all, when you define a useState hook for example, the type is inferred from the initial value.
  Ex:



- But if you want to make it so the value can be **also** null, you can explicitly define the type as follows.
  In ts the logical OR operator is |.
  Ex:

```
const [count, setCount] = useState<number | null>(5);

setCount(null);
```

- It's worth mentioning that 'undefined' is a different type than null, so you can't set 'count' to 'undefined', unless of course you declare it can be that too.
  Ex:

```
const [count, setCount] = useState<number | null | undefined>(5);

setCount(undefined);
```

- You can also pass interfaces here.
  Ex:

```
const [person, setPerson] = useState<Person>({name: 'bob', age:
22});
```