



## Section 30: React + TypeScript

Wednesday, October 2, 2024 9:14 AM




# React + TypeScript

## Adding Type Safety To React Apps

- ▶ What & Why?
- ▶ TypeScript Basics
- ▶ Combining React & TypeScript

### What and Why



# TypeScript is a **superset** to JavaScript

TypeScript does not run on the browser. So we need a compiler to compile our TypeScript code to JavaScript code.

#### A Closer Look At Generics

Generic Types ("Generics") can be tricky to wrap your head around.

But indeed, we are working with them all the time - one of the most prominent examples is an array.  
Consider this example array:

1. `let numbers = [1, 2, 3];`

Here, the type is inferred, but if we would assign it explicitly, we could do it like this:

2. `let numbers: number[] = [1, 2, 3];`

`number[]` is the TypeScript notation for saying "this is an array of numbers".

But actually, `number[]` is just syntactic sugar!

The actual type is `Array`. ALL arrays are of the `Array` type.

BUT: Since an array type really only makes sense if we also describe the type of items in the array, `Array` actually is a generic type.

You could also write the above example like this:

3. `let numbers: Array<number> = [1, 2, 3];`

Here we have the angle brackets (`<>`) again! But this time NOT to create our own type (as we did it in the previous lecture) but instead to tell TypeScript which actual type should be used for the "generic type placeholder" (`T` in the previous lecture).

Just as shown in the last lecture, TypeScript would be able to infer this as well - we rely on that when we just write:

4. `let numbers = [1, 2, 3];`

But if we want to explicitly set a type, we could do it like this:

5. `let numbers: Array<number> = [1, 2, 3];`

Of course it can be a bit annoying to write this rather long and clunky type, that's why we have this alternative (syntactic sugar) for arrays:

6. `let numbers: number[] = [1, 2, 3];`

If we take the example from the previous lecture, we could've also set the concrete type for our placeholder `T` explicitly:

7. `const stringArray = insertAtBeginning<string>(['a', 'b', 'c'], 'd');`

So we can not just use the angle brackets to define a generic type but also to USE a generic type and explicitly set the placeholder type that should be used - sometimes this is required if TypeScript is not able to infer the (correct) type. We'll see this later in this course section!

From <https://www.udemy.com/course/react-the-complete-guide-incl-redux/learn/lecture/25890354#overview>

### **Creating a React + TypeScript Project.**

In React+typescript project, we do not need to explicitly compile to convert to js code. It happens behind the scene.

```

1 {
2   "name": "react-ts",
3   "version": "0.1.0",
4   "private": true,
5   "dependencies": {
6     "@testing-library/jest-dom": "^5.11.10",
7     "@testing-library/react": "^11.2.6",
8     "@testing-library/user-event": "^12.8.3",
9     "@types/jest": "^26.0.22",
10    "@types/node": "^12.20.7",
11    "@types/react": "^17.0.3",
12    "@types/react-dom": "^17.0.3",
13    "react": "^17.0.2",
14    "react-dom": "^17.0.2",
15    "react-scripts": "4.0.3",
16    "typescript": "^4.2.3",
17    "web-vitals": "^1.1.1"
18  },

```

In typescript project, we get warning. Which we would not get in js project.

```

1 function Todos(props) {
2   return (
3     <ul>
4       | {}
5     </ul>
6   );
7 }
8
9 export default Todos;

```

(parameter) props: any  
 'props' is declared but its value is never read. ts(6133)  
 Parameter 'props' implicitly has an 'any' type. ts(7006)  
[View Problem](#) (`\F8`) [Quick Fix...](#) (`⌘`)

```

2
3 const Todos: React.FC = (props) => {
4   return (
5     <ul>
6       | {}
7     </ul>
8   );
9 }
10
11 export default Todos;
12

```

That's a type defined in React Package.

```

version: "0.1.0",
"private": true,
"dependencies": {
  "@testing-library/jest-dom": "^5.11.10",
  "@testing-library/react": "^11.2.6",
  "@testing-library/user-event": "^12.8.3",
  "@types/jest": "^26.0.22",
  "@types/node": "^12.20.7",
  "@types/react": "^17.0.3",
  "@types/react-dom": "^17.0.3",
  "react": "^17.0.2",
  "react-dom": "^17.0.2",
  "react-scripts": "4.0.3",
  "typescript": "^4.2.3",
  "web-vitals": "^1.1.1"
}

```

```

type FC<P = {}> = FunctionComponent<P>;

```

Now what **React.FC** does in the end as a type definition is it makes it clear that This here is a functional that acts a functional component.

```

const Todos: React.FC = (props) => {
  return (
    <ul>
      <li>
        {}
      </li>
    </ul>
  );
}

export default Todos;

```

Now, react understands that it is a function that receives a props object as a argument.

Which will be an object that always has a children property.

So let's delete those hard-coded Todos here.  
 And now let's say we get some props here.  
 And now my goal is that the props should have items or Todos prop, where I can then map through all those items to then output the list items.  
 And you can already see here that now I'm getting those red squiggly lines here.  
 And if we hover over that, we see that prompts is declared but it's value is never read.  
 So we're getting a warning that we're not using props and that's a little extra help we get into TypeScript project which we don't necessarily get in a vanilla JavaScript project.  
 We are warned if we have some unnecessary code,

code that we're not using.  
And we also get this warning  
that the parameter props is implicitly of type any.  
And that means that we did not assign a type here.  
We're not making it clear which kind of type disease  
and therefore we're basically not getting  
any TypeScript support.  
And that's why you can configure TypeScript  
such that it warns you if you are implicitly using any.  
If I now explicitly set this to type any, it would be okay.  
I'm now explicitly saying I don't have any more  
specific information about this props argument;  
it is of type any.  
It could be anything and that's then fine.  
But if you don't set this type explicitly,  
you are implicitly assuming  
that you don't know anything about that type  
and that's why TypeScript is warning us here.  
We shouldn't make this implicit assumption.  
By the way you can configure the strictness  
of TypeScript here in the tsconfig.json file.  
But I'll come back to that a little bit later.  
For the moment what we need to do here  
is we need to make it clear  
which kind of props we're getting here,  
what the type of props in this function should be.  
And of course, we can do this with a type assignment here.  
And now we could actually say that props is an object type.  
So we add those curly braces.  
And again, that's not a value here,  
that's a type definition.  
It's an object type  
and we know that props in React will always be an object  
with all the attributes we added on our own element here  
in App.tsx as key value pairs.  
So here we could say  
that we expect to get some Todos or some items  
and then in turn could be an array of strings,  
let's say like this.  
That's how we could say that props is an object  
with the items key,  
which then itself holds an array of strings as a value.  
But that would actually not be 100% correct.  
Props does not just have the key value pairs.  
we added on our component when we use it in JSX.  
Instead, props is an object  
which does have those key value pairs.  
But remember that they're also always as a special prop,  
the children prop  
and we don't even know the type of that yet.  
Now we could find out which type that should be  
and then added like this  
but it would quickly become cumbersome  
if for every component that we're defining  
that is using props,  
we have to always add those built-in props to that object  
and then also our custom props.

And because that's cumbersome,  
and because we have that base prop object  
which we get on every component,  
which for example includes this children prop,  
React gives us a different way here  
or React and TypeScript gives us a different way here.  
We can use such a generic type, which you might remember.  
We can use such a generic type or to be precise,  
a functional component out of the box  
can be turned, can be converted into a generic function.  
Which then simply means that in the end  
our functional component will be configured  
such that we make it clear  
that it will be our React component function  
and that it will have all those base props like children.  
And we can then define explicitly our own props  
like the items prop, for example, here  
that should be combined into the props object.  
And if that's not 100% clear yet,  
I'll show you how it works.  
We can actually write this a little bit differently  
and use this arrow function syntax  
which we used throughout this course all the time.  
So that our Todos constant here  
holds this React component function.  
And now we can assign a type here to this Todos constant.  
And the type here should actually be React  
for which we need to import React  
from React like this, dot FC.  
Now that's a type defined in this React package  
or to be precise by this typesreact package  
but we can just import React from React  
and the rest will be done behind the scenes.  
So this is a type definition here.  
Now, if I hold Cmd and click on this  
or Ctrl and click on this, I see the type definition here.  
And I see that it's a function component  
which is yet another type definition built  
into this React package.  
Now what React.FC in the end does as a type definition  
is it makes it clear that this here is a function  
that acts as a functional component.  
That's what FC stands for functional component.  
And you see that now the red squiggly lines for props  
are gone.  
And if I now try to use props here in my JSX code  
and I add a dot, I see that I now get this auto-completion  
for the children props.  
So now, simply by adding this type annotation  
in TypeScript and my IDE therefore,  
which is very smart about that,  
understands that this is a function  
that receives a props object  
as an argument, as a first argument,  
which will be an object that always has a children property,  
a children field.  
And it understands does it knows this

because of this type assignment here.  
Now I mentioned that generic types would be important  
because this is actually a generic type  
and we can now merge our own type definition  
for it as props object.  
So our own props, this functional component we'll get  
with those built-in based props like the children prop.  
And for this, we add angled brackets or after FC here.  
And then between those angle brackets  
we define our own props.  
Now, when we use angle brackets here,  
we actually use them slightly differently  
than we did before when I introduced you to generic types.  
There we used angle brackets  
to create our own generic function  
that has this generic type parameter  
if you want to call it like this,  
this generic type T here,  
which then could be used inside of the function,  
and which in the example I gave you then was inferred  
by TypeScript when we used that generic function.  
Here we're using it slightly differently.  
Here, React.FC already is a generic type.  
It describes a type defined by the React package  
that is actually generic.  
So that's type internally is also defined  
with those angle brackets.  
And here, when I also add angled brackets in our code,  
I'm not setting up a new generic type  
with some placeholder type T,  
but I'm plugging in a concrete value  
for that internally used generic type  
before that type T defined by that React.FC type.  
And I'm doing that here because here  
we can't let TypeScript infer that generic type  
as it did it in the example I showed you before.  
We can't let TypeScript infer the type here  
because here we're not calling some generic function  
with some parameters where the values then could be used  
for the inference,  
but instead we're defining a function  
and we want to let TypeScript know  
how it should then treat this function internally  
that it should get some props to find by us  
and merge those with some base props like the children prop  
which all functional components have.  
And what you see here, therefore, is the other side  
of generic types that you use a generic type  
and you explicitly set the concrete type that should be used  
for this usage of this generic FC type.  
Here, I'm saying this FC type is generic  
and the concrete value I'm plugging in now  
is my own props object where I describe my own props  
for this specific functional component.  
And it's generic because different functional components  
have different props definitions.  
And that's how we do that with React and TypeScript here,

how we can also use this generic types feature.  
Now this angle brackets index  
is the generic type syntax here,  
and by adding it here, we actually unlock a feature  
built into this FC type, which we're using here,  
which we'll merge, whichever object type we're defining here  
with that base object type, with the children property.  
And here we could then say  
that we want to have items which has an array of strings.  
And what this does now is that here, if I, again  
type props dot, it still knows  
that there will be a children property  
at least that there could be a children property.  
But that we now also have this items property  
which will be a string array.  
And it knows that because we now merge  
our own prop object definition  
with that base prop object definition.  
And that might all sound super complex  
but it's just sounding like this because I'm explaining  
that behind the scenes stuff here.  
In the end, using it as super simple,  
you build functional components with React and TypeScript  
by using this React.FC type here  
on your functional component constant here,  
and then you use those angle brackets.  
And between those angle brackets,  
you define your own props  
your own prop object type  
if your component gets some custom props  
and then in the component, you can use those custom props.  
And here we can then use props items, use map,  
and we get auto-completion here for this array method  
because TypeScript knows that items will be an array  
because here we defined that it will be an array.  
And we can then map all our items and turn every item  
into a list item here where I output item  
and where I set the key equal to item.  
So for the moment my key will just be the string  
inside of this items array.  
And now I'll reformat that, and that's not as finished  
Todos component with proper type notation.  
Now, the great thing about using TypeScript with React  
is now not just that this is now more descriptive  
and we get the auto-completion here  
when we work inside of this component,  
though that's already pretty good.  
But then now you also see we have an error in App.tsx  
Because of our type annotations,  
React now understands  
or this project set up to be precise,  
now understands that we're using this Todos component  
in an incorrect way.  
Because I making it clear here,  
in this type definition of the Todos component  
that this component will have an items prop  
and this is not an optional prop.



We could make it optional by adding a question mark here but then here, we need to handle the case that we don't necessarily have items. But here it's also not optional at all, and therefore we have to add it here. And that's another strong reason for using TypeScript. We can now really describe the shape of our components and which props they need and therefore using our components incorrectly. And for example, not passing in all the props that component needs is pretty much impossible because we get errors like this directly in the IDE now. So here I can now add my items and for the moment, just add some dummy Todos here, like learn React and learn TypeScript, we format this, and now all errors are gone. And we now see that output again, but now thanks to our props and those props now work properly. Thanks to our type annotations. And I did spend a lot of times talking about this type annotation, but I hope it's clear how it works and even more importantly, it's clear how you use it. And as you see using it is super simple. It's this type annotation, which you want to add to every custom component you're building. And then if you are using your own props, you add this generic type and you describe your props between those angle brackets.

From <<https://www.udemy.com/course/react-the-complete-guide-incl-redux/learn/lecture/25890212#overview>>

## **Working with refs & useRef**

```

1 import { useRef } from 'react';
2
3 const NewTodo = () => {
4   const todoTextInputRef = useRef();
5
6   const submitHandler = (event: React.FormEvent) => {
7     event.preventDefault();
8
9   };
10
11   return (
12     <form onSubmit={submitHandler}>
13       <label htmlFor='text'>Todo text</label>
14       <input type='text' id='text' ref={todoTextInputRef} />
15       <button>Add Todo</button>
16     </form>
17   );
18 }

```

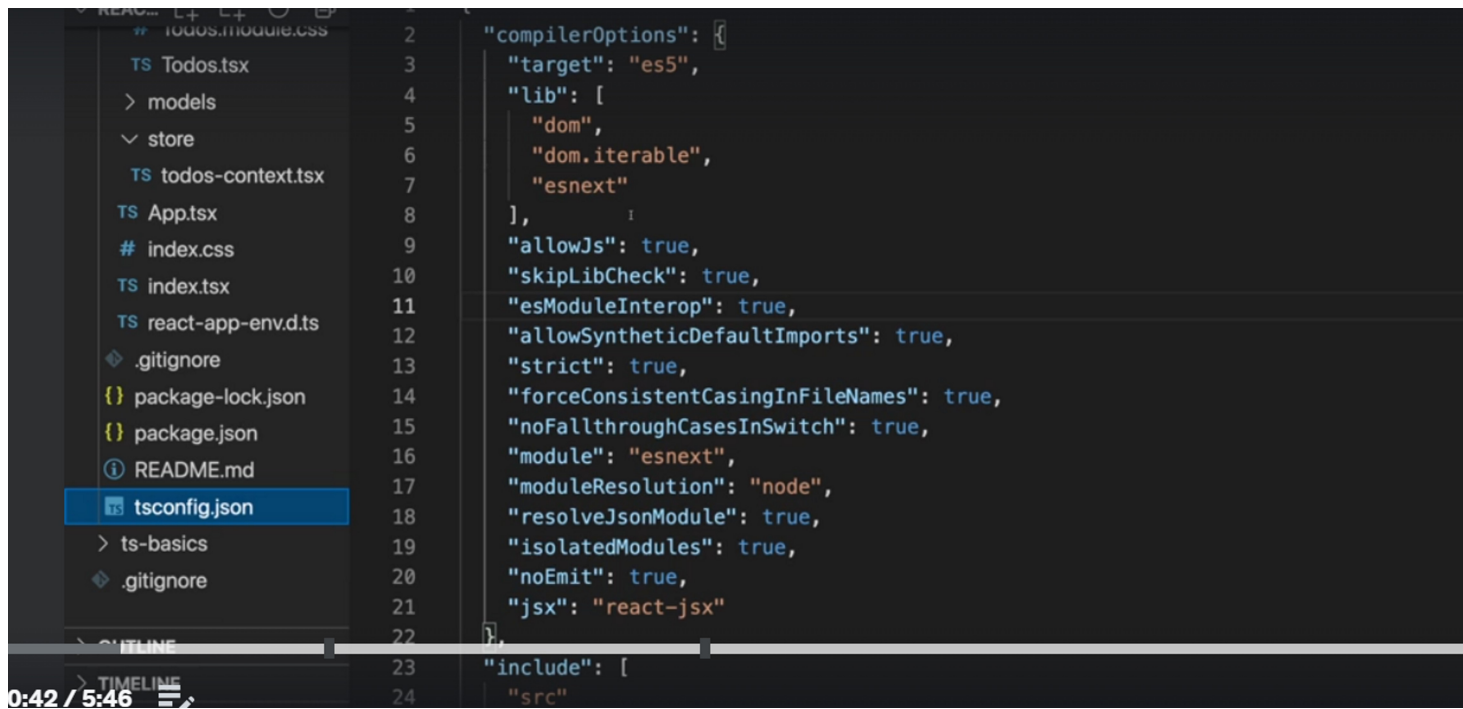
In vanilla JavaScript, this code works because we don't have the concept of extra type annotations. But when working with TypeScript, TypeScript actually wants to know a bit more about our ref.

When we create it here, TypeScript actually has no idea that we eventually will connect it to a input element. That's only clear down there. But since it's not clear in this line, we could use this ref in any possible way. We can connect it to any HTML element. And that's, in the end, the kind of error we get here.

We're passing a ref as a ref to this input that is not clearly focused on this kind of HTML element. This could also be a ref that wants a button. So we have to be more explicit about the kind of data we will store in this ref eventually. And we do this again with a generic type.

That's why useRef actually is a generic type out of the box.

## Exploring tsconfig.json



```
2  "compilerOptions": {
3    "target": "es5",
4    "lib": [
5      "dom",
6      "dom.iterable",
7      "esnext"
8    ],
9    "allowJs": true,
10   "skipLibCheck": true,
11   "esModuleInterop": true,
12   "allowSyntheticDefaultImports": true,
13   "strict": true,
14   "forceConsistentCasingInFileNames": true,
15   "noFallthroughCasesInSwitch": true,
16   "module": "esnext",
17   "moduleResolution": "node",
18   "resolveJsonModule": true,
19   "isolatedModules": true,
20   "noEmit": true,
21   "jsx": "react-jsx"
22 },
23 "include": [
24   "src"
```

0:42 / 5:46

```
compilerOptions: {  
  "target": "es5",  
  "lib": [  

```

This in the end controls the target javascript version to which your code will be transformed. And in this case it's ES5 javascript. And in this case, that's es5 JavaScript, which is quite an old version which of course means that you should generally have a broad browser support.

Though I will say, that depending on your project setup, this target option might not be the only thing that influences the output. There also can be project setups where your TypeScript code might be compiled to JavaScript, and then you might have yet another compilation step.

For example, with other tools like Babel, that take that output JavaScript code and transform it even more. So that's not necessarily the last step in line.

Now the DOM lib which is added here, for example, ensures that some default DOM types are understood by TypeScript. And a great example here can be found in the NewTodo.tsx file. This `HTMLInputElement` type. I mentioned that this is a built-in type and it's actually such a built-in type that it's unlocked because we have that DOM lib added here.

```
"lib": [  
  "dom",  
  "dom.iterable",  
  "esnext"  
]
```

```

4 import classes from './NewTodo.module.css';
5
6 const NewTodo: React.FC = () => {
7   const todosCtx = useContext(TodosContext);
8
9   const todoTextInputRef = useRef<HTMLInputElement>(null);
10
11   const submitHandler = (event: React.FormEvent) => {
12     event.preventDefault();
13
14     const enteredText = todoTextInputRef.current!.value;
15
16     if (enteredText.trim().length === 0) {
17       // throw an error

```

If you wanna mix and match with javascript and typescript file.

```

"allowJs": true,

```

We can't have implicitly any values.

```

"strict": true,

```

```

6 const NewTodo: React.FC = () => {
7   const todosCtx = useCon (parameter) event: any
8
9   const todoTextInputRef Parameter 'event' implicitly has an 'any' type. ts(7006)
10
11   const submitHandler = (event) => {
12     event.preventDefault();
13
14     const enteredText = todoTextInputRef.current!.value;
15
16     if (enteredText.trim().length === 0) {
17       // throw an error
18       return;
19     }

```