Using TypeScript

TypeScript is a popular way to add type definitions to JavaScript codebases. Out of the box, TypeScript supports JSX and you can get full React Web support by adding `@types/react` and `@types/react-dom` to your project.

You will learn

- TypeScript with React Components
- Examples of typing with Hooks
- Common types from `@types/react`
- Further learning locations

Installation

All production-grade React frameworks offer support for using TypeScript. Follow the framework specific guide for installation:

- Next.js
- Remix
- Gatsby
- Expo

Adding TypeScript to an existing React project

To install the latest version of React's type definitions:

Terminal Copy

```
npm install @types/react @types/react-dom
```

The following compiler options need to be set in your `tsconfig.json`:

1. `dom` must be included in `lib` (Note: If no `lib` option is specified, `dom` is included by default).
2. `jsx` must be set to one of the valid options. `preserve` should suffice for most applications. If you're publishing a library, consult the `jsx` documentation on what value to choose.

TypeScript with React Components

> **Note**
>
> Every file containing JSX must use the `.tsx` file extension. This is a TypeScript-specific extension that tells TypeScript that this file contains JSX.

Writing TypeScript with React is very similar to writing JavaScript with React. The key difference when working with a component is that you can provide types for your component's props. These types can be used for correctness checking and providing inline documentation in editors.

Taking the `MyButton` component from the Quick Start guide, we can add a type describing the `title` for the button:

App.tsx

Reset Fork TypeScript Playground

```tsx
function MyButton({ title }: { title: string }) {
  return (
    <button>{title}</button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton title="I'm a button" />
    </div>
```

Unable to establish connection with the sandpack bundler. Make sure you are online or try again later. If the problem persists, please report it via email or submit an issue on GitHub.

```
  );
}

```

Note

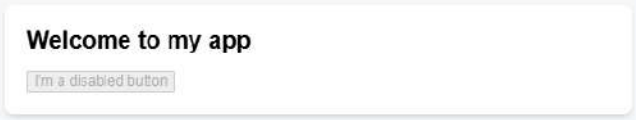
These sandboxes can handle TypeScript code, but they do not run the type-checker. This means you can amend the TypeScript sandboxes to learn, but you won't get any type errors or warnings. To get type-checking, you can use the TypeScript Playground or use a more fully-featured online sandbox.

This inline syntax is the simplest way to provide types for a component, though once you start to have a few fields to describe it can become unwieldy. Instead, you can use an `interface` or `type` to describe the component's props:

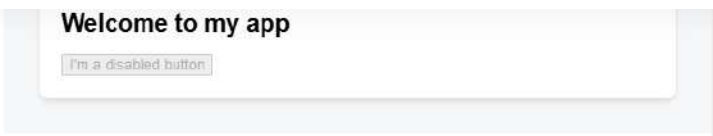
App.tsx

Reset Fork TypeScript Playground

```tsx
interface MyButtonProps {
  /** The text to display inside the button */
  title: string;
  /** Whether the button can be interacted with */
  disabled: boolean;
}

function MyButton({ title, disabled }: MyButtonProps) {
  return (
    <button disabled={disabled}>{title}</button>
  );
}

export default function MyApp() {
  return (
    <div>
      <h1>Welcome to my app</h1>
      <MyButton title="I'm a disabled button" disabled={true}/>
```



```
    </div>
  );
}

```



Show less

The type describing your component's props can be as simple or as complex as you need, though they should be an object type described with either a `type` or `interface`. You can learn about how TypeScript describes objects in Object Types but you may also be interested in using Union Types to describe a prop that can be one of a few different types and the Creating Types from Types guide for more advanced use cases.

Example Hooks

The type definitions from `@types/react` include types for the built-in Hooks, so you can use them in your components without any additional setup. They are built to take into account the code you write in your component, so you will get inferred types a lot of the time and ideally do not need to handle the minutiae of providing the types.

However, we can look at a few examples of how to provide types for Hooks.

`useState`

The `useState` Hook will re-use the value passed in as the initial state to determine what the type of the value should be. For example:

```
// Infer the type as "boolean"
const [enabled, setEnabled] = useState(false);
```

This will assign the type of `boolean` to `enabled`, and `setEnabled` will be a function accepting either a `boolean` argument, or a function that returns a `boolean`. If you want to explicitly provide a type for the state, you can do so by providing a type argument to the `useState` call:

```tsx
// Explicitly set the type to "boolean"
const [enabled, setEnabled] = useState<boolean>(false);
```

This isn't very useful in this case, but a common case where you may want to provide a type is when you have a union type. For example, `status` here can be one of a few different strings:

```tsx
type Status = "idle" | "loading" | "success" | "error";

const [status, setStatus] = useState<Status>("idle");
```

Or, as recommended in Principles for structuring state, you can group related state as an object and describe the different possibilities via object types:

```tsx
type RequestState =
  | { status: 'idle' }
  | { status: 'loading' }
  | { status: 'success', data: any }
  | { status: 'error', error: Error };

const [requestState, setRequestState] = useState<RequestState>({ status: 'idle' });
```

useReducer

The `useReducer` Hook is a more complex Hook that takes a reducer function and an initial state. The types for the reducer function are inferred from the initial state. You can optionally provide a type argument to the `useReducer` call to provide a type for the state, but it is often better to set the type on the initial state instead:

```
    count: number
};

type CounterAction =
  | { type: "reset" }
  | { type: "setCount"; value: State["count"] }

const initialState: State = { count: 0 };

function stateReducer(state: State, action: CounterAction): State {
  switch (action.type) {
    case "reset":
      return initialState;
    case "setCount":
      return { ...state, count: action.value };
    default:
      throw new Error("Unknown action");
  }
}

export default function App() {
  const [state, dispatch] = useReducer(stateReducer, initialState);

  const addFive = () => dispatch({ type: "setCount", value: state.count
  const reset = () => dispatch({ type: "reset" });

  return (
    <div>
      <h1>Welcome to my counter</h1>

      <p>Count: {state.count}</p>
      <button onClick={addFive}>Add 5</button>
      <button onClick={reset}>Reset</button>
    </div>
  );
}
```

Show less

We are using TypeScript in a few key places:

- `interface State` describes the shape of the reducer's state.
- `type CounterAction` describes the different actions which can be dispatched to the reducer.
- `const initialState: State` provides a type for the initial state, and also the type which is used by `useReducer` by default.
- `stateReducer(state: State, action: CounterAction): State` sets the types for the reducer function's arguments and return value.

A more explicit alternative to setting the type on `initialState` is to provide a type argument to `useReducer`:

```
import { stateReducer, State } from './your-reducer-implementation';

const initialState = { count: 0 };

export default function App() {
  const [state, dispatch] = useReducer<State>(stateReducer, initialState);
}
```

`useContext`

The `useContext` Hook is a technique for passing data down the component tree without having to pass props through components. It is used by creating a provider component and often by creating a Hook to consume the value in a child component.

The type of the value provided by the context is inferred from the value passed to the `createContext` call:

```tsx
const useGetTheme = () => useContext(ThemeContext);

export default function MyApp() {
  const [theme, setTheme] = useState<Theme>('light');

  return (
    <ThemeContext.Provider value={theme}>
      <MyComponent />
    </ThemeContext.Provider>
  )
}

function MyComponent() {
  const theme = useGetTheme();

  return (
    <div>
      <p>Current theme: {theme}</p>
    </div>
  )
}

```

Show less

This technique works when you have a default value which makes sense - but there are occasionally cases when you do not, and in those cases `null` can feel reasonable as a default value. However, to allow the type-system to understand your code, you need to explicitly set `ContextShape | null` on the `createContext`.

This causes the issue that you need to eliminate the `| null` in the type for context consumers. Our recommendation is to have the Hook do a runtime check for it's existence and throw an error when not present:

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

// This is a simpler example, but you can imagine a more complex object here
type ComplexObject = {
  kind: string
};
```

```tsx
// The context is created with `| null` in the type, to accurately reflect the default value.
const Context = createContext<ComplexObject | null>(null);

// The `| null` will be removed via the check in the Hook.
const useGetComplexObject = () => {
  const object = useContext(Context);
  if (!object) { throw new Error("useGetComplexObject must be used within a Provider") }
  return object;
}

export default function MyApp() {
  const object = useMemo(() => ({ kind: "complex" }), []);

  return (
    <Context.Provider value={object}>
      <MyComponent />
    </Context.Provider>
  )
}

function MyComponent() {
  const object = useGetComplexObject();

  return (
    <div>
      <p>Current object: {object.kind}</p>
    </div>
  )
}
```

`useMemo`

The `useMemo` Hooks will create/re-access a memorized value from a function call, re-running the function only when dependencies passed as the 2nd parameter are changed. The result of calling the Hook is inferred from the return value from the function in the first parameter. You can be more explicit by providing a type argument to the Hook.

```
// The type of visibleTodos is inferred from the return value of filterTodos
const visibleTodos = useMemo(() => filterTodos(todos, tab), [todos, tab]);
```

useCallback

The `useCallback` provide a stable reference to a function as long as the dependencies passed into the second parameter are the same. Like `useMemo`, the function's type is inferred from the return value of the function in the first parameter, and you can be more explicit by providing a type argument to the Hook.

```
const handleClick = useCallback(() => {
  // ...
}, [todos]);
```

When working in TypeScript strict mode `useCallback` requires adding types for the parameters in your callback. This is because the type of the callback is inferred from the return value of the function, and without parameters the type cannot be fully understood.

Depending on your code-style preferences, you could use the `*EventHandler` functions from the React types to provide the type for the event handler at the same time as defining the callback:

```
import { useState, useCallback } from 'react';

export default function Form() {
  const [value, setValue] = useState("Change me");

  const handleChange = useCallback<React.ChangeEventHandler<HTMLInputElement>>((event) => {
    setValue(event.currentTarget.value);
  }, [setValue])

  return (
    <>
      <input value={value} onChange={handleChange} />
      <p>Value: {value}</p>
    </>
```

```
  );
}
```

Useful Types

There is quite an expansive set of types which come from the `@types/react` package, it is worth a read when you feel comfortable with how React and TypeScript interact. You can find them in React's folder in DefinitelyTyped. We will cover a few of the more common types here.

DOM Events

When working with DOM events in React, the type of the event can often be inferred from the event handler. However, when you want to extract a function to be passed to an event handler, you will need to explicitly set the type of the event.

App.tsx

Reset Fork TypeScript Playground

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

export default function Form() {
  const [value, setValue] = useState("Change me");

  function handleChange(event: React.ChangeEvent<HTMLInputElement>) {
    setValue(event.currentTarget.value);
  }

  return (
    <>
      <input value={value} onChange={handleChange} />
      <p>Value: {value}</p>
    </>
  );
}

```

Show less

There are many types of events provided in the React types - the full list can be found here which is based on the most popular events from the DOM.

When determining the type you are looking for you can first look at the hover information for the event handler you are using, which will show the type of the event.

If you need to use an event that is not included in this list, you can use the `React.SyntheticEvent` type, which is the base type for all events.

Children

There are two common paths to describing the children of a component. The first is to use the `React.ReactNode` type, which is a union of all the possible types that can be passed as children in JSX:

```
interface ModalRendererProps {
  title: string;
  children: React.ReactNode;
}
```

This is a very broad definition of children. The second is to use the `React.ReactElement` type, which is only JSX elements and not JavaScript primitives like strings or numbers:

```
interface ModalRendererProps {
  title: string;
  children: React.ReactElement;
}
```

Note, that you cannot use TypeScript to describe that the children are a certain type of JSX elements, so you cannot use the type-system to describe a component which only accepts `<li>` children.

You can see an example of both `React.ReactNode` and `React.ReactElement` with the type-checker in this TypeScript playground.

Style Props

When using inline styles in React, you can use `React.CSSProperties` to describe the object passed to the `style` prop. This type is a union of all the possible CSS properties, and is a good way to ensure you are passing valid CSS properties to the `style` prop, and to get auto-complete in your editor.

```
interface MyComponentProps {
  style: React.CSSProperties;
}
```

Further learning

This guide has covered the basics of using TypeScript with React, but there is a lot more to learn. Individual API pages on the docs may contain more in-depth documentation on how to use them with TypeScript.

We recommend the following resources:

- The TypeScript handbook is the official documentation for TypeScript, and covers most key language features.
- The TypeScript release notes cover new features in depth.
- React TypeScript Cheatsheet is a community-maintained cheatsheet for using TypeScript with React, covering a lot of useful edge cases and providing more breadth than this document.
- TypeScript Community Discord is a great place to ask questions and get help with TypeScript and React issues.