typescript-cheatsheets / **react**

Cheatsheets for experienced React developers getting started with TypeScript

react-typescript-cheatsheet.netlify.app/

MIT License

17.6k stars 1.2k forks

Star Watch

Code Issues 13 Pull requests 1 Actions Projects Wiki Security Insights

main

dance2die ... 23 hours ago

View code

README.md

React+TypeScript Cheatsheets

Cheatsheets for experienced React developers getting started with TypeScript

Web docs | 中文翻译 | **Español** | Contribute! | Ask!

This repo is maintained by @swyx, @ferdaber, @eps1lon, @jsjoeio and @arvindcheenu, we're so happy you want to try out TypeScript with React! If you see anything wrong or missing, please file an issue!

All React + TypeScript Cheatsheets

- **The Basic Cheatsheet** (`/README.md`) is focused on helping React devs just start using TS in React **apps**

    - Focus on opinionated best practices, copy+pastable examples.
    - Explains some basic TS types usage and setup along the way.
    - Answers the most Frequently Asked Questions.
    - Does not cover generic type logic in detail. Instead we prefer to teach simple troubleshooting techniques for newbies.
    - The goal is to get effective with TS without learning *too much* TS.
- **The Advanced Cheatsheet** (`/ADVANCED.md`) helps show and explain advanced usage of generic types for people writing reusable type utilities/functions/render prop/higher order components and TS+React **libraries**.
    - It also has miscellaneous tips and tricks for pro users.
    - Advice for contributing to DefinitelyTyped.
    - The goal is to take *full advantage* of TypeScript.
- **The Migrating Cheatsheet** (`/MIGRATING.md`) helps collate advice for incrementally migrating large codebases from JS or Flow, **from people who have done it**.
    - We do not try to convince people to switch, only to help people who have already decided.
    - ⚠ This is a new cheatsheet, all assistance is welcome.
- **The HOC Cheatsheet** (`/HOC.md`) specifically teaches people to write HOCs with examples.
    - Familiarity with Generics is necessary.
    - ⚠ This is the newest cheatsheet, all assistance is welcome.

Basic Cheatsheet Table of Contents

▶ **Expand Table of Contents**

Section 1: Setup

Prerequisites

1. good understanding of React
2. familiarity with TypeScript Types (2ality's guide is helpful. If you're an absolute beginner in TypeScript, check out chibicode's tutorial.)
3. having read the TypeScript section in the official React docs.
4. having read the React section of the new TypeScript playground (optional: also step through the 40+ examples under the playground's Examples section)

This guide will always assume you are starting with the latest TypeScript version. Notes for older versions will be in expandable `<details>` tags.

React + TypeScript Starter Kits

1. Create React App v2.1+ with TypeScript: `npx create-react-app my-app --template typescript`

- We used to recommend `create-react-app-typescript` but it is now deprecated. see migration instructions

2. Basarat's guide for **manual setup** of React + TypeScript + Webpack + Babel

- In particular, make sure that you have `@types/react` and `@types/react-dom` installed (Read more about the DefinitelyTyped project if you are unfamiliar)
- There are also many React + TypeScript boilerplates, please see our Resources list below.

Import React

```
import * as React from "react";
import * as ReactDOM from "react-dom";
```

In TypeScript 2.7+, you can run TypeScript with `--allowSyntheticDefaultImports` (or add `"allowSyntheticDefaultImports": true` to tsconfig) to import like in regular jsx:

```
import React from "react";
import ReactDOM from "react-dom";
```

▶ Explanation

Section 2: Getting Started

Function Components

These can be written as normal functions that take a `props` argument and return a JSX element.

```
type AppProps = { message: string }; /* could also use interface */
const App = ({ message }: AppProps) => <div>{message}</div>;
```

▶ **What about `React.FC`/`React.FunctionComponent`?**

▶ **Minor Pitfalls**

Hooks

Hooks are supported in `@types/react` from v16.8 up.

useState

Type inference works very well most of the time:

```
const [val, toggle] = React.useState(false); // `val` is inferred to be a boolean, `t
```

See also the Using Inferred Types section if you need to use a complex type that you've relied on inference for.

However, many hooks are initialized with null-ish default values, and you may wonder how to provide types. Explicitly declare the type, and use a union type:

```
const [user, setUser] = React.useState<IUser>();

// later...
setUser(newUser);
```

useRef

When using `useRef`, you have two options when creating a ref container that does not have an initial value:

```
const ref1 = useRef<HTMLElement>(null!);
const ref2 = useRef<HTMLElement | null>(null);
```

The first option will make `ref1.current` read-only, and is intended to be passed in to built-in `ref` attributes that React will manage (because React handles setting the `current` value for you).

▶ What is the `!` at the end of `null!`?

The second option will make `ref2.current` mutable, and is intended for "instance variables" that you manage yourself.

useEffect

When using `useEffect`, take care not to return anything other than a function or `undefined`, otherwise both TypeScript and React will yell at you. This can be subtle when using arrow functions:

```
function DelayedEffect(props: { timerMs: number }) {
  const { timerMs } = props;
  // bad! setTimeout implicitly returns a number because the arrow function body isn'
  useEffect(
    () =>
      setTimeout(() => {
        /* do stuff */
      }, timerMs),
    [timerMs]
  );
  return null;
}
```

useRef

```
function TextInputWithFocusButton() {
  // initialise with null, but tell TypeScript we are looking for an HTMLInputElement
  const inputEl = React.useRef<HTMLInputElement>(null);
  const onButtonClick = () => {
    // strict null checks need us to check if inputEl and current exist.
    // but once current exists, it is of type HTMLInputElement, thus it
    // has the method focus! ✅
    if (inputEl && inputEl.current) {
      inputEl.current.focus();
    }
  };
  return (
    <>
      {/* in addition, inputEl only can be used with input elements. Yay! */}
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

View in the TypeScript Playground

example from Stefan Baumgartner

useReducer

You can use Discriminated Unions for reducer actions. Don't forget to define the return type of reducer, otherwise TypeScript will infer it.

```
type AppState = {};
type Action =
  | { type: "SET_ONE"; payload: string } // typescript union types allow for leading
  | { type: "SET_TWO"; payload: number };
```

```
export function reducer(state: AppState, action: Action): AppState {
  switch (action.type) {
    case "SET_ONE":
      return {
        ...state,
        one: action.payload, // `payload` is string
      };
    case "SET_TWO":
      return {
        ...state,
        two: action.payload, // `payload` is number
      };
    default:
      return state;
  }
}
```

Setting the type for `dispatch` function:

```
import { Dispatch } from "react";
import { Action } from "./reducer";

const handleClick = (dispatch: Dispatch<Action>) => {
  dispatch({ type: "SET_ONE", payload: "some string" });
};
```

View in the TypeScript Playground

▶ **Usage with `Reducer` from `redux`**

Custom Hooks

If you are returning an array in your Custom Hook, you will want to avoid type inference as TypeScript will infer a union type (when you actually want different types in each position of the array). Instead, use TS 3.4 const assertions:

```
export function useLoading() {
  const [isLoading, setState] = React.useState(false);
  const load = (aPromise: Promise<any>) => {
    setState(true);
    return aPromise.finally(() => setState(false));
  };
  return [isLoading, load] as const; // infers [boolean, typeof load] instead of (boo
}
```

View in the TypeScript Playground

This way, when you destructure you actually get the right types based on destructure position.

▶ **Alternative: Asserting a tuple return type**

Note that the React team recommends that custom hooks that return more than two values should use proper objects instead of tuples, however.

More Hooks + TypeScript reading:

- https://medium.com/@jrwebdev/react-hooks-in-typescript-88fce7001d0d
- https://fettblog.eu/typescript-react/hooks/#useref

If you are writing a React Hooks library, don't forget that you should also expose your types for users to use.

Example React Hooks + TypeScript Libraries:

- https://github.com/mweststrate/use-st8
- https://github.com/palmerhq/the-platform
- https://github.com/sw-yx/hooks

Something to add? File an issue.

Class Components

Within TypeScript, `React.Component` is a generic type (aka `React.Component<PropType, StateType>`), so you want to provide it with (optional) prop and state type parameters:

```
type MyProps = {
  // using `interface` is also ok
  message: string;
};
type MyState = {
  count: number; // like this
};
class App extends React.Component<MyProps, MyState> {
  state: MyState = {
    // optional second annotation for better type inference
    count: 0,
  };
  render() {
    return (
      <div>
        {this.props.message} {this.state.count}
      </div>
    );
  }
}
```

View in the TypeScript Playground

Don't forget that you can export/import/extend these types/interfaces for reuse.

▶ **Why annotate `state` twice?**

▶ **No need for `readonly`**

Class Methods: Do it like normal, but just remember any arguments for your functions also need to be typed:

```
class App extends React.Component<{ message: string }, { count: number }> {
  state = { count: 0 };
  render() {
    return (
      <div onClick={() => this.increment(1)}>
        {this.props.message} {this.state.count}
      </div>
    );
  }
  increment = (amt: number) => {
    // like this
    this.setState((state) => ({
      count: state.count + amt,
    }));
  };
}
```

View in the TypeScript Playground

Class Properties: If you need to declare class properties for later use, just declare it like `state`, but without assignment:

```
class App extends React.Component<{
  message: string;
}> {
  pointer: number; // like this
  componentDidMount() {
    this.pointer = 3;
  }
  render() {
    return (
      <div>
        {this.props.message} and {this.pointer}
      </div>
    );
  }
}
```

View in the TypeScript Playground

Something to add? File an issue.

Typing defaultProps

For TypeScript 3.0+, type inference should work, although some edge cases are still problematic. Just type your props like normal, except don't use `React.FC`.

```
// ////////////////
// function components
// ////////////////
type GreetProps = { age: number } & typeof defaultProps;
const defaultProps = {
  age: 21,
};

const Greet = (props: GreetProps) => {
  /*...*/
};
Greet.defaultProps = defaultProps;
```

For **Class components**, there are a couple ways to do it(including using the `Pick` utility type) but the recommendation is to "reverse" the props definition:

```
type GreetProps = typeof Greet.defaultProps & {
  age: number;
};

class Greet extends React.Component<GreetProps> {
  static defaultProps = {
    age: 21,
  };
  /*...*/
}

// Type-checks! No type assertions needed!
let el = <Greet age={3} />;
```

▶ An alternative approach

▶ Why does React.FC break defaultProps?

▶ TypeScript 2.9 and earlier

Something to add? File an issue.

Types or Interfaces?

`interface`s are different from `type`s in TypeScript, but they can be used for very similar things as far as common React uses cases are concerned. Here's a helpful rule of thumb:

- always use `interface` for public API's definition when authoring a library or 3rd party ambient type definitions, as this allows a consumer to extend them via *declaration merging* if some definitions are missing.
- consider using `type` for your React Component Props and State, for consistency and because it is more constrained.

You can read more about the reasoning behind this rule of thumb in Interface vs Type alias in TypeScript 2.7.

Types are useful for union types (e.g. `type MyType = TypeA | TypeB`) whereas Interfaces are better for declaring dictionary shapes and then `implementing` or `extending` them.

▶ **Useful table for Types vs Interfaces**

Something to add? File an issue.

Basic Prop Types Examples

```tsx
type AppProps = {
  message: string;
  count: number;
  disabled: boolean;
  /** array of a type! */
  names: string[];
  /** string literals to specify exact string values, with a union type to join them
  status: "waiting" | "success";
  /** any object as long as you dont use its properties (not common) */
  obj: object;
  obj2: {}; // almost the same as `object`, exactly the same as `Object`
  /** an object with defined properties (preferred) */
  obj3: {
    id: string;
    title: string;
  };
  /** array of objects! (common) */
  objArr: {
    id: string;
    title: string;
  }[];
  /** any function as long as you don't invoke it (not recommended) */
  onSomething: Function;
  /** function that doesn't take or return anything (VERY COMMON) */
  onClick: () => void;
  /** function with named prop (VERY COMMON) */
  onChange: (id: number) => void;
```

```
  /** alternative function type syntax that takes an event (VERY COMMON) */
  onClick(event: React.MouseEvent<HTMLButtonElement>): void;
  /** an optional prop (VERY COMMON!) */
  optional?: OptionalType;
};
```

Notice we have used the TSDoc `/** comment */` style here on each prop. You can and are encouraged to leave descriptive comments on reusable components. For a fuller example and discussion, see our Commenting Components section in the Advanced Cheatsheet.

Useful React Prop Type Examples

```
export declare interface AppProps {
  children1: JSX.Element; // bad, doesnt account for arrays
  children2: JSX.Element | JSX.Element[]; // meh, doesn't accept strings
  children3: React.ReactChildren; // despite the name, not at all an appropriate type
  children4: React.ReactChild[]; // better
  children: React.ReactNode; // best, accepts everything
  functionChildren: (name: string) => React.ReactNode; // recommended function as a c
  style?: React.CSSProperties; // to pass through style props
  onChange?: React.FormEventHandler<HTMLInputElement>; // form events! the generic pa
  props: Props & React.PropsWithoutRef<JSX.IntrinsicElements["button"]>; // to impers
}
```

▶ **JSX.Element vs React.ReactNode?**

More discussion: Where ReactNode does not overlap with JSX.Element

Something to add? File an issue.

getDerivedStateFromProps

Before you start using `getDerivedStateFromProps`, please go through the documentation and You Probably Don't Need Derived State. Derived State can be easily achieved using hooks which can also help set up memoization easily.

Here are a few ways in which you can annotate `getDerivedStateFromProps`

1. If you have explicitly typed your derived state and want to make sure that the return value from `getDerivedStateFromProps` conforms to it.

```
class Comp extends React.Component<Props, State> {
  static getDerivedStateFromProps(
    props: Props,
    state: State
```

```
  ): Partial<State> | null {
    //
  }
}
```

2. When you want the function's return value to determine your state.

```
class Comp extends React.Component<
  Props,
  ReturnType<typeof Comp["getDerivedStateFromProps"]>
> {
  static getDerivedStateFromProps(props: Props) {}
}
```

3. When you want derived state with other state fields and memoization

```
type CustomValue = any;
interface Props {
  propA: CustomValue;
}
interface DefinedState {
  otherStateField: string;
}
type State = DefinedState & ReturnType<typeof transformPropsToState>;
function transformPropsToState(props: Props) {
  return {
    savedPropA: props.propA, // save for memoization
    derivedState: props.propA,
  };
}
class Comp extends React.PureComponent<Props, State> {
  constructor(props: Props) {
    super(props);
    this.state = {
      otherStateField: "123",
      ...transformPropsToState(props),
    };
  }
  static getDerivedStateFromProps(props: Props, state: State) {
    if (isEqual(props.propA, state.savedPropA)) return null;
    return transformPropsToState(props);
  }
}
```

View in the TypeScript Playground

Forms and Events

If performance is not an issue, inlining handlers is easiest as you can just use type inference and contextual typing:

```
const el = (
  <button
    onClick={(event) => {
      /* ... */
    }}
  />
);
```

But if you need to define your event handler separately, IDE tooling really comes in handy here, as the @type definitions come with a wealth of typing. Type what you are looking for and usually the autocomplete will help you out. Here is what it looks like for an `onChange` for a form event:

```
class App extends React.Component<
  {},
  {
    // no props
    text: string;
  }
> {
  state = {
    text: "",
  };

  // typing on RIGHT hand side of =
  onChange = (e: React.FormEvent<HTMLInputElement>): void => {
    this.setState({ text: e.currentTarget.value });
  };
  render() {
    return (
      <div>
        <input type="text" value={this.state.text} onChange={this.onChange} />
      </div>
    );
  }
}
```

View in the TypeScript Playground

Instead of typing the arguments and return values with `React.FormEvent<>` and `void`, you may alternatively apply types to the event handler itself (*contributed by @TomasHubelbauer*):

```
  // typing on LEFT hand side of =
  onChange: React.ChangeEventHandler<HTMLInputElement> = (e) => {
```

```
    this.setState({text: e.currentTarget.value})
  }
```

▶ **Why two ways to do the same thing?**

Typing onSubmit, with Uncontrolled components in a Form

If you don't quite care about the type of the event, you can just use React.SyntheticEvent. If your target form has custom named **input**s that you'd like to access, you can use type widening:

```
<form
  ref={formRef}
  onSubmit={(e: React.SyntheticEvent) => {
    e.preventDefault();
    const target = e.target as typeof e.target & {
      email: { value: string };
      password: { value: string };
    };
    const email = target.email.value; // typechecks!
    const password = target.password.value; // typechecks!
    // etc...
  }}
>
  <div>
    <label>
      Email:
      <input type="email" name="email" />
    </label>
  </div>
  <div>
    <label>
      Password:
      <input type="password" name="password" />
    </label>
  </div>
  <div>
    <input type="submit" value="Log in" />
  </div>
</form>
```

View in the TypeScript Playground

Of course, if you're making any sort of significant form, you should use Formik, which is written in TypeScript.

Context

Using `React.createContext` with an empty object as default value.

```
interface ContextState {
  // set the type of state you want to handle with context e.g.
  name: string | null;
}
//set an empty object as default state
const Context = React.createContext({} as ContextState);
// set up context provider as you normally would in JavaScript [React Context API](ht
```

Using `React.createContext` and context getters to make a `createCtx` with **no `defaultValue`, yet no need to check for `undefined`**:

```
import * as React from "react";

const currentUserContext = React.createContext<string | undefined>(undefined);

function EnthusasticGreeting() {
  const currentUser = React.useContext(currentUserContext);
  return <div>HELLO {currentUser!.toUpperCase()}!</div>;
}

function App() {
  return (
    <currentUserContext.Provider value="Anders">
      <EnthusasticGreeting />
    </currentUserContext.Provider>
  );
}
```

Notice the explicit type arguments which we need because we don't have a default `string` value:

```
const currentUserContext = React.createContext<string | undefined>(undefined);
//                                             ^^^^^^^^^^^^^^^^^^
```

along with the non-null assertion to tell TypeScript that `currentUser` is definitely going to be there:

```
return <div>HELLO {currentUser!.toUpperCase()}!</div>;
//                              ^
```

This is unfortunate because *we know* that later in our app, a `Provider` is going to fill in the context.

There are a few solutions for this:

1. You can get around this by asserting non null:

```
const currentUserContext = React.createContext<string>(undefined!);
```

(Playground here) This is a quick and easy fix, but this loses type-safety, and if you forget to supply a value to the Provider, you will get an error.

2. We can write a helper function called `createCtx` that guards against accessing a `Context` whose value wasn't provided. By doing this, API instead, **we never have to provide a default and never have to check for `undefined`**:

```
import * as React from "react";

/**
 * A helper to create a Context and Provider with no upfront default value, and
 * without having to check for undefined all the time.
 */
function createCtx<A extends {} | null>() {
  const ctx = React.createContext<A | undefined>(undefined);
  function useCtx() {
    const c = React.useContext(ctx);
    if (c === undefined)
      throw new Error("useCtx must be inside a Provider with a value");
    return c;
  }
  return [useCtx, ctx.Provider] as const; // 'as const' makes TypeScript infer a
}

// Usage:

// We still have to specify a type, but no default!
export const [useCurrentUserName, CurrentUserProvider] = createCtx<string>();

function EnthusasticGreeting() {
  const currentUser = useCurrentUserName();
  return <div>HELLO {currentUser.toUpperCase()}!</div>;
}

function App() {
  return (
    <CurrentUserProvider value="Anders">
      <EnthusasticGreeting />
    </CurrentUserProvider>
  );
}
```

View in the TypeScript Playground

3. You can go even further and combine this idea using `React.createContext` and context getters.

```
/**
 * A helper to create a Context and Provider with no upfront default value, and
 * without having to check for undefined all the time.
 */
function createCtx<A extends {} | null>() {
  const ctx = React.createContext<A | undefined>(undefined);
  function useCtx() {
    const c = React.useContext(ctx);
    if (c === undefined)
      throw new Error("useCtx must be inside a Provider with a value");
    return c;
  }
  return [useCtx, ctx.Provider] as const; // 'as const' makes TypeScript infer a
}

// usage

export const [useCtx, SettingProvider] = createCtx<string>(); // specify type, bu
export function App() {
  const key = useCustomHook("key"); // get a value from a hook, must be in a comp
  return (
    <SettingProvider value={key}>
      <Component />
    </SettingProvider>
  );
}
export function Component() {
  const key = useCtx(); // can still use without null check!
  return <div>{key}</div>;
}
```

View in the TypeScript Playground

4. Using `React.createContext` and `useContext` to make a `createCtx` with `unstated`-like context setters:

```
export function createCtx<A>(defaultValue: A) {
  type UpdateType = React.Dispatch<
    React.SetStateAction<typeof defaultValue>
  >;
  const defaultUpdate: UpdateType = () => defaultValue;
  const ctx = React.createContext({
    state: defaultValue,
    update: defaultUpdate,
  });
  function Provider(props: React.PropsWithChildren<{}>) {
    const [state, update] = React.useState(defaultValue);
    return <ctx.Provider value={{ state, update }} {...props} />;
  }
  return [ctx, Provider] as const; // alternatively, [typeof ctx, typeof Provider
}
```

```
// usage

const [ctx, TextProvider] = createCtx("someText");
export const TextContext = ctx;
export function App() {
  return (
    <TextProvider>
      <Component />
    </TextProvider>
  );
}
export function Component() {
  const { state, update } = React.useContext(TextContext);
  return (
    <label>
      {state}
      <input type="text" onChange={(e) => update(e.target.value)} />
    </label>
  );
}
```

View in the TypeScript Playground

5. A useReducer-based version may also be helpful.

▶ **Mutable Context Using a Class component wrapper**

Something to add? File an issue.

forwardRef/createRef

Check the Hooks section for `useRef`.

`createRef`:

```
class CssThemeProvider extends React.PureComponent<Props> {
  private rootRef = React.createRef<HTMLDivElement>(); // like this
  render() {
    return <div ref={this.rootRef}>{this.props.children}</div>;
  }
}
```

`forwardRef`:

```
type Props = { children: React.ReactNode; type: "submit" | "button" };
export type Ref = HTMLButtonElement;
export const FancyButton = React.forwardRef<Ref, Props>((props, ref) => (
  <button ref={ref} className="MyClassName" type={props.type}>
```

```
      {props.children}
    </button>
  ));
```

If you are grabbing the props of a component that forwards refs, use `ComponentPropsWithRef`.

More info: https://medium.com/@martin_hotell/react-refs-with-typescript-a32d56c4d315

You may also wish to do Conditional Rendering with `forwardRef`.

Something to add? File an issue.

Portals

Using `ReactDOM.createPortal`:

```
const modalRoot = document.getElementById("modal-root") as HTMLElement;
// assuming in your html file has a div with id 'modal-root';

export class Modal extends React.Component {
  el: HTMLElement = document.createElement("div");

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(this.props.children, this.el);
  }
}
```

View in the TypeScript Playground

▶ **Context of Example**

Error Boundaries

Not written yet.

Something to add? File an issue.

Concurrent React/React Suspense

Not written yet. watch https://github.com/sw-yx/fresh-async-react for more on React Suspense and Time Slicing.

Something to add? File an issue.

Troubleshooting Handbook: Types

> ⚠ Have you read the TypeScript FAQ?) Your answer might be there!

Facing weird type errors? You aren't alone. This is the hardest part of using TypeScript with React. Be patient - you are learning a new language after all. However, the more you get good at this, the less time you'll be working *against* the compiler and the more the compiler will be working *for* you!

Try to avoid typing with `any` as much as possible to experience the full benefits of typescript. Instead, let's try to be familiar with some of the common strategies to solve these issues.

Union Types and Type Guarding

Union types are handy for solving some of these typing problems:

```
class App extends React.Component<
  {},
  {
    count: number | null; // like this
  }
> {
  state = {
    count: null,
  };
  render() {
    return <div onClick={() => this.increment(1)}>{this.state.count}</div>;
  }
  increment = (amt: number) => {
    this.setState((state) => ({
      count: (state.count || 0) + amt,
    }));
  };
}
```

View in the TypeScript Playground

Type Guarding: Sometimes Union Types solve a problem in one area but create another downstream. If `A` and `B` are both object types, `A | B` isn't "either A or B", it is "A or B or both at once", which causes some confusion if you expected it to be the former. Learn how to write checks, guards, and assertions (also see the Conditional Rendering section below). For example:

```
interface Admin {
  role: string;
}
interface User {
  email: string;
}

// Method 1: use `in` keyword
function redirect(user: Admin | User) {
  if ("role" in user) {
    // use the `in` operator for typeguards since TS 2.7+
    routeToAdminPage(user.role);
  } else {
    routeToHomePage(user.email);
  }
}

// Method 2: custom type guard, does the same thing in older TS versions or where `in
function isAdmin(user: Admin | User): user is Admin {
  return (user as any).role !== undefined;
}
```

View in the TypeScript Playground

Method 2 is also known as User-Defined Type Guards and can be really handy for readable code. This is how TS itself refines types with `typeof` and `instanceof`.

If you need `if...else` chains or the `switch` statement instead, it should "just work", but look up Discriminated Unions if you need help. (See also: Basarat's writeup). This is handy in typing reducers for `useReducer` or Redux.

Optional Types

If a component has an optional prop, add a question mark and assign during destructure (or use defaultProps).

```
class MyComponent extends React.Component<{
  message?: string; // like this
}> {
  render() {
    const { message = "default" } = this.props;
```

```
    return <div>{message}</div>;
  }
}
```

You can also use a `!` character to assert that something is not undefined, but this is not encouraged.

Something to add? File an issue with your suggestions!

Enum Types

Enums in TypeScript default to numbers. You will usually want to use them as strings instead:

```
export enum ButtonSizes {
  default = "default",
  small = "small",
  large = "large",
}
```

Usage:

```
export const PrimaryButton = (
  props: Props & React.HTMLProps<HTMLButtonElement>
) => <Button size={ButtonSizes.default} {...props} />;
```

A simpler alternative to enum is just declaring a bunch of strings with union:

```
export declare type Position = "left" | "right" | "top" | "bottom";
```

This is handy because TypeScript will throw errors when you mistype a string for your props.

Type Assertion

Sometimes you know better than TypeScript that the type you're using is narrower than it thinks, or union types need to be asserted to a more specific type to work with other APIs, so assert with the `as` keyword. This tells the compiler you know better than it does.

```
class MyComponent extends React.Component<{
  message: string;
}> {
  render() {
    const { message } = this.props;
    return (
```

```
        <Component2 message={message as SpecialMessageType}>{message}</Component2>
      );
    }
  }
```

View in the TypeScript Playground

Note that you cannot assert your way to anything - basically it is only for refining types. Therefore it is not the same as "casting" a type.

You can also assert a property is non-null, when accessing it:

```
element.parentNode!.removeChild(element) // ! before the period
myFunction(document.getElementById(dialog.id!)! // ! after the property accessing
let userID!: string // definite assignment assertion... be careful!
```

Of course, try to actually handle the null case instead of asserting :)

Simulating Nominal Types

TS' structural typing is handy, until it is inconvenient. However you can simulate nominal typing with `type branding`:

```
type OrderID = string & { readonly brand: unique symbol };
type UserID = string & { readonly brand: unique symbol };
type ID = OrderID | UserID;
```

We can create these values with the Companion Object Pattern:

```
function OrderID(id: string) {
  return id as OrderID;
}
function UserID(id: string) {
  return id as UserID;
}
```

Now TypeScript will disallow you from using the wrong ID in the wrong place:

```
function queryForUser(id: UserID) {
  // ...
}
queryForUser(OrderID("foobar")); // Error, Argument of type 'OrderID' is not assignab
```

In future you can use the `unique` keyword to brand. See this PR.

Intersection Types

Adding two types together can be handy, for example when your component is supposed to mirror the props of a native component like a `button`:

```
export interface Props {
  label: string;
}
export const PrimaryButton = (
  props: Props & React.HTMLProps<HTMLButtonElement> // adding my Props together with
) => <Button {...props} />;
```

You can also use Intersection Types to make reusable subsets of props for similar components:

```
type BaseProps = {
   className?: string,
   style?: React.CSSProperties
   name: string // used in both
}
type DogProps = {
  tailsCount: number
}
type HumanProps = {
  handsCount: number
}
export const Human: React.FC<BaseProps & HumanProps> = // ...
export const Dog: React.FC<BaseProps & DogProps> = // ...
```

View in the TypeScript Playground

Make sure not to confuse Intersection Types (which are **and** operations) with Union Types (which are **or** operations).

Union Types

This section is yet to be written (please contribute!). Meanwhile, see our commentary on Union Types usecases.

The ADVANCED cheatsheet also has information on Discriminated Union Types, which are helpful when TypeScript doesn't seem to be narrowing your union type as you expect.

Overloading Function Types

Specifically when it comes to functions, you may need to overload instead of union type. The most common way function types are written uses the shorthand:

```
type FunctionType1 = (x: string, y: number) => number;
```

But this doesn't let you do any overloading. If you have the implementation, you can put them after each other with the function keyword:

```
function pickCard(x: { suit: string; card: number }[]): number;
function pickCard(x: number): { suit: string; card: number };
function pickCard(x): any {
  // implementation with combined signature
  // ...
}
```

However, if you don't have an implementation and are just writing a `.d.ts` definition file, this won't help you either. In this case you can forego any shorthand and write them the old-school way. The key thing to remember here is as far as TypeScript is concerned, `functions are just callable objects with no key`:

```
type pickCard = {
  (x: { suit: string; card: number }[]): number;
  (x: number): { suit: string; card: number };
  // no need for combined signature in this form
  // you can also type static properties of functions here eg `pickCard.wasCalled`
};
```

Note that when you implement the actual overloaded function, the implementation will need to declare the combined call signature that you'll be handling, it won't be inferred for you. You can see readily see examples of overloads in DOM APIs, e.g. `createElement`.

Read more about Overloading in the Handbook.

Using Inferred Types

Leaning on TypeScript's Type Inference is great... until you realize you need a type that was inferred, and have to go back and explicitly declare types/interfaces so you can export them for reuse.

Fortunately, with `typeof`, you won't have to do that. Just use it on any value:

```
const [state, setState] = React.useState({
  foo: 1,
  bar: 2,
}); // state's type inferred to be {foo: number, bar: number}
```

```
const someMethod = (obj: typeof state) => {
  // grabbing the type of state even though it was inferred
  // some code using obj
  setState(obj); // this works
};
```

Using Partial Types

Working with slicing state and props is common in React. Again, you don't really have to go and explicitly redefine your types if you use the `Partial` generic type:

```
const [state, setState] = React.useState({
  foo: 1,
  bar: 2,
}); // state's type inferred to be {foo: number, bar: number}

// NOTE: stale state merging is not actually encouraged in React.useState
// we are just demonstrating how to use Partial here
const partialStateUpdate = (obj: Partial<typeof state>) =>
  setState({ ...state, ...obj });

// later on...
partialStateUpdate({ foo: 2 }); // this works
```

▶ Minor caveats on using `Partial`

The Types I need weren't exported!

This can be annoying but here are ways to grab the types!

- Grabbing the Prop types of a component: Use `React.ComponentProps` and `typeof`, and optionally `Omit` any overlapping types

```
import { Button } from "library"; // but doesn't export ButtonProps! oh no!
type ButtonProps = React.ComponentProps<typeof Button>; // no problem! grab your own!
type AlertButtonProps = Omit<ButtonProps, "onClick">; // modify
const AlertButton: React.FC<AlertButtonProps> = (props) => (
  <Button onClick={() => alert("hello")} {...props} />
);
```

You may also use `ComponentPropsWithoutRef` (instead of ComponentProps) and `ComponentPropsWithRef` (if your component specifically forwards refs)

- Grabbing the return type of a function: use `ReturnType`:

```
// inside some library - return type { baz: number } is inferred but not exported
function foo(bar: string) {
  return { baz: 1 };
}

//  inside your app, if you need { baz: number }
type FooReturn = ReturnType<typeof foo>; // { baz: number }
```

In fact you can grab virtually anything public: see this blogpost from Ivan Koshelev

```
function foo() {
  return {
    a: 1,
    b: 2,
    subInstArr: [
      {
        c: 3,
        d: 4,
      },
    ],
  };
}

type InstType = ReturnType<typeof foo>;
type SubInstArr = InstType["subInstArr"];
type SubIsntType = SubInstArr[0];

let baz: SubIsntType = {
  c: 5,
  d: 6, // type checks ok!
};

//You could just write a one-liner,
//But please make sure it is forward-readable
//(you can understand it from reading once left-to-right with no jumps)
type SubIsntType2 = ReturnType<typeof foo>["subInstArr"][0];
let baz2: SubIsntType2 = {
  c: 5,
  d: 6, // type checks ok!
};
```

- TS also ships with a `Parameters` utility type for extracting the parameters of a function
- for anything more "custom", the `infer` keyword is the basic building block for this, but takes a bit of getting used to. Look at the source code for the above utility types, and this example to get the idea. Basarat also has a good video on `infer`.

The Types I need don't exist!

What's more annoying than modules with unexported types? Modules that are **untyped**!

Fret not! There are more than a couple of ways in which you can solve this problem.

A **lazier** way would be to create a new type declaration file, say `typedec.d.ts` – if you don't already have one. Ensure that the path to file is resolvable by TypeScript by checking the `include` array in the `tsconfig.json` file at the root of your directory.

```
// inside tsconfig.json
{
  // ...
  "include": [
    "src" // automatically resolves if the path to declaration is src/typedec.d.ts
  ]
  // ...
}
```

Within this file, add the `declare` syntax for your desired module, say `my-untyped-module` – to the declaration file:

```
// inside typedec.d.ts
declare module "my-untyped-module";
```

This one-liner alone is enough if you just need it to work without errors. A even hackier, write-once-and-forget way would be to use `"*"` instead which would then apply the `Any` type for all existing and future untyped modules.

This solution works well as a workaround if you have less than a couple untyped modules. Anything more, you now have a ticking type-bomb in your hands. The only way of circumventing this problem would be to define the missing types for those untyped modules as explained in the following sections.

Typing Exported Hooks

Typing Hooks is just like typing pure functions.

The following steps work under two assumptions:

- You have already created a type declaration file as stated earlier in the section.
- You have access to the source code - specifically the code that directly exports the functions you will be using. In most cases, it would be housed in an `index.js` file. Typically you need a minimum of **two** type declarations (one for **Input Prop** and the other for **Return Prop**) to define a hook completely. Suppose the hook you wish to type follows the following structure,

```
// ...
const useUntypedHook = (prop) => {
  // some processing happens here
```

```
  return {
    /* ReturnProps */
  };
};
export default useUntypedHook;
```

then, your type declaration should most likely follow the following syntax.

```
declare module 'use-untyped-hook' {
  export interface InputProps { ... }   // type declaration for prop
  export interface ReturnProps { ... } // type declaration for return props
  export default function useUntypedHook(
    prop: InputProps
    // ...
  ): ReturnProps;
}
```

▶

For instance, the useDarkMode hook exports the functions that follows a similar structure.

Typing Exported Components

In case of typing untyped class components, there's almost no difference in approach except for the fact that after declaring the types, you export the extend the type using `class UntypedClassComponent extends React.Component<UntypedClassComponentProps, any> {}` where `UntypedClassComponentProps` holds the type declaration.

For instance, sw-yx's Gist on React Router 6 types implemented a similar method for typing the then untyped RR6.

```
declare module "react-router-dom" {
  import * as React from 'react';
  // ...
  type NavigateProps<T> = {
    to: string | number,
    replace?: boolean,
    state?: T
  }
  //...
  export class Navigate<T = any> extends React.Component<NavigateProps<T>>{}
  // ...
```

For more information on creating type definitions for class components, you can refer to this post for reference.

Troubleshooting Handbook: Images and other non-TS/TSX files

What's more annoying than modules with unexported types? Modules that are **untyped**!

Fret not! There are more than a couple of ways in which you can solve this problem.

A **lazier** way would be to create a new type declaration file, say `typedec.d.ts` – if you don't already have one. Ensure that the path to file is resolvable by TypeScript by checking the `include` array in the `tsconfig.json` file at the root of your directory.

```
// inside tsconfig.json
{
  // ...
  "include": [
    "src" // automatically resolves if the path to declaration is src/typedec.d.ts
  ]
  // ...
}
```

Within this file, add the `declare` syntax for your desired module, say `my-untyped-module` – to the declaration file:

```
// inside typedec.d.ts
declare module "my-untyped-module";
```

This one-liner alone is enough if you just need it to work without errors. A even hackier, write-once-and-forget way would be to use `"*"` instead which would then apply the `Any` type for all existing and future untyped modules.

This solution works well as a workaround if you have less than a couple untyped modules. Anything more, you now have a ticking type-bomb in your hands. The only way of circumventing this problem would be to define the missing types for those untyped modules as explained in the following sections.

Typing Exported Hooks

Typing Hooks is just like typing pure functions.

The following steps work under two assumptions:

- You have already created a type declaration file as stated earlier in the section.
- You have access to the source code - specifically the code that directly exports the functions you will be using. In most cases, it would be housed in an `index.js` file.

Typically you need a minimum of **two** type declarations (one for **Input Prop** and the other for **Return Prop**) to define a hook completely. Suppose the hook you wish to type follows the following structure,

```
// ...
const useUntypedHook = (prop) => {
  // some processing happens here
  return {
    /* ReturnProps */
  };
};
export default useUntypedHook;
```

then, your type declaration should most likely follow the following syntax.

```
declare module 'use-untyped-hook' {
  export interface InputProps { ... }   // type declaration for prop
  export interface ReturnProps { ... } // type declaration for return props
  export default function useUntypedHook(
    prop: InputProps
    // ...
  ): ReturnProps;
}
```

▶

For instance, the useDarkMode hook exports the functions that follows a similar structure.

Typing Exported Components

In case of typing untyped class components, there's almost no difference in approach except for the fact that after declaring the types, you export the extend the type using `class UntypedClassComponent extends React.Component<UntypedClassComponentProps, any> {}` where `UntypedClassComponentProps` holds the type declaration.

For instance, sw-yx's Gist on React Router 6 types implemented a similar method for typing the then untyped RR6.

```
declare module "react-router-dom" {
  import * as React from 'react';
  // ...
  type NavigateProps<T> = {
    to: string | number,
    replace?: boolean,
    state?: T
  }
  //...
  export class Navigate<T = any> extends React.Component<NavigateProps<T>>{}
  // ...
```

For more information on creating type definitions for class components, you can refer to this post for reference.

Troubleshooting Handbook: Images and other non-TS/TSX files

Use declaration merging:

Troubleshooting Handbook: Operators

- `typeof` and `instanceof` : type query used for refinement
- `keyof` : get keys of an object
- `O[K]` : property lookup
- `[K in O]` : mapped types
- `+` or `-` or `readonly` or `?` : addition and subtraction and readonly and optional modifiers
- `x ? Y : Z` : Conditional types for generic types, type aliases, function parameter types
- `!` : Nonnull assertion for nullable types
- `=` : Generic type parameter default for generic types
- `as` : type assertion
- `is` : type guard for function return types

Conditional Types are a difficult topic to get around so here are some extra resources:

- fully walked through explanation https://artsy.github.io/blog/2018/11/21/conditional-types-in-typescript/
- Bailing out and other advanced topics https://github.com/sw-yx/ts-spec/blob/master/conditional-types.md
- Basarat's video https://www.youtube.com/watch?v=SbVgPQDealg&list=PLYvdvJlnTOjF6aJsWWAt7kZRJvzw-en8B&index=2&t=0s

Troubleshooting Handbook: Utilities

these are all built in, see source in es5.d.ts:

- `ConstructorParameters` : a tuple of class constructor's parameter types
- `Exclude` : exclude a type from another type
- `Extract` : select a subtype that is assignable to another type

- `InstanceType` : the instance type you get from a `new` ing a class constructor
- `NonNullable` : exclude `null` and `undefined` from a type
- `Parameters` : a tuple of a function's parameter types
- `Partial` : Make all properties in an object optional
- `Readonly` : Make all properties in an object readonly
- `ReadonlyArray` : Make an immutable array of the given type
- `Pick` : A subtype of an object type with a subset of its keys
- `Record` : A map from a key type to a value type
- `Required` : Make all properties in an object required
- `ReturnType` A function's return type

This section needs writing, but you can probably find a good starting point with Wes Bos' ESLint config (which comes with a YouTube intro).

Troubleshooting Handbook: tsconfig.json

You can find all the Compiler options in the TypeScript docs. The new TS docs also has per-flag annotations of what each does. This is the setup I roll with for APPS (not libraries - for libraries you may wish to see the settings we use in `tsdx`):

```
{
  "compilerOptions": {
    "incremental": true,
    "outDir": "build/lib",
    "target": "es5",
    "module": "esnext",
    "lib": ["dom", "esnext"],
    "sourceMap": true,
    "importHelpers": true,
    "declaration": true,
    "rootDir": "src",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "allowJs": false,
    "jsx": "react",
    "moduleResolution": "node",
    "baseUrl": "src",
    "forceConsistentCasingInFileNames": true,
    "esModuleInterop": true,
    "suppressImplicitAnyIndexErrors": true,
    "allowSyntheticDefaultImports": true,
    "experimentalDecorators": true
  },
```

```
    "include": ["src/**/*"],
    "exclude": ["node_modules", "build", "scripts"]
  }
```

Please open an issue and discuss if there are better recommended choices for React.

Selected flags and why we like them:

- `esModuleInterop` : disables namespace imports (`import * as foo from "foo"`) and enables CJS/AMD/UMD style imports (`import fs from "fs"`)
- `strict` : `strictPropertyInitialization` forces you to initialize class properties or explicitly declare that they can be undefined. You can opt out of this with a definite assignment assertion.
- `"typeRoots": ["./typings", "./node_modules/@types"]` : By default, TypeScript looks in `node_modules/@types` and parent folders for third party type declarations. You may wish to override this default resolution so you can put all your global type declarations in a special `typings` folder.

Compilation speed grows linearly with size of codebase. For large projects, you will want to use Project References. See our ADVANCED cheatsheet for commentary.

Troubleshooting Handbook: Bugs in official typings

If you run into bugs with your library's official typings, you can copy them locally and tell TypeScript to use your local version using the "paths" field. In your `tsconfig.json` :

```
{
  "compilerOptions": {
    "paths": {
      "mobx-react": ["../typings/modules/mobx-react"]
    }
  }
}
```

Thanks to @adamrackis for the tip.

If you just need to add an interface, or add missing members to an existing interface, you don't need to copy the whole typing package. Instead, you can use declaration merging:

```
// my-typings.ts
declare module "plotly.js" {
  interface PlotlyHTMLElement {
    removeAllListeners(): void;
```

```
  }
}

// MyComponent.tsx
import { PlotlyHTMLElement } from "plotly.js";

const f = (e: PlotlyHTMLElement) => {
  e.removeAllListeners();
};
```

You dont always have to implement the module, you can simply import the module as `any` for a quick start:

```
// my-typings.ts
declare module "plotly.js"; // each of its imports are `any`
```

Because you don't have to explicitly import this, this is known as an ambient module declaration. You can do AMD's in a script-mode `.ts` file (no imports or exports), or a `.d.ts` file anywhere in your project.

You can also do ambient variable and ambient type declarations:

```
// ambient utiltity type
type ToArray<T> = T extends unknown[] ? T : T[];
// ambient variable
declare let process: {
  env: {
    NODE_ENV: "development" | "production";
  };
};
process = {
  env: {
    NODE_ENV: "production",
  },
};
```

You can see examples of these included in the built in type declarations in the `lib` field of `tsconfig.json`

Troubleshooting Handbook: Images and other non-TS/TSX files

Use declaration merging:

```
// declaration.d.ts
// anywhere in your project, NOT the same name as any of your .ts/tsx files
declare module "*.png";

// importing in a tsx file
import * as logo from "./logo.png";
```

Note that `tsc` cannot bundle these files for you, you will have to use Webpack or Parcel.

Related issue: https://github.com/Microsoft/TypeScript-React-Starter/issues/12 and StackOverflow

Recommended React + TypeScript codebases to learn from

- https://github.com/jaredpalmer/formik
- https://github.com/jaredpalmer/react-fns
- https://github.com/palantir/blueprint
- https://github.com/Shopify/polaris
- https://github.com/NullVoxPopuli/react-vs-ember/tree/master/testing/react
- https://github.com/artsy/reaction
- https://github.com/benawad/codeponder (with coding livestream!)
- https://github.com/artsy/emission (React Native)
- @reach/ui's community typings

React Boilerplates:

- https://github.com/rwieruch/nextjs-firebase-authentication: Next.js + Firebase Starter: styled, tested, typed, and authenticated
- @jpavon/react-scripts-ts alternative react-scripts with all TypeScript features using ts-loader
- webpack config tool is a visual tool for creating webpack projects with React and TypeScript
- https://github.com/innFactory/create-react-app-material-typescript-redux ready to go template with Material-UI, routing and Redux

React Native Boilerplates: *contributed by @spoeck*

- https://github.com/GeekyAnts/react-native-seed
- https://github.com/lopezjurip/ReactNativeTS
- https://github.com/emin93/react-native-template-typescript

- https://github.com/Microsoft/TypeScript-React-Native-Starter

Editor Tooling and Integration

- VSCode
  - swyx's VSCode Extension: https://github.com/sw-yx/swyx-react-typescript-snippets
  - amVim: https://marketplace.visualstudio.com/items?itemName=auiworks.amvim
- VIM
  - https://github.com/Quramy/tsuquyomi
  - nvim-typescript?
  - https://github.com/leafgarland/typescript-vim
  - peitalin/vim-jsx-typescript
  - NeoVim: https://github.com/neoclide/coc.nvim
  - other discussion: https://mobile.twitter.com/ryanflorence/status/1085715595994095620

Linting

> ⚠ Note that TSLint is now in maintenance and you should try to use ESLint instead. If you are interested in TSLint tips, please check this PR from @azdanov. The rest of this section just focuses on ESLint. You can convert TSlint to ESlint with this tool.

> ⚠ This is an evolving topic. `typescript-eslint-parser` is no longer maintained and work has recently begun on `typescript-eslint` in the ESLint community to bring ESLint up to full parity and interop with TSLint.

Follow the TypeScript + ESLint docs at https://github.com/typescript-eslint/typescript-eslint:

```
yarn add -D @typescript-eslint/eslint-plugin @typescript-eslint/parser eslint
```

add a `lint` script to your `package.json`:

```
  "scripts": {
    "lint": "eslint 'src/**/*.ts'"
  },
```

and a suitable `.eslintrc.js` (using `.js` over `.json` here so we can add comments):

```
module.exports = {
  env: {
    es6: true,
```

```
    node: true,
    jest: true,
  },
  extends: "eslint:recommended",
  parser: "@typescript-eslint/parser",
  plugins: ["@typescript-eslint"],
  parserOptions: {
    ecmaVersion: 2017,
    sourceType: "module",
  },
  rules: {
    indent: ["error", 2],
    "linebreak-style": ["error", "unix"],
    quotes: ["error", "single"],
    "no-console": "warn",
    "no-unused-vars": "off",
    "@typescript-eslint/no-unused-vars": [
      "error",
      { vars: "all", args: "after-used", ignoreRestSiblings: false },
    ],
    "@typescript-eslint/explicit-function-return-type": "warn", // Consider using exp
    "no-empty": "warn",
  },
};
```

Most of this is taken from the `tsdx` PR which is for **libraries**.

More `.eslintrc.json` options to consider with more options you may want for **apps**:

```
{
  "extends": [
    "airbnb",
    "prettier",
    "prettier/react",
    "plugin:prettier/recommended",
    "plugin:jest/recommended",
    "plugin:unicorn/recommended"
  ],
  "plugins": ["prettier", "jest", "unicorn"],
  "parserOptions": {
    "sourceType": "module",
    "ecmaFeatures": {
      "jsx": true
    }
  },
  "env": {
    "es6": true,
    "browser": true,
    "jest": true
  },
  "settings": {
    "import/resolver": {
```

```
      "node": {
        "extensions": [".js", ".jsx", ".ts", ".tsx"]
      }
    }
  },
  "overrides": [
    {
      "files": ["**/*.ts", "**/*.tsx"],
      "parser": "typescript-eslint-parser",
      "rules": {
        "no-undef": "off"
      }
    }
  ]
}
```

You can read a fuller TypeScript + ESLint setup guide here from Matterhorn, in particular check https://github.com/MatterhornDev/learn-typescript-linting.

Another great resource is "Using ESLint and Prettier in a TypeScript Project" by @robertcoopercode.

If you're looking for information on Prettier, check out the Prettier.

Other React + TypeScript resources

- me! https://twitter.com/swyx
- https://github.com/piotrwitek/react-redux-typescript-guide - **HIGHLY HIGHLY RECOMMENDED**, i wrote this repo before knowing about this one, this has a lot of stuff I don't cover, including **REDUX** and **JEST**.
- Ultimate React Component Patterns with TypeScript 2.8
- Basarat's TypeScript gitbook has a React section with an Egghead.io course as well.
- Palmer Group's TypeScript + React Guidelines as well as Jared's other work like disco.chat
- Stefan Baumgartner's TypeScript + React Guide, which serves as a side-by-side guide to the official docs with extra articles on styling, custom hooks and patterns
- Sindre Sorhus' TypeScript Style Guide
- TypeScript React Starter Template by Microsoft A starter template for TypeScript and React with a detailed README describing how to use the two together. Note: this doesnt seem to be frequently updated anymore.
- Brian Holt's Intermediate React course on Frontend Masters (paid) - Converting App To TypeScript Section
- TypeScript conversion:
  - Lyft's React-To-TypeScript conversion CLI

    - Gustav Wengel's blogpost - converting a React codebase to TypeScript
    - Microsoft React TypeScript conversion guide
- DefinitelyTyped React source code
- You?.

Recommended React + TypeScript talks

- Ultimate React Component Patterns with TypeScript, by Martin Hochel, GeeCon Prague 2018
- Please help contribute this new section!

Time to Really Learn TypeScript

Believe it or not, we have only barely introduced TypeScript here in this cheatsheet. There is a whole world of generic type logic that you will eventually get into, however it becomes far less dealing with React than just getting good at TypeScript so it is out of scope here. But at least you can get productive in React now :)

It is worth mentioning some resources to help you get started:

- Step through the 40+ examples under the playground's Examples section, written by @Orta
- Anders Hejlsberg's overview of TS: https://www.youtube.com/watch?v=ET4kT88JRXs
- Marius Schultz: https://blog.mariusschulz.com/series/typescript-evolution with an Egghead.io course
- Basarat's Deep Dive: https://basarat.gitbook.io/typescript/
- Rares Matei: Egghead.io course's advanced TypeScript course on Egghead.io is great for newer typescript features and practical type logic applications (e.g. recursively making all properties of a type `readonly`)
- Go through Remo Jansen's TypeScript ladder
- Shu Uesugi: TypeScript for Beginner Programmers

Example App

- Create React App TypeScript Todo Example 2020

My question isn't answered here!

- File an issue.

Contributors

This project follows the all-contributors specification. See CONTRIBUTORS.md for the full list. Contributions of any kind welcome!

Releases

No releases published

Packages

No packages published

Contributors 87

+ 76 contributors

Languages

JavaScript 96.9% CSS 3.1%