Specifying a type for function component props

In this lesson, we will learn how to create strongly-typed props for function components.

WE'LL COVER THE FOLLOWING

One of the most beneficial parts of using TypeScript with React is the ability to have strongly-typed component props. This allows developers using the component to quickly understand what they can pass and helps them avoid making mistakes.

Specifying a type for function component props

In this section, we will create a React `Hello` component that has a simple prop for who to say hello to. We will explore two different ways of strongly-typing the component props.

We are going to use CodeSandbox to explore this topic. So, let's open the CodeSandbox project by clicking the following link:

CodeSandbox exercise

Can you spot a problem with the `Hello` component at the moment?

Show Answer

Using type annotations for `props`

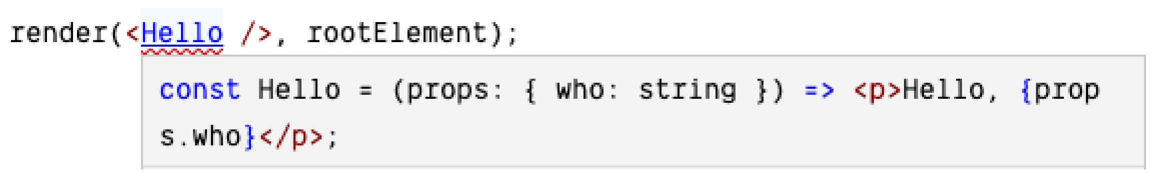
React function components are just functions that take in a `props` parameter and return JSX. We have already learned how to strongly-type function parameters, so, let's use this knowledge:

- Let's start by adding a type annotation for the `props`:

```
const Hello = (props: { who: string }) => (
  <p>Hello, {props.who}</p>
);
```

We have defined a `who` prop that is a `string` and included this in the hello message.

Notice that a type error is now raised where `Hello` is consumed:



```
render(<Hello />, rootElement);
const Hello = (props: { who: string }) => <p>Hello, {prop
s.who}</p>;
```

Congratulations, we have just made `Hello`'s props strongly-typed!

We will continue to use this CodeSandbox project throughout this lesson, so keep it open.

- Resolve the type error by passing `"Bob"` into `Hello`:

```
render(<Hello who="Bob" />, rootElement);
```

- We can shorten the `Hello` component slightly by destructuring the `props` parameter:

```
const Hello = ({ who }: { who: string }) => (
  <p>Hello, {who}</p>
);
```

- We can also use an interface or a type alias for the props:

```
type Props = { who: string }
const Hello = ({ who }: Props) => <p>Hello, {who}</p>
```

This is arguably more readable when the component has more than a couple of props. This is also better if the type is referenced elsewhere in the app.

- At the moment, our `Hello` component is an arrow function. Let’s refactor it to be a regular function:

```
function Hello({ who }: Props) {
 return <p>Hello, {who}</p>;
}
```

So, a React components props can be strongly-typed with a type annotation, just like with any regular function. We can use this approach in regular functions as well as arrow functions.

Using the `React.FC` type

There is a standard React type, `FC`, that we can use as an alternative approach to strongly-typing React component props on arrow function components. “FC” stands for *Function Component*, and it aliases a type called `FunctionComponent`.

- Let’s change the `Hello` component back to an arrow function and use the `FC` type:

```
const Hello: React.FC<Props> = ({ who }) => (
  <p>Hello, {who}</p>
);
```

The `FC` type is used on the variable assigned to the arrow function. It is a generic type that we pass the component’s props’ type into.

Are there any benefits of using the `FC` type approach? Well, here are some minor benefits:

- `FC` provides some type safety on `defaultProps`, which we will learn about later in this chapter. However, `defaultProps` may be removed from React in a future release.
- `FC` includes the `children` prop type, so you don’t have to define it explicitly. However, it is arguably better to explicitly define this so that consumers know for sure whether this is available.

So, the `FC` type doesn’t provide any significant benefits.

Wrap up

Well done! We can now create strongly-typed props for function components by using a type annotation for the `props` parameter, or alternatively, use the `FC` generic type.

What if we want to specify a prop as optional? We’ll learn how to do this in the next lesson. Keep the CodeSandbox project safe as we’ll need it in the next lesson.